

Logiciel de Base

Ensimag 1A Apprentissage

Examen — Juin 2011

Consignes :

- Durée : 2h.
- Tous documents autorisés.
- Le barème est donné à titre indicatif.
- On attend des réponses courtes et pertinentes, inutile de recopier le cours.
- Les parties sont indépendantes les unes des autres. La plupart des questions du problème sont également indépendantes. Pensez à lire le sujet en entier avant de commencer à répondre.

Consignes relatives à l'écriture de code C et assembleur Pentium :

- Pour chaque question, une partie des points sera affectée à la clarté du code et au respect des consignes ci-dessous.
- Pour les questions portant sur la traduction d'une fonction C en assembleur, on demande d'indiquer en commentaire chaque ligne du programme C original avant d'écrire les instructions assembleur correspondantes.
- Pour améliorer la lisibilité du code assembleur, il est conseillé d'utiliser des constantes (i.e. déclarations du type `x=42`) pour les déplacements relatifs à `%ebp` (i.e. paramètres des fonctions et variables locales). Par exemple, si une variable locale s'appelle `var` en langage C, on y fera référence avec `var(%ebp)`.
- Sauf indication contraire dans l'énoncé, on demande de traduire le code C en assembleur de façon systématique, sans chercher à faire la moindre optimisation : en particulier, **on stockera les variables locales dans la pile** (pas dans des registres), comme le fait le compilateur C par défaut.
- On respectera les conventions de gestion des registres Intel vues en cours, c'est à dire :
 - `%eax`, `%ecx` et `%edx` sont des registres scratch ;
 - `%ebx`, `%esi` et `%edi` ne sont pas des registres scratch.

1 Exercices sur le langage d'assemblage et GDB

On considère le programme assembleur suivant :

```
.data
fmt:      .asciz "%d\n"
.text
```

```

plus_one:
    pushl %ebp
    movl %esp, %ebp

debut_plus_one:
    movl 8(%ebp), %eax
    addl $1, %eax

    leave
apres_leave:
    ret

    .globl main
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp

    pushl $42
    call plus_one
    addl $4, %esp

apres_plus_one:
    pushl %eax
    pushl $fmt
    call printf
    addl $8, %esp

    leave
    ret

```

On assemble ce programme et on l'exécute dans GDB. Une trace incomplète est donnée ci-dessous :

```

(gdb) break debut_plus_one
Breakpoint 1 at 0x80483ca: file plus_one.S, line 11.
(gdb) run
Starting program: plus_one
# Afficher le contenu de la mémoire, en hexa et par mot long
# en partant du pointeur %esp
(gdb) x/16x $esp
0xbfffe514:    0xbfffe528    0x080483dc    0x_____    0x08048410
0xbfffe524:    0x00000000    0xbfffe5a8    0xb7eafca6    0x00000001
0xbfffe534:    0xbfffe5d4    0xbfffe5dc    0xb7fe1bd8    0xbfffe590
0xbfffe544:    0xffffffff    0xb7ffe1f4    0x08048234    0x00000001

```

```

# Afficher le contenu du registre %esp
(gdb) print $esp
$1 = (void *) 0xbfffe510
(gdb) print $ebp
$2 = (void *) 0xbfffe514
(gdb) print $eax
$3 = -1073748524
(gdb) break apres_leave
Breakpoint 2 at 0x80483d1: file plus_one.S, line 16.
(gdb) continue
Continuing.
Breakpoint 2, apres_leave () at plus_one.S:16
16          ret
(gdb) print $esp
$4 = (void *) 0x_____
(gdb) print $ebp
$5 = (void *) 0x_____
(gdb) break apres_plus_one
Breakpoint 3 at 0x80483e2: file plus_one.S, line 30.
(gdb) continue
Continuing.
Breakpoint 3, apres_plus_one () at plus_one.S:30
30          movl %eax, 4(%esp)
(gdb) print $esp
$6 = (void *) 0x_____
(gdb) print $ebp
$7 = (void *) 0x_____
(gdb) next
(gdb) next
32          call printf
(gdb) x/16x $esp
0xbfffe520:    0x080495cc    0x0000002b    0xbfffe5a8    0xb7eafca6
0xbfffe530:    0x00000001    0xbfffe5d4    0xbfffe5dc    0xb7fe1bd8
0xbfffe540:    0xbfffe590    0xffffffff    0xb7ffe5ff4    0x08048234
0xbfffe550:    0x00000001    0xbfffe590    0xb7ff0966    0xb7fffab0
(gdb) print /x &fmt
$8 = 0x80495cc

```

Question 1 (2 points) 6 valeurs ont été remplacées par des _____. Donnez ces 6 valeurs, avec pour chacune une explication d'une ou deux phrases.

```

(gdb) break debut_plus_one
Breakpoint 1 at 0x80483ca: file plus_one.S, line 11.
(gdb) run

```

```

Starting program: plus_one
(gdb) x/16x $esp
0xbffff514:      0xbffff528      0x080483dc      0x0000002a      0x08048410
0xbffff524:      0x00000000      0xbffff5a8      0xb7eafca6      0x00000001
0xbffff534:      0xbffff5d4      0xbffff5dc      0xb7fe1bd8      0xbffff590
0xbffff544:      0xffffffff      0xb7ffeff4      0x08048234      0x00000001
(gdb) print $esp
$1 = (void *) 0xbffff514
(gdb) print $ebp
$2 = (void *) 0xbffff514
(gdb) print $eax
$3 = -1073748524
(gdb) break apres_leave
Breakpoint 2 at 0x80483d1: file plus_one.S, line 16.
(gdb) continue
Continuing.
Breakpoint 2, apres_leave () at plus_one.S:16
16          ret
(gdb) print $esp
$4 = (void *) 0xbffff518
(gdb) print $ebp
$5 = (void *) 0xbffff528
(gdb) break apres_plus_one
Breakpoint 3 at 0x80483e2: file plus_one.S, line 30.
(gdb) continue
Continuing.
Breakpoint 3, apres_plus_one () at plus_one.S:30
30          movl %eax, 4(%esp)
(gdb) print $esp
$6 = (void *) 0xbffff520
(gdb) print $ebp
$7 = (void *) 0xbffff528
(gdb) next
31          movl $fmt, 0(%esp)
(gdb) next
32          call printf
(gdb) x/16x $esp
0xbffff520:      0x080495cc      0x0000002b      0xbffff5a8      0xb7eafca6
0xbffff530:      0x00000001      0xbffff5d4      0xbffff5dc      0xb7fe1bd8
0xbffff540:      0xbffff590      0xffffffff      0xb7ffeff4      0x08048234
0xbffff550:      0x00000001      0xbffff590      0xb7ff0966      0xb7fffab0
(gdb) print /x &fmt
$8 = 0x80495cc

```

2 Implémentation d'une fonction simple en assembleur

Soit la fonction C suivante :

```
void incrementer(int source, int *destination) {
    *destination = source + 1;
}
```

Question 2 (1 point) *Traduire cette fonction en assembleur.*

```
source=8
destination=12
.globl incrementer
incrementer:
    pushl %ebp
    movl %esp, %ebp

    movl source(%ebp), %eax
    addl $1, %eax
    movl destination(%ebp), %ecx
    movl %eax, (%ecx)

    leave
    ret
```

3 Liste chaînée

Soit une liste chaînée définie par :

```
struct cellule {
    int val;
    struct cellule *suiv;
};
```

Question 3 (1 point) *Écrire en C une fonction compter_elem_liste(...) (la plus simple possible) qui compte le nombre d'éléments de cette liste.*

```
// Passage par valeur :
// on modifie les elements, pas les pointeurs.
int incrementer_liste(struct cellule *l)
{
    int res = 0;
```

```

        while (l != NULL) {
            res++;
            l = l->suiv;
        }
        return res;
    }
}
4

```

4 Problème : Implémentation d'une structure de donnée représentant une chaîne

En C, il n'y a pas à proprement parler de type « string », on utilise à la place le type « char * ». La convention est qu'une chaîne de caractères est terminée par le caractère '\0'.

Un inconvénient de cette représentation est que l'algorithme pour compter le nombre d'éléments d'une chaîne (`strlen`) est en $O(n)$ puisqu'il doit parcourir toute la chaîne pour trouver le '\0' final.

Dans cet exercice, nous allons utiliser une autre solution : stocker explicitement la taille de la chaîne dans une structure de données.

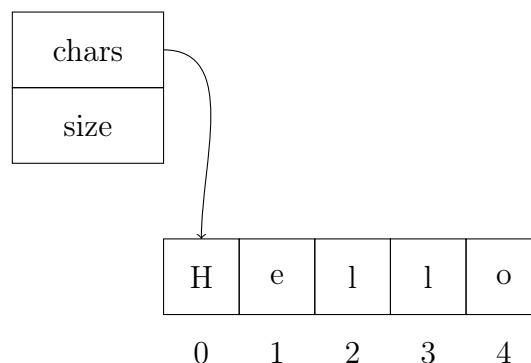
```

struct string {
    /* Les caracteres de la chaîne */
    char *chars;
    /* Nombre de caracteres de la chaîne */
    size_t size;
};

```

(`size_t` est équivalent à `unsigned int`)

Le schéma ci-dessous illustre la représentation de la chaîne « hello ».



Sur cet exemple, `size` vaut 5, et `chars` est un pointeur sur le H du début de la chaîne. On voit que le '\0' final n'est pas nécessaire.

Au cours de cet exercice, nous utiliserons la fonction `memcpy`, donc on rappelle la spécification :

NAME

memcpy - copy memory area

SYNOPSIS

```
#include <string.h>
```

```
void *memcpy(void *dest, const void *src, size_t n);
```

DESCRIPTION

The `memcpy()` function copies `n` bytes from memory area `src` to memory area `dest`. The memory areas should not overlap. Use `memmove(3)` if the memory areas do overlap.

RETURN VALUE

The `memcpy()` function returns a pointer to `dest`.

On va implémenter une fonction permettant d'initialiser une structure de donnée de type `struct string`. En C, cette fonction est implémentée comme suit :

```
void create_string (struct string *str, char *val) {
    size_t size = strlen(val);
    str->size = size;
    str->chars = malloc(size);
    memcpy(str->chars, val, size);
};
```

Question 4 (2 points) *Traduire la fonction `create_string` en assembleur.*

On peut obtenir la longueur d'une `string` avec la fonction suivante :

```
size_t string_length (struct string v) {
    return v.size;
}
```

Pour vous aider, on donne la traduction de cette fonction en assembleur :

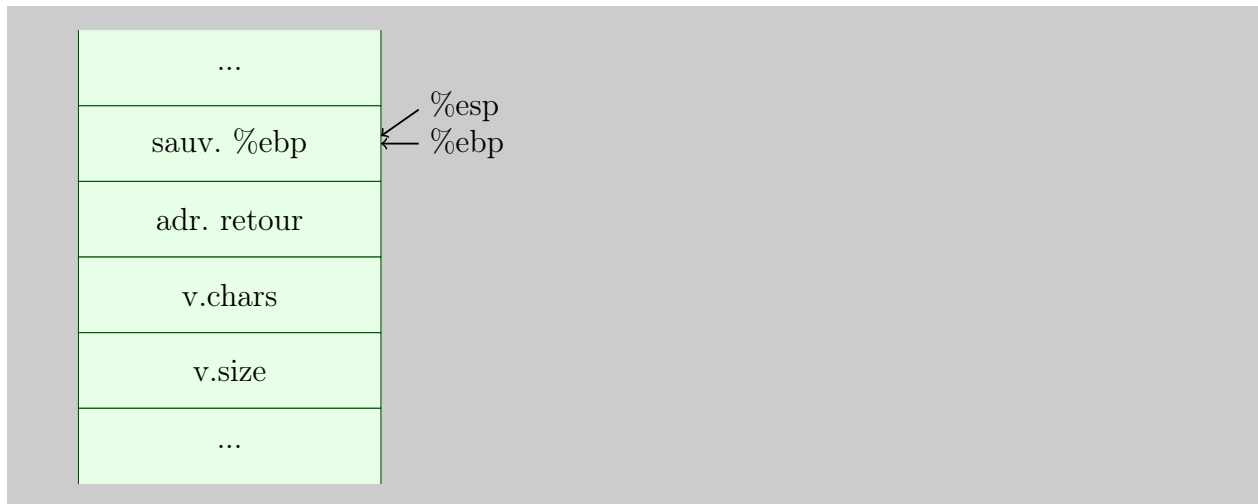
```
chars = 0
size = 4
    .globl string_length_asm
string_length_asm:
    pushl %ebp
    movl %esp, %ebp

v_param=8
    ## return v.size
    movl (v_param + size)(%ebp), %eax
    # on aurait aussi pu écrire
```

```
# movl 12(%ebp), %eax

leave
ret
```

Question 5 (0.5 point) *Dessinez la pile pendant l'appel à cette fonction (la pile est la même bien sûr pour la fonction `string_length` et pour sa traduction `string_length_asm` en assembleur).*



On va maintenant implémenter une fonction `delete_string` avec le profile suivant :

```
void delete_string (struct string v);
```

Cette fonction libère le tableau de caractères de la structure `v` (mais la structure `v` elle-même n'est pas libérée).

Question 6 (0.5 point) *Implémentez cette fonction en C.*

Cf. la fonction `delete_string` dans `string.c`.

Question 7 (1 point) *Traduisez cette fonction en assembleur.*

Cf. la fonction `delete_string_asm` dans `string-asm.S`.

On s'intéresse maintenant à la fonction suivante. Pour vous aider, les commentaires `## ...` correspondent aux endroits où on aurait pu écrire le code C correspondant en suivant les conventions du cours.

```
.globl XXX_string_asm
XXX_string_asm:
    pushl %ebp
```



```

    movl %esp, %ebp
    subl $(4+12), %esp

res_loc=-4
    ## ...
    movl 12(%ebp), %eax
    addl $1, %eax
    pushl %eax
    call malloc
    addl $4, %esp
    movl %eax, res_loc(%ebp)

    ## ...
    pushl 12(%ebp)
    pushl 8(%ebp)
    pushl res_loc(%ebp)
    call memcpy
    addl $12, %esp

    ## ...
    movl res_loc(%ebp), %eax
    addl 12(%ebp), %eax
    movb $0, (%eax)

    ## ...
    movl res_loc(%ebp), %eax

    leave
    ret

```

Question 8 (1.5 points) *Traduire cette fonction en C.*

Cf. fonction `c_string` dans `string.c`.

Question 9 (1 point) *Que fait cette fonction ?*

Elle renvoie une chaîne au format C classique (zero-terminée).

On considère maintenant la fonction :

```

.data
percent_c:      .asciz "%c"
.text
    .globl YY_string_asm

```

```

YYY_string_asm:
    pushl %ebp
    movl %esp, %ebp
    subl $(8+4), %esp

    ## int i;
i_loc = -4
    ## i = 0
    movl $0, i_loc(%ebp)
    ## ...
start:
    movl i_loc(%ebp), %eax
    cmpl 12(%ebp), %eax
    jge end

    ## ...
    movl 8(%ebp), %eax
    addl i_loc(%ebp), %eax
    # zeros pour les bits de poids fort de %ecx
    movl $0, %ecx
    movb (%eax), %cl
    pushl %ecx
    pushl $percent_c
    call printf
    ## ...
    addl $1, i_loc(%ebp)
    ## ...
    jmp start
end:
    leave
    ret

```

Question 10 (1 point) *Que fait cette fonction ?*

Elle affiche les caractères de la chaîne un par un.

Question 11 (2.5 points) *Écrire une version optimisée de cette fonction :*

- La nouvelle fonction ne devra pas avoir de variable locale, mais devra utiliser uniquement les registres.
- La fonction pré-chargera ses arguments dans des registres pour éviter de devoir utiliser des instructions comme `movl 8(%ebp), %eax` à l'intérieur d'une boucle.
- On remplacera l'utilisation de l'instruction `addl i_loc(%ebp), %eax` par le mode d'adressage approprié.

Cf. la fonction `print_string_asm_opt` dans `string-asm.S`.

Question 12 (1.5 points) *Écrire, en C, une fonction*

```
struct string string_concat(struct string v1, struct string v2);
```

Cette fonction crée une nouvelle structure, représentant la concaténation des structures `v1` et `v2`. Il est conseillé d'utiliser la fonction `memcpy()`.

Cf. la fonction `string_concat` dans `string.c`.

Question 13 (3 points) *Traduire cette fonction en assembleur.*

Cf. la fonction `string_concat_asm` dans `string-asm.S`.

Question 14 (1.5 points)

En utilisant les fonctions ci-dessus, écrire, en C, une fonction `main` qui crée 3 structures `string` sur la pile, initialise les deux premières avec respectivement les chaînes "Hello, " et "world!", concatène les deux chaînes dans la troisième.

La fonction `main` affiche ensuite le résultat de la concaténation, puis désalloue ce qui est désallouable avant de terminer le programme.

Cf. la fonction `main` dans `string.c`.

4.1 Fichier `string.c`

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

struct string {
    /* Les caracteres de la chaine */
    char *chars;
    /* Nombre de caracteres de la chaine */
    size_t size;
};

void create_string (struct string *str, char *val) {
    size_t size = strlen(val);
    str->size = size;
```

```

        str->chars = malloc(size);
        memcpy(str->chars, val, size);
};

void delete_string (struct string v) {
    free(v.chars);
}

size_t string_length_asm (struct string v);
size_t string_length (struct string v) {
    return v.size;
}

char *c_string_asm (struct string v);
char *c_string (struct string v) {
    char *res = malloc(v.size + 1);
    memcpy(res, v.chars, v.size);
    res[v.size] = '\0';
    return res;
}

void print_string_ext_asm(struct string v);
void print_string_ext(struct string v) {
    char *cstr = c_string(v);
    printf("%s", cstr);
    free(cstr);
}

void print_string_asm(struct string v);
void print_string_asm_opt(struct string v);
void print_string(struct string v) {
    int i;
    for (i = 0; i < v.size; i++) {
        printf("%c", v.chars[i]);
    }
}

struct string *string_concat_asm(struct string v1, struct string v2);
struct string *string_concat(struct string v1, struct string v2) {
    struct string *res = malloc(sizeof(struct string));
    res->size = v1.size + v2.size;
    res->chars = malloc(res->size);
    memcpy(res->chars, v1.chars, v1.size);
    memcpy(res->chars + v1.size, v2.chars, v2.size);
    return res;
};

```

```

int main(void) {
    struct string v1, v2, *v3;
    create_string(&v1, "Hello\n");
    print_string(v1);
    print_string_ext(v1);
    printf("size=%d\n", string_length(v1));

    create_string_asm(&v2, "world\n");
    printf("print_string          : "); print_string(v2);
    printf("print_string_asm      : "); print_string_asm(v2);
    printf("print_string_asm_opt  : "); print_string_asm_opt(v2);
    printf("size=%d\n", string_length_asm(v2));

    printf("string_concat:\n");
    v3 = string_concat(v1, v2);
    print_string_asm(*v3);
    delete_string(*v3);
    free(v3);

    printf("string_concat_asm:\n");
    v3 = string_concat_asm(v1, v2);
    print_string_asm(*v3);

    delete_string(v1);
    delete_string_asm(v2);
    delete_string_asm(*v3);
    free(v3);
    return 0;
}

```

4.2 Fichier string-asm.S

```

chars = 0
size = 4
.text
    .globl create_string_asm
create_string_asm:
    pushl %ebp
    movl %esp, %ebp
    subl $(4+12), %esp

size_loc=-4
str_param=8
val_param=12
    ## size_t size = strlen(val);
    pushl val_param(%ebp)
    call strlen
    addl $4, %esp

```

```

    movl %eax, size_loc(%ebp)

    ## str->size = size;
    movl size_loc(%ebp), %eax
    movl str_param(%ebp), %ecx
    movl %eax, size(%ecx)

    ## str->chars = malloc(size);
    pushl size_loc(%ebp)
    call malloc
    addl $4, %esp
    movl str_param(%ebp), %ecx
    movl %eax, chars(%ecx)

    ## memcpy(str->chars, val, size);
    movl str_param(%ebp), %eax
    pushl size_loc(%ebp)
    pushl val_param(%ebp)
    pushl chars(%eax)
    call memcpy
    addl $12, %esp

    leave
    ret

    .globl delete_string_asm
delete_string_asm:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp

    ## free(v.chars);
    pushl 8(%ebp)
    call free
    addl $4, %esp

    leave
    ret

.data
percent_s:      .asciz "%s"
.text
    .globl print_string_ext_asm
print_string_ext_asm:
    pushl %ebp
    movl %esp, %ebp
    subl $(4+4), %esp

```

```

    ## char *cstr = c_string(v);
    pushl 12(%ebp)
    pushl 8(%ebp)
    call c_string_asm
    addl $8, %esp
    movl %eax, -4(%ebp)

    ## printf("%s", cstr);
    pushl -4(%ebp)
    pushl $percent_s
    call printf
    addl $8, %esp

    ## free(cstr);
    pushl -4(%ebp)
    call free
    addl $4, %esp

    leave
    ret

.globl string_length_asm
string_length_asm:
    pushl %ebp
    movl %esp, %ebp

v_param=8
    ## return v.size
    movl (v_param + size)(%ebp), %eax
    # on aurait aussi pu écrire
    # movl 12(%ebp), %eax

    leave
    ret

.globl c_string_asm
c_string_asm:
    pushl %ebp
    movl %esp, %ebp
    subl $(4+12), %esp

res_loc=-4
    ## char *res = malloc(v.size + 1);
    movl 12(%ebp), %eax
    addl $1, %eax
    pushl %eax

```

```

    call malloc
    addl $4, %esp
    movl %eax, res_loc(%ebp)

    ## memcpy(res, v.chars, v.size);
    pushl 12(%ebp)
    pushl 8(%ebp)
    pushl res_loc(%ebp)
    call memcpy
    addl $12, %esp

    ## res[v.size] = '\0';
    movl res_loc(%ebp), %eax
    addl 12(%ebp), %eax
    movb $0, (%eax)

    ## return res;
    movl res_loc(%ebp), %eax

    leave
    ret

.data
percent_c:      .asciz "%c"
.text
    .globl print_string_asm
print_string_asm:
    pushl %ebp
    movl %esp, %ebp
    subl $(8+4), %esp

    ## int i;
i_loc = -4
    ## i = 0
    movl $0, i_loc(%ebp)
    ## while (i < v.size) {
start:
    movl i_loc(%ebp), %eax
    cmpl 12(%ebp), %eax
    jge end

    ##      printf("%c", v.chars[i]);
    movl 8(%ebp), %eax
    addl i_loc(%ebp), %eax
    # zeros pour les bits de poids fort de %ecx
    movl $0, %ecx

```



```

        movb (%eax), %cl
        pushl %ecx
        pushl $percent_c
        call printf
        ##      i++;
        addl $1, i_loc(%ebp)
        ## }
        jmp start
end:

        leave
        ret

        .globl print_string_asm_opt
print_string_asm_opt:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        pushl %esi
        pushl %edi
        pushl %ebx

        movl 8(%ebp), %edi // v.chars
        movl 12(%ebp), %ebx // v.size
        ## int i;
        ## i = 0
        xorl %esi, %esi // i
        ## while (i < v.size) {
start_opt:
        cmpl %ebx, %esi
        jge end_opt

        ##      printf("%c", v.chars[i]);
        # zeros pour les bits de poids fort de %ecx
        movl $0, %ecx
        movb (%esi, %edi), %cl
        pushl %ecx
        pushl $percent_c
        call printf
        ##      i++;
        addl $1, %esi
        ## }
        jmp start_opt
end_opt:

        movl -12(%ebp), %esi
        movl -16(%ebp), %edi

```

```

        movl -20(%ebp), %ebx
        leave
        ret

        .globl string_concat_asm
string_concat_asm:
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
res_loc = -4
v1_param = 8
v2_param = 16
        ## struct string *res = malloc(sizeof(struct string));
        pushl $8
        call malloc
        addl $4, %esp
        movl %eax, res_loc(%ebp)

        ## res->size = v1.size + v2.size;
        movl (v1_param+size)(%ebp), %eax
        movl (v2_param+size)(%ebp), %ecx
        addl %ecx, %eax
        movl res_loc(%ebp), %ecx
        movl %eax, size(%ecx)

        ## res->chars = malloc(res->size);
        movl res_loc(%ebp), %eax
        pushl size(%eax)
        call malloc
        addl $4, %esp
        movl res_loc(%ebp), %ecx
        movl %eax, chars(%ecx)

        ## memcpy(res->chars, v1.chars, v1.size);
        pushl (v1_param+size)(%ebp)
        pushl (v1_param+chars)(%ebp)
        movl res_loc(%ebp), %eax
        pushl chars(%eax)
        call memcpy
        addl $12, %esp

        ## memcpy(res->chars + v1.size, v2.chars, v2.size);
        pushl (v2_param+size)(%ebp)
        pushl (v2_param+chars)(%ebp)
        movl res_loc(%ebp), %eax
        movl chars(%eax), %eax
        addl (v1_param+size)(%ebp), %eax

```

```
pushl %eax
call memcpy
addl $12, %esp

## return res;
movl res_loc(%ebp), %eax
leave
ret
```

16

20