Magic : l'Assembleur

Formation C++ bas niveau

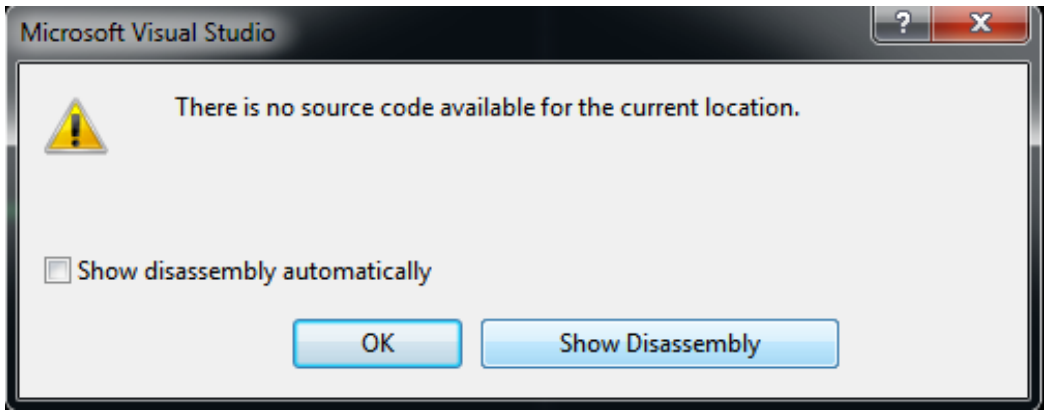
Valentin Roussellet

Centrale Reseaux

February 22, 2013

louen@via.ecp.fr

Abandonnez tout espoir



Pourquoi connaitre l'assembleur

Si vous vivez en 1895 (Nom de Zeus, Marty !)

- Parce qu'il n'y a pas d'autre langage disponible
- Pour écrire du code super optimisé

Mais en 2012 :

- Pour comprendre ce qu'il se passe "sous le capot".
- Pour débugger quand on a pas la source (code optimisé par exemple)

Pourquoi connaitre l'assembleur

Si vous vivez en 1895 (Nom de Zeus, Marty !)

- Parce qu'il n'y a pas d'autre langage disponible
- Pour écrire du code super optimisé

Mais en 2012 :

- Pour comprendre ce qu'il se passe "sous le capot".
- Pour débugger quand on a pas la source (code optimisé par exemple)

Par ou commencer

- On va compiler du code C++ et regarder l'assembleur généré
- J'utiliserai Visual Studio 2010 (syntaxe "Intel")
  - Générer l'assembleur : **cl.exe /FA**
  - Observer le code désassemblé : *Debug* > *Windows* > *Disassembly*
- Mais si vous êtes linuxien :
  - Générer l'assembleur : **gcc -S**
  - Observer le code désassemblé : **disass** dans gdb
  - **-masm=intel** pour GCC ou **set disassembly-flavor intel** dans GDB pour avoir la syntaxe Intel
- A la fin, vous en saurez un peu plus sur les **types**, les **pointeurs**, les **fonctions** et la **pile**, comment sont gérés les **objets** C++, et beaucoup d'autres choses.

Par ou commencer

- On va compiler du code C++ et regarder l'assembleur généré
- J'utiliserai Visual Studio 2010 (syntaxe "Intel")
  - Générer l'assembleur : **cl.exe /FA**
  - Observer le code désassemblé : *Debug* > *Windows* > *Disassembly*
- Mais si vous êtes linuxien :
  - Générer l'assembleur : **gcc -S**
  - Observer le code désassemblé : **disass** dans gdb
  - **-masm=intel** pour GCC ou **set disassembly-flavor intel** dans GDB pour avoir la syntaxe Intel
- A la fin, vous en saurez un peu plus sur les **types**, les **pointeurs**, les **fonctions** et la **pile**, comment sont gérés les **objets** C++, et beaucoup d'autres choses.

Par ou commencer

- On va compiler du code C++ et regarder l'assembleur généré
- J'utiliserai Visual Studio 2010 (syntaxe "Intel")
  - Générer l'assembleur : **cl.exe /FA**
  - Observer le code désassemblé : *Debug* > *Windows* > *Disassembly*
- Mais si vous êtes linuxien :
  - Générer l'assembleur : **gcc -S**
  - Observer le code désassemblé : **disass** dans gdb
  - **-masm=intel** pour GCC ou **set disassembly-flavor intel** dans GDB pour avoir la syntaxe Intel
- A la fin, vous en saurez un peu plus sur les **types**, les **pointeurs**, les **fonctions** et la **pile**, comment sont gérés les **objets** C++, et beaucoup d'autres choses.

Sommaire

What next ?

Introduction

Avengers, Disassemble !

Casse toi, pauvre type !

Sexe a Pile

Condition zéro

La boucle est bouclée

Notre premier désassemblage


```
1  int x = 1;
2  int y = 2;
3  int z = 0;
4
5  z = x + y;
```

```
1      ;1:          int x = 1;
2  00CB139E mov         dword ptr [ebp-8],1
3      ;2:          int y = 2;
4  00CB13A5 mov         dword ptr [ebp-14h],2
5      ;3:          int z = 0;
6  00CB13AC mov         dword ptr [ebp-20h],0
7      ;4:
8      ;5:          z = x + y;
9  00CB13B3 mov         eax,dword ptr [ebp-8]
10 00CB13B6 add         eax,dword ptr [ebp-14h]
11 00CB13B9 mov         dword ptr [ebp-20h],eax
```

Notre premier désassemblage


```
1  int x = 1;
2  int y = 2;
3  int z = 0;
4
5  z = x + y;
```

```
1      ;1:          int x = 1;
2  00CB139E mov         dword ptr [ebp-8],1
3      ;2:          int y = 2;
4  00CB13A5 mov         dword ptr [ebp-14h],2
5      ;3:          int z = 0;
6  00CB13AC mov         dword ptr [ebp-20h],0
7      ;4:
8      ;5:          z = x + y;
9  00CB13B3 mov         eax,dword ptr [ebp-8]
10 00CB13B6 add         eax,dword ptr [ebp-14h]
11 00CB13B9 mov         dword ptr [ebp-20h],eax
```

Qu'est-ce que ca veut dire ?

add Instruction d'addition (you don't say ?)

mov Instruction qui copie de la mémoire :

```
mov dest src
```

eax C'est un **registre** (mémoire embarquée du processeur). Les choses intéressantes se passent dans des registres.

ebp Un autre registre, appelé **base pointer**. On l'utilise pour accéder aux variables locales en mémoire.

dword ptr Valeur du mot mémoire de 32 bits situé a l'adresse entre crochets.

Qu'est-ce que ca veut dire ?


```
1  ; la memoire a l'adresse ebp-8 (&x) prend la valeur 1
2  mov         dword ptr [ebp-8],1
3  ; la memoire a ebp-14h (hex) prend la valeur 2.
4  mov         dword ptr [ebp-14h],2
5  ; idem pour z.
6  mov         dword ptr [ebp-20h],0
7  ; charge le contendu de [ebp-8] (x) dans eax
8  mov         eax,dword ptr [ebp-8]
9  ; additione eax avec le contenu de [ebp-14h] (y)
10 add         eax,dword ptr [ebp-14h]
11 ; deplace la valeur de eax dans [ebp-20h] (z)
12 mov         dword ptr [ebp-20h],eax
```

Pfiouu !

- C'était pas si horrible !
- En assembleur il y a les **instructions** (`mov`, `add`, etc.) et leurs **opérandes**.
- Tout se passe avec les **registres** qui contiennent les résultats temporaires (`eax`, `ebx`,...) ou servent a indexer la mémoire (`esp`,`ebp`...).

What next ?

Petit rappel sur les types

- En C : types *fondamentaux* (mots-clefs du langage)
- 3 catégories : **void**, **entiers** et **nombres en virgule flottante**.
- Chaque type a des *specifications* déterminant leur représentation dans les types *intrinseques* du CPU.
- Les types *dérivés* : **pointeur** sur un type, **structure** ou **union** de plusieurs types.

Petit rappel sur les types

- En C : types *fondamentaux* (mots-clefs du langage)
- 3 catégories : **void**, **entiers** et **nombres en virgule flottante**.
- Chaque type a des *specifications* déterminant leur représentation dans les types *intrinseques* du CPU.
- Les types *dérivés* : **pointeur** sur un type, **structure** ou **union** de plusieurs types.

Les types entiers

- Par ordre croissant de taille
  `char`, `short`, `int`, `long`, `long long`
- Plus les variantes `unsigned`, `signed` (quel est le défaut ?).
- `sizeof(char)== 1` (et au moins 8 bits).
- `char*` permet d'accéder a toute la mémoire.
- Sur x86/Visual Studio

char	1 octet (−128 a 127 ou 0 a 255)
short	2 octets (−32 768 a 32 767 ou 0 a 65535)
int	4 octets (−2 147 483 648 a 2 147 483 647 ou 0 a 4 294 967 296)
long	Identique a int.
long long	8 octets (je vous laisse faire le calcul).

Les types entiers

- Par ordre croissant de taille
  `char`, `short`, `int`, `long`, `long long`
- Plus les variantes `unsigned`, `signed` (quel est le défaut ?).
- `sizeof(char)== 1` (et au moins 8 bits).
- `char*` permet d'accéder a toute la mémoire.
- Sur x86/Visual Studio

char	1 octet (−128 a 127 ou 0 a 255)
short	2 octets (−32 768 a 32 767 ou 0 a 65535)
int	4 octets (−2 147 483 648 a 2 147 483 647 ou 0 a 4 294 967 296)
long	Identique a int.
long long	8 octets (je vous laisse faire le calcul).

Types intrinseques

- Les types réellement manipulés par le CPU : **intrinseques**
- Sous x86 : `dword` (32 bits) (`word` étant 16 bits)
- Un type de 64 bits (`long long`) sera représenté par deux `dword`s.

```
1  ;10:  long long l = 1;
2  00E0139E mov     dword ptr [ebp-0Ch],1
3  00E013A5 mov     dword ptr [ebp-8],0
4  ;11:        long long m = 2;
5  00E013AC mov     dword ptr [ebp-1Ch],2
6  00E013B3 mov     dword ptr [ebp-18h],0
7  ;12:        long long n = l+m;
8  00E013BA mov     eax,dword ptr [ebp-0Ch]
9  00E013BD add     eax,dword ptr [ebp-1Ch]
10 00E013C0 mov     ecx,dword ptr [ebp-8]
11 00E013C3 adc     ecx,dword ptr [ebp-18h]
12 00E013C6 mov     dword ptr [ebp-2Ch],eax
13 00E013C9 mov     dword ptr [ebp-28h],ecx
```

Représentation des entiers

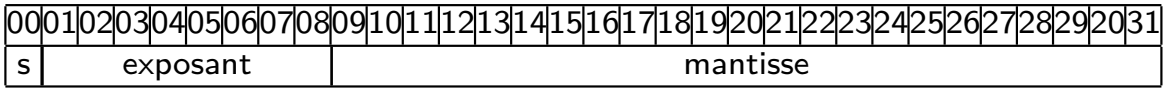
- `unsigned` : valeur binaire
  exemple : `unsigned short x = 0x8A2F` $= 35\ 375$
- `signed` : **complément a deux** $-i = NOT(i) + 1 = 2^n - i$

127	=	0x7F
...		
2	=	0x02
1	=	0x01
0	=	0x00
-1	=	0xFF
-2	=	0xFE
...		
-127	=	0x81
-128	=	0x80

- Le type `bool` : pas de taille précisée, seulement 2 valeurs. (C++ seulement).

Représentation des réels

- Norme IEEE 754.
- `float` (4 octets), `double` (8 octets) `long double` (8 ou 10 octets)



- $val = s \times 2^{\text{exposant}} \times \text{mantisse}$
- mantisse : chaque bit $= \frac{1}{2^b}$
- Il y a donc $+0$ et -0
- Ainsi que $+\infty$ et $-\infty$
- Des valeurs NaN et dénormalisées.
- Attention a la précision (et a `printf`)

Les pointeurs

- Les pointeurs **de données** ont tous la meme taille.
- La taille des pointeurs définit l'*espace adressable*
- Sur win32 : 32 bits (4Gib) – mais ce n'est pas toujours la meme taille que `int` !

Représentation mémoire : l'endianness

- Ordre des octets : **big-endian** et **little-endian**
- Sur PC x86 (et x64) : c'est du little-endian.
- D'autres machines sont big-endian : POWERPC (vieux Mac, XBox 360), Sun SPARC...
- On va regarder la mémoire pour voir comment est représenté un `int`
- Sous VS : *Debug>Windows>Memory* (gdb : *display*)

- code : `unsigned int x = 0x12345678;`
- asm : `mov dword ptr [ebp-8],12345678h`
- mémoire @ebp - 8 : `0x004BF848 78 56 34 12`

Représentation mémoire : l'endianness

- Ordre des octets : **big-endian** et **little-endian**
- Sur PC x86 (et x64) : c'est du little-endian.
- D'autres machines sont big-endian : POWERPC (vieux Mac, XBox 360), Sun SPARC...
- On va regarder la mémoire pour voir comment est représenté un `int`
- Sous VS : *Debug>Windows>Memory* (gdb : *display*)

- code : `unsigned int x = 0x12345678;`
- asm : `mov dword ptr [ebp-8],12345678h`
- mémoire @ebp - 8 : `0x004BF848 78 56 34 12`

Représentation mémoire : l'endianness

- Ordre des octets : **big-endian** et **little-endian**
- Sur PC x86 (et x64) : c'est du little-endian.
- D'autres machines sont big-endian : POWERPC (vieux Mac, XBox 360), Sun SPARC...
- On va regarder la mémoire pour voir comment est représenté un `int`
- Sous VS : *Debug>Windows>Memory* (gdb : *display*)

- code : `unsigned int x = 0x12345678;`
- asm : `mov dword ptr [ebp-8],12345678h`
- mémoire @ebp - 8 : `0x004BF848 78 56 34 12`

Alignement et remplissage

- Les données sont alignées sur la taille d'un mot.

```
1
2 struct Test {
3   int a;
4   char b;
5   long c;
6 };
```

```
1 int main() {
2   struct Test inst;
3   inst.a = 42;
4   inst.b = '@';
5   inst.c = 0x12345678;
6 }
```

- Résultat dans la mémoire :

```
1 0x003EF8FC 2a 00 00 00 // a (32 b)
2 0x003EF900 40 cc cc cc // b (8 b) + 24 b de 'padding'
3 0x003EF904 78 56 34 12 // c (32b)
```

Alignement et remplissage

- Les données sont alignées sur la taille d'un mot.

```
1
2 struct Test {
3   int a;
4   char b;
5   long c;
6 };
```

```
1 int main() {
2   struct Test inst;
3   inst.a = 42;
4   inst.b = '@';
5   inst.c = 0x12345678;
6 }
```

- Résultat dans la mémoire :

```
1 0x003EF8FC 2a 00 00 00 // a (32 b)
2 0x003EF900 40 cc cc cc // b (8 b) + 24 b de 'padding'
3 0x003EF904 78 56 34 12 // c (32b)
```

Alignement et remplissage

- On peut changer le comportement du compilateur

```
1 #pragma pack(1)
2 struct Test {
3   int a;
4   char b;
5   long c;
6 };
```

```
1 int main() {
2   struct Test inst;
3   inst.a = 42;
4   inst.b = '@';
5   inst.c = 0x12345678;
6 }
```

- Résultat dans la mémoire :

```
1 0x003EF8FC 2a 00 00 00 // a (32 b)
2 0x003EF904 40 78 56 34 // b (8b) + 24 premiers b de c
3 0x003EF900 12 cc cc cc // fin de c
```

Alignement et remplissage

- On peut changer le comportement du compilateur

```
1 #pragma pack(1)
2 struct Test {
3   int a;
4   char b;
5   long c;
6 };
```

```
1 int main() {
2   struct Test inst;
3   inst.a = 42;
4   inst.b = '@';
5   inst.c = 0x12345678;
6 }
```

- Résultat dans la mémoire :

```
1 0x003EF8FC 2a 00 00 00 // a (32 b)
2 0x003EF904 40 78 56 34 // b (8b) + 24 premiers b de c
3 0x003EF900 12 cc cc cc // fin de c
```

Alignement et remplissage

- Sous x86 on peut accéder aux données meme non-alignées
- ...sauf dans certains cas (SIMD par exemples).
- Le compilateur aligne par défaut et remplit les structures avec du vide
- ... mais on peut le contraindre : attention a la portabilité !

Ouf !

On a vu pas mal de parametres a prendre en compte pour les types et leur représentation

- Leur taille (par rapport au types intrinseques)
- La convention de signe (ou pas).
- La représentation des flottants (exposant et mantisse)
- L'endianness
- L'alignement

What next ?

A la recherche de La Pile

Ce que vous devriez savoir de La Pile (*stack*).

- C'est une pile (surprise !)
  - Il y a donc les opérations `push()`, `pop()` et `top()`
- Elle contient les variables **automatiques**
- C'est la que sont passés les **arguments** des fonctions (*stack frame*)

Stack frame et appel de fonction

Que se passe t'il quand on appelle une fonction ?

- L'état courant du processeur (i.e. les valeurs des registres) est sauvé sur la Pile.
- Les arguments de la fonction appelée sont positionnés sur la Pile
- L'ensemble de ces données : le **stack frame**.
- Le CPU "saute" a l'adresse de la fonction appelée.

Si on break, on peut voir les stack frames des fonctions empilés :

- Valeur des parametres et variables locales
- Contenu des registres utilisés au moment ou l'appel suivant a été fait
- Adresse de retour de la fonction apres `return`

Les conventions d'appel

La convention d'appel standard de VS/x86 s'appelle `stdcall`

- Ou vont les paramètres ? Sur la pile !
- Ou est la valeur de retour ? dans `eax`
- Qui vide la pile après l'appel ? l'appelé.
- Qui sauve quels registres ? l'appelant (`eax-edx`) et l'appelé le reste

Désassemblons !

Un petit code tout innoffensif :

```
1  int ajouterUn( int argument ){
2    int local = argument + 1;
3    return local;
4  }
5
6  int main() {
7    int result = 0;
8    result = ajouterUn(result);
9    return 0;
10 }
```

- On met un point d'arret ligne 6 (`int main()`)...

Désassemblons !


```
1  push        ebp
2  mov         ebp,esp
3  sub         esp,44h
4  push        ebx
5  push        esi
6  push        edi
7  ;     7:          int result = 0;
8  mov         dword ptr [ebp-4],0
9  ;     8:          result = ajouterUn(result);
10 mov         eax,dword ptr [ebp-4]
11 push        eax
12 call        00C81041
13 add         esp,4
14 mov         dword ptr [ebp-4],eax
15 ;     9:          return 0;
16 xor         eax,eax
17     10: }
18 pop         edi
19 pop         esi
20 pop         ebx
21 mov         esp,ebp
22 pop         ebp
23 ret
```

Don't panic !

esp **Stack Pointer** : pointe en "haut" de la Pile

ebp **Base Pointer** : pointe au début du stack frame courant.

- Les variables sur la Pile sont référencées par rapport a `ebp` : `dword ptr [ebp-4]`
- Mais alors pourquoi des soustractions ?
- Parce que la pile grandit vers le bas ! (le "haut" a une adresse plus petite que le "bas").

Don't panic !

esp **Stack Pointer** : pointe en "haut" de la Pile

ebp **Base Pointer** : pointe au début du stack frame courant.

- Les variables sur la Pile sont référencées par rapport a `ebp` : `dword ptr [ebp-4]`
- Mais alors pourquoi des soustractions ?
- Parce que la pile grandit vers le bas ! (le "haut" a une adresse plus petite que le "bas").

Push it, pop it

- Quand on `push` : `esp` est décrémenté, et l'opérande est stocké dans l'espace pointé par `esp`
- `esp` pointe donc sur la derniere valeur ajoutée dans la stack
- Quand on pop, l'a mémoire par `esp` est copiée dans un registre et `esp` est incrémenté (la pile diminue)

En résumé :

- `push ecx` → `esp --; *esp = ecx;`
- `pop ebx` → `ebx = *esp; esp++;`

Push it, pop it

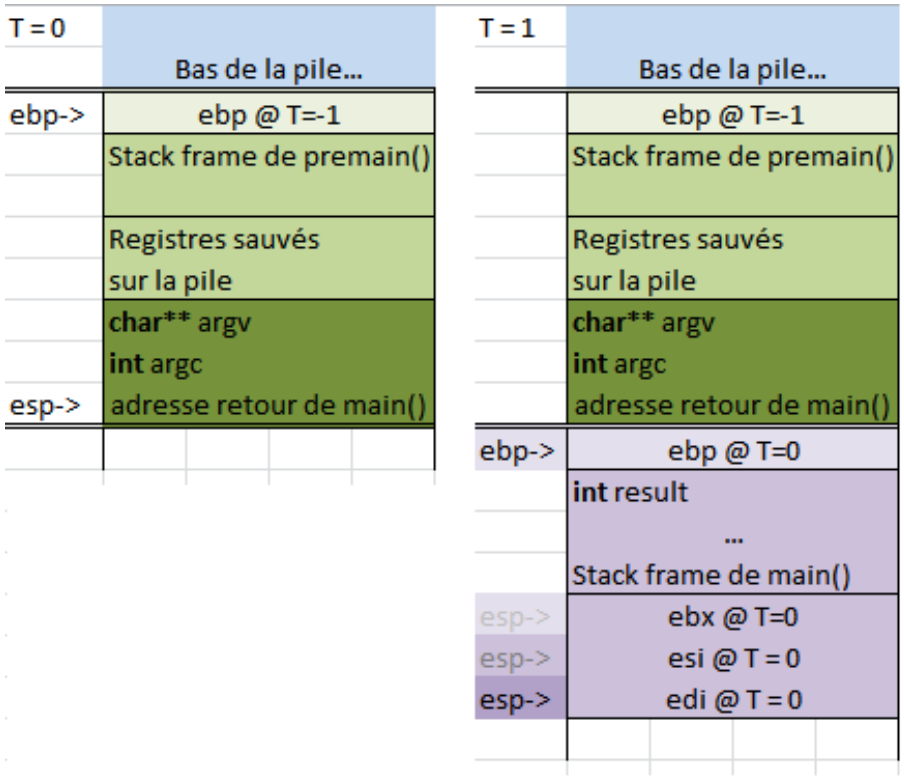
- Quand on `push` : `esp` est décrémenté, et l'opérande est stocké dans l'espace pointé par `esp`
- `esp` pointe donc sur la derniere valeur ajoutée dans la stack
- Quand on pop, l'a mémoire par `esp` est copiée dans un registre et `esp` est incrémenté (la pile diminue)

En résumé :

- `push ecx` → `esp --; *esp = ecx;`
- `pop ebx` → `ebx = *esp; esp++;`

Le préambule

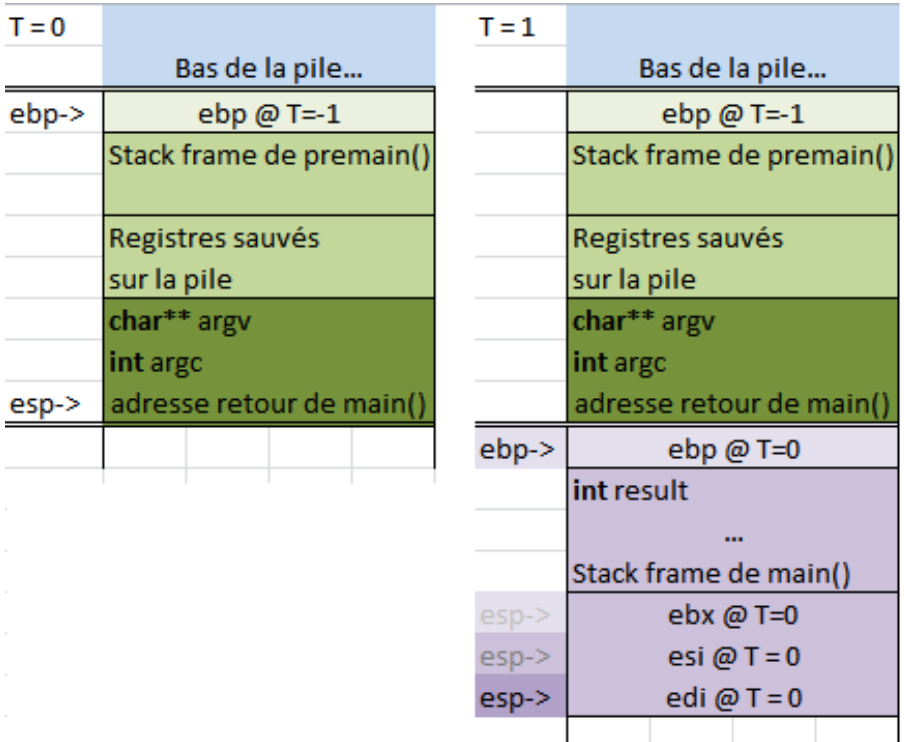

```
1  push        ebp ; l'ancien bp est sauve
2  mov         ebp,esp ; nouveau bp
3  sub         esp,44h ; ajoute de l'espace
4  push        ebx  ; sauve les registres
5  push        esi
6  push        edi
```



L'épilogue

Il fait l'inverse du préambule :

```
1 013112A1 pop      edi
2 013112A2 pop      esi
3 013112A3 pop      ebx
4 013112A4 mov      esp,ebp ; restore esp a son ancienne valeur
5 013112A6 pop      ebp ; l'ancien bp est juste la !
6 013112A7 ret
```



Et notre code ?


```
1  ;     7: int result = 0;
2  mov         dword ptr [ebp-4],0 ; result est a ebp-4 dans notre SF
3  ;     8: result = ajouterUn(result);
4  mov         eax,dword ptr [ebp-4] ; copie le parametre (via eax)
5  push        eax                   ; pour le mettre sur la pile
6  call        00C81041              ; saute a l'adresse de la fonction
7  add         esp,4                 ; pareil que pop(ignore la valeur)
8  mov         dword ptr [ebp-4],eax ; le resultat (eax) va dans result
9  ;     9:          return 0;
10 xor         eax,eax
```

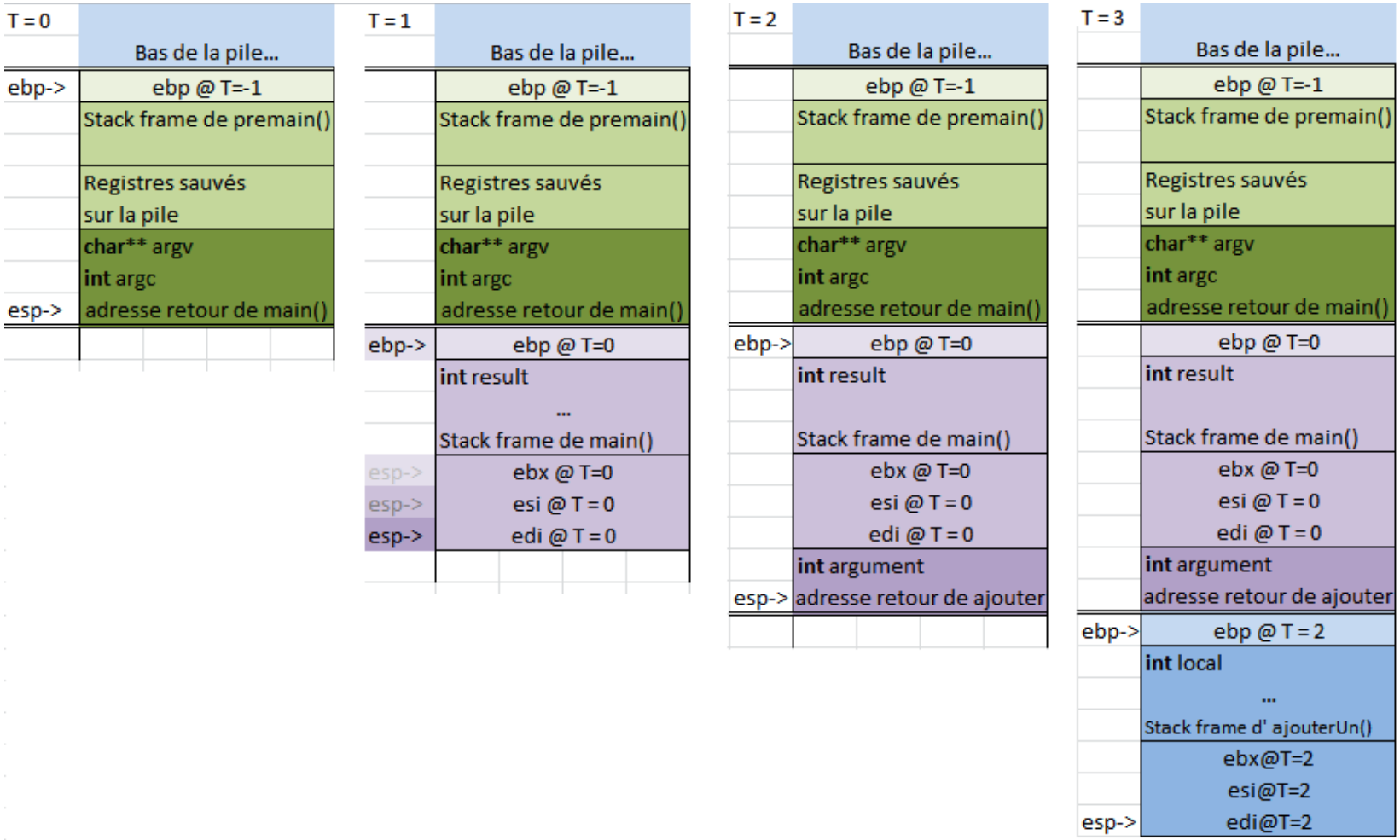
- Le parametre est copié sur la pile (via un registre)
- On sait que le résultat est dans eax.

Notre fonction

Il est temps (enfin) de désassembler notre fonction !

```
1  ; 1: int ajouterUn( int argument ){
2  push        ebp     ;preambule : on connait !
3  mov         ebp,esp
4  sub         esp,44h
5  push        ebx
6  push        esi
7  push        edi
8  ;2:  int local = argument + 1;
9  mov         eax,dword ptr [ebp+8] ;c'est hors de notre SF
10 add         eax,1
11 mov         dword ptr [ebp-4],eax ; variable locale dans notre SF
12 ;3:  return local;
13 mov         eax,dword ptr [ebp-4] ; resultat dans eax
14 ;4: }
15 pop         edi  ; epilogue : on connait aussi
16 pop         esi
17 pop         ebx
18 mov         esp,ebp
19 pop         ebp
20 ret
```

Notre fonction



Résumé

La pile sert a :

- Stocker les variables locales
- Sauver les registres et passer les parametres (un ou plusieurs !)

La **convention d'appel** détermine qui fait quoi sur la pile.
Le **stack frame**

- Un par appel de fonction
- Compris entre `esp` et `ebp`
- Les arguments : dans le SF de la fonction appelante

What next ?

If I had a hammer

- Démarrons avec un code tres facile :

```
1 int f (int arg) {
2   int local = 0;
3   if ( arg < 0 ) {
4     local = 1;
5   }
6   return 0;
7 }
```

If I had a hammer

- Voila ce que donne le désassemblage :

```
1  ;2:         int local = 0;
2  00B91299 mov         dword ptr [ebp-4],0
3  ;3:         if ( arg < 0 ) {
4  00B912A0 cmp         dword ptr [ebp+8],0
5  00B912A4 jge         00B912AD
6  ;4:                 local = 1;
7  00B912A6 mov         dword ptr [ebp-4],1
8  ;5:         }
9  ;6:         return 0;
10 00B912AD xor         eax,eax
```

Que remarquons-nous ?

- La ligne du `if` devient deux instructions
  - `cmp` comparaison (aucun effet mais change les EFLAGS)
  - `jge` Jump if Greater or Equal : saute a l'adresse indiquée selon la valeur des EFLAGS
- C'est l'inverse de ce qu'on a écrit en C !
- **si** *condition* **alors** entrer dans les accolades
- **si** ¬*condition* **alors** sauter

```
1 int local = 0;
2 if ( arg >= 0 ) goto Positive;
3 local = 1;
4 Positive:
5 return 0;
```

- Bonus : c'est le registre **eip** (instruction pointer) qui est modifié par `jge` !)

Que remarquons-nous ?

- La ligne du `if` devient deux instructions
  - `cmp` comparaison (aucun effet mais change les EFLAGS)
  - `jge` Jump if Greater or Equal : saute a l'adresse indiquée selon la valeur des EFLAGS
- C'est l'inverse de ce qu'on a écrit en C !
- **si** *condition* **alors** entrer dans les accolades
- **si** $\neg$*condition* **alors** sauter

```
1 int local = 0;
2 if ( arg >= 0 ) goto Positive;
3 local = 1;
4 Positive:
5 return 0;
```

- Bonus : c'est le registre **eip** (instruction pointer) qui est modifié par `jge` !)

Que remarquons-nous ?

- La ligne du `if` devient deux instructions
  - `cmp` comparaison (aucun effet mais change les EFLAGS)
  - `jge` Jump if Greater or Equal : saute a l'adresse indiquée selon la valeur des EFLAGS
- C'est l'inverse de ce qu'on a écrit en C !
- **si** *condition* **alors** entrer dans les accolades
- **si** $\neg$*condition* **alors** sauter

```
1 int local = 0;
2 if ( arg >= 0 ) goto Positive;
3 local = 1;
4 Positive:
5 return 0;
```

- Bonus : c'est le registre **eip** (instruction pointer) qui est modifié par `jge` !)

Que remarquons-nous ?

- La ligne du `if` devient deux instructions
  - `cmp` comparaison (aucun effet mais change les EFLAGS)
  - `jge` Jump if Greater or Equal : saute a l'adresse indiquée selon la valeur des EFLAGS
- C'est l'inverse de ce qu'on a écrit en C !
- **si** *condition* **alors** entrer dans les accolades
- **si** $\neg$*condition* **alors** sauter

```
1 int local = 0;
2 if ( arg >= 0 ) goto Positive;
3 local = 1;
4 Positive:
5 return 0;
```

- Bonus : c'est le registre **eip** (instruction pointer) qui est modifié par `jge` !)

Un peu plus complexe


```
1  int local = 0;
2  if ( arg == 0 ){
3    local = 12;
4  }
5  else if (arg != 42) {
6    local = 6 * 9;
7  }
8  else {
9    local = 666;
10 }
11 return 0;
```

Un peu plus complexe


```
1  ;     2:  if ( arg == 0 ){
2  00F51260 cmp         dword ptr [ebp+8],0
3  00F51264 jne         0F5126Fh ; jump if not equal
4  ;     3:    local = 12;
5  00F51266 mov         dword ptr [ebp-4],0Ch
6  00F5126D jmp         0F51285h; on sort du bloc if
7  ;     4:  }
8  ;     5:  else if (arg != 42) {
9  00F5126F cmp         dword ptr [ebp+8],2Ah
10 00F51273 je          0F5127Eh ; jump if equal
11 ;     6:    local = 6 * 9;
12 00F51275 mov         dword ptr [ebp-4],36h
13 00F5127C jmp         0F51285h  ; on sort du bloc if
14 ;     7:  }
15 ;     8:  else {
16 ;     9:    local = 666;
17 00F5127E mov         dword ptr [ebp-4],29Ah ;
18 ;    10:  }
19 ;    11:  return 0;
20 00F51285 xor         eax,eax
```

Dernier exemple : opérations logiques


```
1  int f (int arg) {
2    int local = 0;
3
4    if ( arg >= 7 && arg <= 12){
5      local = 9001;
6    }
7    else if (arg || ! arg || arg == 69 ) { // stupide :)
8      local = 6 * 9;
9    }
10   return 0;
11 }
```

Dernier exemple : operations logiques


```
1  ;    4:  if ( arg >= 7 && arg <= 12){
2  00E61260 cmp         dword ptr [ebp+8],7 ; condition 1
3  00E61264 jl          00E61275            ; saute au else si arg < 7
4  00E61266 cmp         dword ptr [ebp+8],0Ch ; condition 2
5  00E6126A jg          00E61275              ; saute au else si arg > 12
6  ;    5:    local = 9001;
7  00E6126C mov         dword ptr [ebp-4],2329h
8  00E61273 jmp         00E6128E
9  ;    6:  }
10 ;    7:  else if (arg || ! arg || arg == 69 ) {
11 00E61275 cmp         dword ptr [ebp+8],0 condition 1
12 00E61279 jne         00E61287            ; surprise ! pas d'inversion
13 00E6127B cmp         dword ptr [ebp+8],0
14 00E6127F je          00E61287            ; toujours pas d'inversion
15 00E61281 cmp         dword ptr [ebp+8],45h
16 00E61285 jne         00E6128E            ; cette fois si.
17 ;    8:    local = 6 * 9;
18 00E61287 mov         dword ptr [ebp-4],36h
19 ;    9:  }
20 ;   10:  return 0;
21 00E6128E xor         eax,eax
22 ;   11: }
```

Résumons

- Chaque condition est transformée en `cmp` et `jXX`
- Les conditions des sauts conditionnels sont **inversées** par rapport au `if`.
- L'évaluation "paresseuse" des conditions se fait naturellement : chaque "atome" génere un couple comparaison + saut.
- Testez avec un operateur ternaire (`a?b:c;`) ou un `switch`.

What next ?

Petite histoire des boucles

Protozoaire supérieur : `if` - `goto`

```
1  int main() {
2    int tab [5] ={ 8, 4, 22, 904, 30 };
3    int somme = 0; int compteur = 0;
4
5  DebutBoucle:
6    if (compteur < 5 ){
7      somme += tab[compteur];
8      ++compteur;
9      goto DebutBoucle;
10   }
11   return 0;
12 }
```

Étudions ce fossile


```
1  ;     5: DebutBoucle:
2  ;     6:  if (compteur < 5 ) {
3  013F1284 cmp         dword ptr [ebp-20h],5 ; on reconnait un if
4  013F1288 jge         13F12A2h              ; cmp + jump !
5  ;     7:  somme += tab[compteur];
6  013F128A mov         eax,dword ptr [ebp-20h]     ; compteur dans eax
7  013F128D mov         ecx,dword ptr [ebp-1Ch]     ; somme dans ecx
8  013F1290 add         ecx,dword ptr [ebp+eax*4-18h] ; WTF ??
9  013F1294 mov         dword ptr [ebp-1Ch],ecx     ; resultat dans somme
10 ;     8:  ++compteur;
11 013F1297 mov         eax,dword ptr [ebp-20h]
12 013F129A add         eax,1
13 013F129D mov         dword ptr [ebp-20h],eax
14 ;     9:  goto DebutBoucle;
15 013F12A0 jmp         13F1284h
16 ;    10:  }
17 ;    11:  return 0;
18 013F12A2 xor         eax,eax
```

Tombés sur un os

- `add ecx,dword ptr [ebp+eax*4-18h]`
- On veut accéder a l'élément du tableau désigné par `compteur` (dans `eax`).
- Réarangeons la formule : `(epb - 18h) + (eax * 4)`
- `&tab[0] + (compteur * sizeof(int))`

Petite histoire des boucles

Bas Moyen-Age : `while`

```
1  int main() {
2    int tab [5] ={ 8, 4, 22, 904, 30 };
3    int somme = 0; int compteur = 0;
4
5    while (compteur < 5 ){
6      somme += tab[compteur];
7      ++compteur;
8    }
9    return 0;
10 }
```

Faith in humanity restored


```
1  ;     5:  while (compteur < 5 ){
2  01161284 cmp         dword ptr [ebp-20h],5
3  01161288 jge         11612A2h
4  ;     6:  somme += tab[compteur];
5  0116128A mov         eax,dword ptr [ebp-20h]
6  0116128D mov         ecx,dword ptr [ebp-1Ch]
7  01161290 add         ecx,dword ptr [ebp+eax*4-18h]
8  01161294 mov         dword ptr [ebp-1Ch],ecx
9  ;     7:  ++compteur;
10 01161297 mov         eax,dword ptr [ebp-20h]
11 0116129A add         eax,1
12 0116129D mov         dword ptr [ebp-20h],eax
13 ;     8:  }
14 011612A0 jmp         1161284h
15 ;     9:  return 0;
16 011612A2 xor         eax,eax
```

Vous pouvez vérifier le `do{...} while(...)`.

Petite histoire des boucles

Époque moderne : `for`

```
1 int main() {
2   int tab [5] ={ 8, 4, 22, 904, 30 };
3   int somme = 0;
4   for (int compteur = 0; compteur < 5; ++compteur){
5     somme += tab[compteur];
6   }
7   return 0;
8 }
```

Hail to the king, baby!


```
1  ;     5:  for (int compteur = 0; compteur < 5; ++compteur){
2  00AA127D mov         dword ptr [ebp-20h],0  ; compteur = 0
3  00AA1284 jmp         00AA128F
4
5  00AA1286 mov         eax,dword ptr [ebp-20h]
6  00AA1289 add         eax,1                  ; ++compteur
7  00AA128C mov         dword ptr [ebp-20h],eax
8
9  00AA128F cmp         dword ptr [ebp-20h],5 ;on retrouve notre if
10 00AA1293 jge         00AA12A4              ;(test de sortie de boucle)
11
12 ;     6:  somme += tab[compteur];
13 00AA1295 mov         eax,dword ptr [ebp-20h]
14 00AA1298 mov         ecx,dword ptr [ebp-1Ch]
15 00AA129B add         ecx,dword ptr [ebp+eax*4-18h]
16 00AA129F mov         dword ptr [ebp-1Ch],ecx
17 ;     7:  }
18 00AA12A2 jmp         00AA1286
19 ;     8:  return 0;
20 00AA12A4 xor         eax,eax
21 ;     9: }
```

Final : hardcore loop porn

Etes vous prets a affronter de l'assembleur optimisé ?

```
1  #include <cstdlib>
2  int main(int argc, char**argv) {
3    int tab [5] ={ 8, 4, 22, 904, 30 };
4    int somme = 0;
5    int taille = atoi(argv[0]);
6    for (int compteur = 0; compteur<taille; ++compteur){
7      somme += tab[compteur];
8    }
9    return 0;
10 }
```

- **/O2** : Active les optimisations
- **/Ob0** : Désactive l'inlining

Hardcore loop porn


```
1  00F21042 xor      ecx,ecx  ; ecx = 0
2  00F21044 cmp      eax,2    ; si eax est < 2
3  00F21047 jl       0F2105Fh ; sauter a la fin de la boucle
4  00F21049 lea      edx,[eax-1] ; edx = eax -1
5  00F2104C lea      esp,[esp]
6  ; -- debut boucle
7  00F21050 add      esi,dword ptr [ebp+ecx*4-14h]; esi += tab [i]
8  00F21054 add      edi,dword ptr [ebp+ecx*4-10h]; esi += tab [i + 1]
9  00F21058 add      ecx,2;  ecx = ecx + 2
10 00F2105B cmp      ecx,edx  ; si ecx est < eax -1
11 00F2105D jl       0F21050h ; boucler
12 ; -- fin boucle
13 00F2105F cmp      ecx,eax  ; si ecx est < eax
14 00F21061 jge      0F21067h ; sauter l'instruction suivante (pair)
15 00F21063 mov      ebx,dword ptr ebx,dword ptr [ebp+ecx*4-14h]
16 00F21067 lea      eax,[edi+esi] ; eax = somme finale
17 00F2106A pop      edi
18 00F2106B pop      esi
19 00F2106C add      eax,ebx ; eax += impair eventuel
```

Looping

- Au niveau de l'assembleur les boucles ne sont pas si différentes du `if` - `goto`.
- Mais elles automatisent les choses (et évitent de se tromper)
- Nous avons vu un exemple d'optimisation : le déroulage de boucle

Fini !

- Alex Darby, *A Low Level Curriculum For C and C++* (AltDevBlogADay)
- *Intel IA 32 Architecture Software Developper Manual* (intel.com)
- MazeGen, *coder32* ASM reference (refx86asm.net)

```
ret
```