Formation C++ Ubisoft - Module 4

Romain Arcila[1,2]
Charles de Rousiers[1]

10 mai 2009

[1] INRIA Grenoble
[2] Liris - CNRS Lyon

Plan

Objectifs

Les principaux points abordés :

- Les mécanismes de compilation
- L'impact du code source sur le code ASM
- Les points importants pour produire du code efficace

Objectif

Comprendre la génération du code assembleur depuis le code C++

- Plateforme : PC sous Windows
- Compilateur : Visual Studio C++ 8

Et savoir comment modifier le code source pour améliorer cette génération

Plan

Exercices

Exercices pratiques

1. Trouver les mauvaises pratiques de codage utilisées pour rendre un code plus optimisé
2. Écrire un programme cache friendly

Plan

Compilation

Besoins pour le programmeur

- Avoir une abstraction forte
- Expression de concept aisée
- Intelligibilité facile

Code haut niveau

Écriture de programme en langage de haut niveau (Ex : C++)

Compilation

CPU = machine à états : exécution pas à pas des instructions d'un programme

Fonctionnement (Sur plusieurs cycles)

1. Lecture de l'instruction
2. Décodage
3. Récupération des valeurs (opérandes)
4. Exécution de l'instruction

Besoins pour le CPU

- Interprétation rapide et non ambiguë
- Pas besoin d'abstraction : simple machine à états
- Utilisation d'un langage bas niveau (ASM)

Conslusion

- Langage haut niveau / langage ASM $\Rightarrow$ Objectifs différents
- Besoin d'une étape de traduction intermédiaire : **Compilation**

Compilation

Contexte

Compilateur : réalise la traduction code haut niveau $\longrightarrow$ code bas niveau

Cette étape nécessite :

- De respecter les concepts exprimés en langage haut niveau
- De rendre les concepts interprétable par la processeur
- De produire le code le plus **efficace** possible

Code **efficace** ?

- En taille
- En vitesse
- En exploitation des ressources

$\Rightarrow$ Traduction directe non optimale : **besoin d'optimisation** !

Compilation

Comment réaliser ces optimisations ?

- Analyse code haut niveau : repérage des contraintes
- Simplification selon des schémas connus

Étape difficile pour le compilateur : besoins d'aide de la part du développeur !

Règle

- Plus une portion de code a de degrés de liberté, plus le code est difficile à optimiser
- **Raison** : compilateur ne sait pas dans quelles mesures les données sont lues, modifiées, ... etc

Compilation

Que contient un exécutable ?

- Instructions (en langage machine)
- Et des variables (Adresses, constantes, variables static, ...)

Au lancement du programme, allocation d'un espace mémoire par l'OS, servant à stocker :

- Segment de code
- Segment d'instruction
- Pile pour les allocation statiques/contextuelles
- Tas pour les allocations dynamiques

Assembleur

Définition

Assembleur : langage bas niveau (quasi-)interprétable par le processeur
$\longrightarrow$ Traduction ensuite en langage machine (code binaire)

Avantages

- Granularité d'action très fine
- Possibilité d'optimisation importante

Pourquoi ne pas tout coder en ASM ?

- Difficulté de conception/d'intelligibilité
- Optimisation vs. Maintenabilité

Assembleur - Syntaxe

Syntaxe relativement simple, mais pas toujours aisé à lire

- Succession d'instructions
- Instruction possède de 0 à plusieurs opérandes
- Opérande : adresses, valeurs de données ...

Remarque

- Existance de plusieurs syntaxe pour la rédaction en assembleur.
- Assembleur gérer par Visual Studio : **MASM**

Code

```
mov     eax, $1
mov     ecx, $2
add     eax, ecx
```

Assembleur - Données

Initialement les données sont présentes en mémoire et proviennent :

- Entrées utilisateur
- Fichiers
- Constantes ou Variables static dans le segments de données
- Valeurs brutes dans le segment de données
- ...

Stockage en mémoire

Manipulation des données depuis trois zones

- Soit en mémoire (au sens large RAM, cache, ...)
- Soit sur la pile
- Soit dans les registres

Assembleur - Registres

Principaux registres

- Registre de données : *eax*, *ebx*, *ecx*, *edx*
- Registre de pointeur d'instruction *eip*
- Registre de pile : *esp* (sommet) et *ebp* (base)

Sur processeur 32 bits, décomposition des registres de la manière suivante :

- *eax* : 32 bits
- *ax* : 16 bits (bas)
- *al* : 8 bits (haut)
- *ah* : 8 bits (bas)

Assembleur - Instructions

Instructions pour les opérations arithmétiques

- mov dst, src
- add dst, src
- sub dst, src
- mul dst, src

Instructions de manipulation de la pile

- push reg
- pop reg
- call functionAddress
- ret val

Instructions de manipulation de mémoire

- lea reg, adr
- mov dst, reg[reg + val]

Caches

Notion de cache

Interface entre la RAM et le processeur

- Permet au processeur d'avoir la donnée souhaitée rapidement
- Permet armortir la latence d'accès

Processeurs manipule des données de deux types : donc 2 caches

- Cache d'instruction
- Cache de données : adresses, valeurs, ...

Attention

Importance du cache dans les processeurs actuels en raison de leur architecture 'stream'

1. Si défaut de cache : trou dans le flux de traitement
2. Processeur n'a plus d'instruction à traiter
3. Latence dans le traitement : résultat non immédiat

$\longrightarrow$ Perte de performances

Optimisation - Généralités

But

Optimisation : que souhaite-on faire réellement ?
$\Rightarrow$ Rendre le code plus efficace (rapide, concis, ...)

D'une manière générale

- Plusieurs niveaux d'optimisation (algorithme, architecture, code ...)
- Identification des besoins : Profiler (nombre appel, temps passé dans une fonction, cache, ...)
- Optimisation : pour un code, pour un processeur, pour un compilateur

Écrire du code assembleur à la place du compilateur car pas assez performant ?

Non ! Compilateur extrêment doué dans cette tâche.
$\rightarrow$ Écrire tout à la main : risque de faire moins bien que le compilateur

Optimisation - ASM

Possibilité au sein du code C++ d'écrire des portions ASM

Code

```
void foo()
{
        asm
        {
                mov ax, 0x13
                int 0x10
        }
}
```

Attention

- Strictement réservé à l'optimisation de procédures identifiées 'hot' (CPU intensive/bottleneck) lors de phase profiling.
- Ne sert stricement à rien d'optimiser de cette manière dès le début du projet !
- Code ASM généré peut semblé plus lent localement mais être plus rapide globalement
- Code optimisé pour les caches miss, effect de bords ... etc

Optimisation - Objectifs

Objectif

Aider le compilateur à générer le code le plus efficace possible.

Comment faire ?

Donner au compilateur

- Réduire la liberté d'interprétation du code
- Donner des indications sur nos intentions

Plan

Optimisation globale

Question : Comment marche l'optimisation lors de la phase de création du programme ?

Habituellement

1. Analyse de la structure des .cpp
2. Optimisation et Compilation des objets .obj
3. Édition des liens avec tous les objets

Optimisation : Whole Program Optimization

1. Analyse de la structure des .cpp
2. Optimisation et Compilation des objets .obj
3. Analyze des dépendances entres les objets
4. Re-Optimisation et édition des liens avec tous les objets

Optimisation globale

Avantages

Lors de la une seconde passe d'optimisation à l'édition des liens :

- Utilisation plus performante des registres
- Modification des conventions d'appels
- Inline supplémentaire
- Suppression du 'code mort'
- ...

Optimisation par analyse

Profile Guided Optimization

Analyse statistique du code pour effectuer les optimisations les plus pertinentes

1. Compilation optimisée de l'application
2. Édition des liens normale (option /LTCG :PGI)
3. Test de l'application sur des scénario d'exécution (profiling)
4. Modification du code à l'édition des liens à partir les informations de profiling

Types d'optimisation effectués

- Réordonnancement de blocs conditionnelle (les plus fréquent en premier)
- Switch expansion
- Inlining intelligent / partielle
- Meilleur gestion des appels virtuel
- Dissociation code hot/cold pour favoriser le cache

Optimisation par analyse

Switch Expansion : Idenitification des données les plus probables et ajout de lien directs

Code avant optimisation

```
switch(i) {
        case 1 : ...
        case 2 : ...
        case 3 : ...
        default : ...
}
```

Phase de profiling : Analyse montre que dans 90% des cas i == 10

Code après optimisation

```
if(i ==10)
        goto default:
switch(i) {
        case 1 : ...
        case 2 : ...
        case 3 : ...
        default : ...
}
```

Optimisation par analyse

Autres optimisations

Utilisation de OpenMP : Directives données sous forme de pragmas

Exemple

```
void test(int first, int last)
{
        #pragma omp parallel for
        for(int i = first; i<=last; ++i) {
                a[i] = b[i] + c[i];
        }
}
```

À la compilation : Répartition des itérations de la boucle sur chaque core

Exemple

Pour $first = 1$ et $last = 1000$

- Core 1 : $1 <= i <= 250$
- Core 2 : $251 <= i <= 500$
- Core 3 : $501 <= i <= 750$
- Core 4 : $751 <= i <= 1000$

Configuration du compilateur

Politique d'optimisation

Possibilité d'influencer la manière dont est fait l'optimisation

- Au niveau **global** du projet
- Au niveau **local** pour une portion de code donnée

Configuration du compilateur

Au niveau **global** :Compilateur possède différentes règles d'optimisation globale

- /*O*1 : optimisation de la vitesse en réduisant la taille du code pour favoriser le cache (pour les grandes applications)
- /*O*2 : optimisation de la vitesse par extension (inline, déroulement de boucle, ...) du code (pour les petites applications)
- /*Ox* : optimisation de la vitesse et tenir compte de la taille (moins performant que /O1 ou /O2)
- /*Os* : optimisation de la taille

Conseil

Il est recommandé d'utiliser les options /Os, /O1 ou /O2 plutôt que /Ox

Configuration du compilateur

Au niveau **local** : utilisation de pragma pour spéficier une politique d'optimisation particulière

- Au niveau programme : optimisation de type $/O1$
- Au niveau local : après une phase de profiler, on s'aperçoit que la méthode Foo est utilisée intensivement : besoin de passer en $/O2$

Code

```
#pragma optimize("t", on)
int Foo(const A& _param)
{
  ...;
}
#pragma optimize("", on)
```

Optimisation et debuggage

Problème

Nécessaire parfois de faire de debug sur une version release : code optimisé !
Debug difficile car :

- Fusion de fonctions
- Suppression de variables locales
- Réarrangement de code
- Simplification d'instructions etc

$\rightarrow$ Nécessaire de regarder le code assembleur pour voir ce que le programme fait concretement !

Exemple

for(x=0 ; x<10 ; x++)

- Suppression de la boucle
- Une initialisation
- Déroulage de la boucle, ...

Plan

Représentation

En C++ existence de plusieurs types dit primitifs

Représentation (processeur 32 bits)

- *Char* : 8 bits
- *Short* : 16 bits
- *Int* : 32 bits (type natif)
- *Float* : 32 bits
- *Double* : 64 bit
- *Long* : 64 bit

Le prefix unsigned modifie uniquement la représentation et non la taille

Nombres flotants

La manipulation particulière des nombres flotants

- Instructions séparées
- Utilisation d'un coprocesseur (FPU)
- Utilisation d'une pile secondaire pour effectuer certain calcul

Quelques idées reçues...

Les opérations arithmétiques sont plus rapides sur les nombres non signés : **Faux !**

Code

```
; TestArithmeticSigned
       _c$ = -16                                ; size = 4
       _b$ = -12                                ; size = 4
       _a$ = -8                                 ; size = 4
       _res$ = -4                               ; size = 4
       mov     DWORD PTR _a$[ebp], 2
       mov     DWORD PTR _b$[ebp], 4
       mov     DWORD PTR _c$[ebp], 7
       mov     eax, DWORD PTR _a$[ebp]
       add     eax, DWORD PTR _b$[ebp]
       imul    eax, DWORD PTR _c$[ebp]
       mov     DWORD PTR _res$[ebp], eax

; TestArithmeticUnsigned
       _c$ = -16                                ; size = 4
       _b$ = -12                                ; size = 4
       _a$ = -8                                 ; size = 4
       _res$ = -4                               ; size = 4
       mov     DWORD PTR _a$[ebp], 2
       mov     DWORD PTR _b$[ebp], 4
       mov     DWORD PTR _c$[ebp], 7
       mov     eax, DWORD PTR _a$[ebp]
       add     eax, DWORD PTR _b$[ebp]
       imul    eax, DWORD PTR _c$[ebp]
       mov     DWORD PTR _res$[ebp], eax
```

Quelques idées reçues...

Les conversions entier $\leftrightarrow$ flotant sont plus rapides avec les entiers signés : **Vrai !**

Code

```
; TestConversionIntToFloat
        _b$ = -12                                       ; size = 4
        _a$ = -8                                        ; size = 4
        _res$ = -4                                      ; size = 4

        mov     DWORD PTR _a$[ebp], 2
        fld     DWORD PTR __real@400ccccd       ; 2.2
        fstp    DWORD PTR _b$[ebp]
        fild    DWORD PTR _a$[ebp]
        fadd    DWORD PTR _b$[ebp]
        fstp    DWORD PTR _res$[ebp]
; TestConversionUIntToFloat
        tv71 = -20                                      ; size = 8
        _b$ = -12                                       ; size = 4
        _a$ = -8                                        ; size = 4
        _res$ = -4                                      ; size = 4

        mov     DWORD PTR _a$[ebp], 2
        fld     DWORD PTR __real@400ccccd
        fstp    DWORD PTR _b$[ebp]
        mov     eax, DWORD PTR _a$[ebp]
        mov     DWORD PTR tv71[ebp], eax
        mov     DWORD PTR tv71[ebp+4], 0
        fild    QWORD PTR tv71[ebp]
        fadd    DWORD PTR _b$[ebp]
        fstp    DWORD PTR _res$[ebp]
```

Quelques idées reçues...

Les conversions double vers float sont couteuses : faux !

Remarque

Idée reçu : Conversion float vers double couteuse (donc mettre 'f' sur vos constantes numériques pour éviter)
Ainsi le 'f' sur vos constantes numériques n'est pas obligation bien que conseillé :

Code

```
void UseFloat(float _value) {
        printf("%f\n",_value);
}

void TestFloatConversion() {
        UseFloat(1.99512452f);
        UseFloat(1.8313212132189);

         float  fvalue = 2.9124869752f;
        double dvalue = 2.92195124558695;
        UseFloat(fvalue);
        UseFloat(dvalue);
}
```

Quelques idées reçues...

Code

```
__value$ = 8                                    ; size = 4
?UseFloat@@YAXM@Z PROC                          ; UseFloat
        fld     DWORD PTR __value$[esp-4]
        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET $SG3743
        call    DWORD PTR __imp__printf
        add     esp, 12                         ; 0000000cH
        ret     0
?UseFloat@@YAXM@Z ENDP                          ; UseFloat
        fld     DWORD PTR __real@3fff603e
        push    ecx
        fstp    DWORD PTR [esp]
        call    ?UseFloat@@YAXM@Z               ; UseFloat
        fld     DWORD PTR __real@3fea68bc
        fstp    DWORD PTR [esp]
        call    ?UseFloat@@YAXM@Z               ; UseFloat
        fld     DWORD PTR __real@403a6630
        fstp    DWORD PTR [esp]
        call    ?UseFloat@@YAXM@Z               ; UseFloat
        fld     DWORD PTR __real@403b0140
        fstp    DWORD PTR [esp]
        call    ?UseFloat@@YAXM@Z               ; UseFloat
```

Compilateur indique seulement qu'il pourrait y avoir une perte de précision

Alignement

Alignement des données sur la taille native des processeurs (32/64 bits)

- **Raison** : processeurs optimisés pour récupérer ce type de données (performance)

Conséquence

Sur un processeur 32 bits : Alignement sur les adresses multiples de 4 octets (32bits)
$\longrightarrow$ Assure que la lecture d'un int en une seule lecture de 32 bits

Alignement

Alignement par défaut des types primitifs (processeurs 32 bits)

- *Char* : adresse de 1 octet
- *Short* : adresse multiple de 2 octets
- *Int* : adresse multiple de 4 octets
- *Float* : adresse multiple de 4 octets
- *Double* : adresse multiple de 8 octets
- *Long* : adresse multiple de 8 octets

Possibilité de modifier l'alignement :

- Par l'intermédiaire d'options du compilateur
- Par l'intermédiaire de pragma / hint du compilateur __declspec(align(#))
- Par l'intermédiare de technique dite de 'padding'

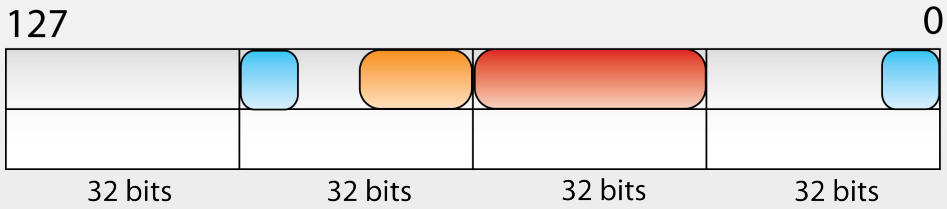
Alignement

Conséquence sur la représentation de structure en mémoire :

- Non nécessairement contiguë
- Grossissement de la taille des structures : perte de place (liée aux trous)

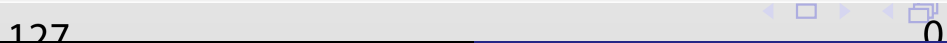
Code

```
struct A {
        char a;
        int b;
        char c;
        short d;
};
```



Code

```
struct B {
        char a, c;
        short d;
        int b;
};
```



Stockage

Différence entre struct et class

- Aucune différence dans la représentation finale (ASM) !
- Notion de visibilté importante en C++ mais non traduite en ASM car non nécessaire pour le stockage (vérification fait en amont)

Notion de pointeur **this**

En ASM pointeur pointant sur le début de la structure de stockage de l'objet

Classe vide

- Attention aux **classes vides** !
- Représenter en mémoire par un octet : permet d'avoir une représentation de la structure et pour que le pointeur this ne point pas sur NULL
- Lorsqu'une classe contenant des données est dérivée d'une classe vide, cette octet symbolique disparaît

Classe de stockage

Chaque variable déclarée possède en C++ une classe de stockage

- auto : pour les variables locales (par défaut)
- extern : déclare une variable sans la définir
- static : rend une définition de variable persistante
- register : demande au compilateur d'utiliser un registre pour stocker cette variable

Remarque : register

- Contrairement aux autres classes de stockage, 'register' n'est qu'une demande fait au compilateur
- Compilateur peut ne pas tenir compte de cette demande
- À utiliser avec parcimonie car
  - Nombre de registre limité
  - Peut nuire aux performance
  - Ne peut servir que pour des données de taille inférieure ou égale à 32bits

Classe de stockage

Code C++

```
void ChangeStatic()
{
        static int staticValue = 0;
        staticValue += 2;
}

extern int externInt;

void TestStock()
{
        int autoValue = 0;
        register int regValue = 2;
        regValue += 5;

        externInt = 0;
        ChangeStatic();
        autoValue = 2 + externInt;
        ChangeStatic();
}
```

Classe de stockage

Code ASM

```
PUBLIC  ?ChangeStatic@@YAXXZ                    ; ChangeStatic

_BSS    SEGMENT
?staticValue@?1??ChangeStatic@@YAXXZ@4HA DD 01H DUP (?) ; 'ChangeStatic'::'2'::staticValue
_BSS    ENDS

_TEXT   SEGMENT
?ChangeStatic@@YAXXZ PROC                       ; ChangeStatic
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR ?staticValue@?1??ChangeStatic@@YAXXZ@4HA
        add     eax, 2
        mov     DWORD PTR ?staticValue@?1??ChangeStatic@@YAXXZ@4HA, eax
        pop     ebp
        ret     0
?ChangeStatic@@YAXXZ ENDP                       ; ChangeStatic
_TEXT   ENDS
```

Classe de stockage

Code ASM

```
PUBLIC ?TestStock@@YAXXZ                    ; TestStock
EXTRN  ?externInt@@3HA:DWORD                ; externInt
_TEXT  SEGMENT
_regValue$ = -8                             ; size = 4
_autoValue$ = -4                            ; size = 4
?TestStock@@YAXXZ PROC                      ; TestStock
       push    ebp
       mov     ebp, esp
       sub     esp, 8
       mov     DWORD PTR _autoValue$[ebp], 0
       mov     DWORD PTR _regValue$[ebp], 2
       mov     eax, DWORD PTR _regValue$[ebp]
       add     eax, 5
       mov     DWORD PTR _regValue$[ebp], eax
       mov     DWORD PTR ?externInt@@3HA, 0     ; externInt
       call    ?ChangeStatic@@YAXXZ             ; ChangeStatic
       mov     ecx, DWORD PTR ?externInt@@3HA   ; externInt
       add     ecx, 2
       mov     DWORD PTR _autoValue$[ebp], ecx
       call    ?ChangeStatic@@YAXXZ             ; ChangeStatic
       mov     esp, ebp
       pop     ebp
       ret     0
?TestStock@@YAXXZ ENDP                      ; TestStock
_TEXT  ENDS
END
```

Qualifieurs

En sus de la classe de stockage, une variable peut posséder un **qualifieur**

- const : pour définir une variable dont la valeur ne devrait jamais changer
- volatile : désigne une variable pouvant être modifiée notamment par une source externe indépendante du programme
- restrict : permet une optimisation pour la gestion des pointeurs (spécifique au C, mais possibilité d'utiliser en C++ voir ci-après)

Important !

Ces qualifiers fournissent des informations au compilateur sur la validité des données manipulées
$\longrightarrow$ Important pour effectuer des optimisations !

Qualifieur

Code C++

```
#include <cstdio>

void Print(const int& _value)
{
	printf("%d\n",_value);
}

void Print( volatile  int& _value)
{
	printf("%d\n",_value);
}

void TestQualifier()
{
	const int constValue = 0;
	 volatile  int volatileValue = 0;

	volatileValue += 2;

	Print(constValue);
	Print(volatileValue);

}
```

Qualifieur

Code ASM

```
PUBLIC ?Print@@YAXABH@Z                         ; Print
_TEXT   SEGMENT
__value$ = 8                                    ; size = 4
?Print@@YAXABH@Z PROC                           ; Print
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR __value$[ebp]
        mov     ecx, DWORD PTR [eax]
        push    ecx
        push    OFFSET $SG3904          ; const segment '%d'
        call    DWORD PTR __imp__printf
        add     esp, 8
        pop     ebp
        ret     0
?Print@@YAXABH@Z ENDP                           ; Print
_TEXT   ENDS

PUBLIC ?Print@@YAXACH@Z                         ; Print
_TEXT   SEGMENT
__value$ = 8                                    ; size = 4
?Print@@YAXACH@Z PROC                           ; Print
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR __value$[ebp]
        mov     ecx, DWORD PTR [eax]
        push    ecx
        push    OFFSET $SG3909          ; const segment '%d'
        call    DWORD PTR __imp__printf
        add     esp, 8
        pop     ebp
        ret     0
?Print@@YAXACH@Z ENDP                           ; Print
_TEXT   ENDS
```

Qualifieur

Code ASM

```
PUBLIC ?TestQualifier@@YAXXZ                    ; TestQualifier
_TEXT   SEGMENT
_volatileValue$ = -8                            ; size = 4
_constValue$ = -4                               ; size = 4
?TestQualifier@@YAXXZ PROC                      ; TestQualifier
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        mov     DWORD PTR _constValue$[ebp], 0
        mov     DWORD PTR _volatileValue$[ebp], 0
        mov     eax, DWORD PTR _volatileValue$[ebp]
        add     eax, 2
        mov     DWORD PTR _volatileValue$[ebp], eax
        lea     ecx, DWORD PTR _constValue$[ebp]
        push    ecx
        call    ?Print@@YAXABH@Z                ; Print
        add     esp, 4
        lea     edx, DWORD PTR _volatileValue$[ebp]
        push    edx
        call    ?Print@@YAXACH@Z                ; Print
        add     esp, 4
        mov     esp, ebp
        pop     ebp
        ret     0
?TestQualifier@@YAXXZ ENDP                      ; TestQualifier
_TEXT   ENDS
END
```

Aliasing de pointeur

Aliasing

Le compilateur ne peut pas connaître de façon automatique tous les pointeurs utilisant une zone mémoire
⇒ Optimisation non possible et manipulation directe de la valeur en mémoire

Exemple du problème

Deux pointeurs pointent sur une une même zone,

1. Les deux sont chargées dans des registres pour être manipulée plus facilement
2. La valeur pointée est modifée par l'un des pointeurs
3. L'autre reste inchangée → État non consistent de la valeur pointée !

Aliasing de pointeur

Techniques de prévention

Objectif : Aider le compilateur

- En copiant les valeurs en local pour assurer au compilateur que la variable n'est pas manipulée ailleurs
- En utilisant des indicateurs de modification / non modification
  - __declspec(noalias)
  - __declspec(restrict)

Aliasing de pointeur : copie locale

Situation problématique

```
void foo1(int *data){
        for(int i=0; i<10; i++) {
                foo2( *data, i);
    }
}
```

Solution

```
void foo1(int *data){
        int localdata = *data;
        for(i=0; i<10; i++) {
                foo2( localdata, i);
    }
}
```

Aliasing de pointeur : noalias

Permet de dire au compilateur : cette fonction est **pure**

- Utilise seulement des variables locales et les arguments
- Utilise seulement le premier niveau d'indirection des pointeurs en argument

→ Permet au compilateur de procéder à des **optimisations plus aggrésives**

Attention

Ceci n'est qu'une information que vous donnez au compilateur.
Comportement indéterminé si non respect du contrat !

Exemple

```
__declspec(noalias) void Foo(float* _a, float * _b, float * _res)
{
    for (int j=0; j<10; j++)
        for (int i=0; i<10; i++)
                    _res[i + j*10] = _a[j] * _b[i];
}
```

Aliasing de pointeur : restrict

- Utilisable uniquement sur les fonctions qui retourne un pointeur.
- Indique que le pointeur retourné est sans alias

→ Permet au compilateur de procéder à des **optimisations plus aggrésives**

Attention

Ceci n'est qu'une information que vous donnez au compilateur.
Comportement indéterminé si non respect du contrat !

Exemple

```
__declspec(restrict) float* Foo()
{
    float  * retval;
    return retval;
}
```

Plan

Rappels ASM sur les fonctions

Qu'est ce qu'une fonction ?

- Ensemble d'instructions stockées à une adresse donnée
- Se termine par l'instruction *ret*
- Appel d'une fonction en assembleur : utilisation de l'instruction *call*

Procedure ASM

```
procedureName PROC
        // asm  instructions
        ret
procedureName ENDP
```

Rappels ASM sur les fonctions

Appel d'une fonction

Composer de 4 phases :

- Prolog : Sauvegarde des registres ESI EDI EBX et EBP à l'appel d'une fonction
- Core : Exécution des instructions de la fonctions
- Stack cleanup : Suppression des paramètres de la pile (soit par la fonction, soit par l'appelant)
- Epilog : Restauration des registres ESI EDI EBX et EBP au retour d'une fonction

Valeur de retour

- Si taille inférieure ou égale à 32 bits : dans le registre EAX
- Sinon passage par la pile

Conventions d'appel

Type de fonction

Le C++ offre plusieurs type de fonctions :

- Fonction normale
- Fonction membre (objet associé à l'appel)
- Fonction membre static (similaire aux fonctions normales)

Type d'appel : Comment passer les paramètres à une fonction ?

Existance de différentes conventions pour passer les paramètres à une fonction :

- __cdecl : convention par défaut pour les fonctions
- __stdcall : passage des paramètres par la pile
- __thiscall : passage implicite du pointeur sur l'objet
- __fastcall : passage des paramètres par les regitres

Conventions d'appel

Convention __cdecl

- Arguments de la fonction empilés de droite à gauche sur la pile
- 'Stack clean up' effectué par l'appelant
- Nom de la fonction décoré avec _funcName

Convention d'appel

Code C++

```
int __cdecl AddByCCall(int _param1, int _param2, int _param3)
{
        return _param1 + _param2 + _param3;
}


void TestConvention()
{
        int a = 1;
        int b = 2;
        int c = 3;

        int res = AddByCCall(a,b,c);
}
```

Convention d'appel

Code ASM

```
; AddByCCall
_TEXT   SEGMENT
__param1$ = 8                                   ; size = 4
__param2$ = 12                                  ; size = 4
__param3$ = 16                                  ; size = 4
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR __param1$[ebp]
        add     eax, DWORD PTR __param2$[ebp]
        add     eax, DWORD PTR __param3$[ebp]
        pop     ebp
        ret     0

; TestConvention
_res$2559 = -16                                 ; size = 4
_c$ = -12                                       ; size = 4
_b$ = -8                                        ; size = 4
_a$ = -4                                        ; size = 4

        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        call    ?AddByCCall@@YAHHHH@Z           ; AddByCCall
        add     esp, 12                         ; 0000000cH
        mov     DWORD PTR _res$2559[ebp], eax
```

Convention d'appel

Convention __stdcall

- Argments de la fonction empilés de droite à gauche sur la pile
- Stack clean up effectué par la fonction
- Nom de la fonction décoré avec _funcName@numberOfBytesOfStackRequired

Remarques

1. Pour les fonctions au nombre de paramètres variables, obligation d'utiliser la convention __cdecl car seul l'appelant connait le nombre réel de paramètres
2. Convention __stdcall produit des exécutables plus petits car dépilement fait au retour

Convention d'appel

Code C++

```
int __stdcall AddByStdCall( int _param1, int _param2, int _param3)
{
        return _param1 + _param2 + _param3;
}

void TestConvention()
{
        int a = 1;
        int b = 2;
        int c = 3;
        int res = AddByStdCall(a,b,c);
}
```

Convention d'appel

Code ASM

```
; AddByStdCall
_TEXT   SEGMENT
__param1$ = 8                                   ; size = 4
__param2$ = 12                                  ; size = 4
__param3$ = 16                                  ; size = 4
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR __param1$[ebp]
        add     eax, DWORD PTR __param2$[ebp]
        add     eax, DWORD PTR __param3$[ebp]
        pop     ebp
        ret     12                              ; 0000000cH

; TestConvention
_res$2560 = -20                                 ; size = 4
_c$ = -12                                       ; size = 4
_b$ = -8                                        ; size = 4
_a$ = -4                                        ; size = 4
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        call    ?AddByStdCall@@YGHHHH@Z         ; AddByStdCall
        mov     DWORD PTR _res$2560[ebp], eax
```

Convention d'appel

Convention __thiscall

- Utiliser pour les fonctions membres
- Passage implicite de l'objet dans le registre ECX
- Argments de la fonction empilés de droite à gauche sur la pile
- Stack clean up effectué par la fonction
- Nom de la fonction décoré avec _funcName@numberOfBytesOfStackRequired

Convention d'appel

Code C++

```
struct A
{
        int __thiscall AddByThisCall( int _param1, int _param2, int _param3)
        {
                return _param1 + _param2 + _param3;
        }
};

void TestConvention()
{
        int a = 1;
        int b = 2;
        int c = 3;
        A s;
        int res = s.AddByThisCall(a,b,c);
}
```

Convention d'appel

Code ASM

```
_this$ = -4                                    ; size = 4
__param1$ = 8                                  ; size = 4
__param2$ = 12                                 ; size = 4
__param3$ = 16                                 ; size = 4
; _this$ = ecx
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _this$[ebp], ecx
        mov     eax, DWORD PTR __param1$[ebp]
        add     eax, DWORD PTR __param2$[ebp]
        add     eax, DWORD PTR __param3$[ebp]
        mov     esp, ebp
        pop     ebp
        ret     12                             ; 0000000cH
```

Convention d'appel

Code ASM

```
; TestConvention
_TEXT  SEGMENT
_c$ = -20                                   ; size = 4
_b$ = -16                                   ; size = 4
_s$ = -9                                    ; size = 1
_a$ = -8                                    ; size = 4
_res$ = -4                                  ; size = 4
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     ecx, DWORD PTR _b$[ebp]
        push    ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        lea     ecx, DWORD PTR _s$[ebp]
        call    ?AddByThisCall@A@@QAEHHHH@Z     ; A::AddByThisCall
        mov     DWORD PTR _res$[ebp], eax
```

Convention d'appel

Convention __fastcall

- Les deux premiers arguments de taille inférieure ou égale à 32bits sont passés par ECX et EDX
- Le reste des paramètres est empilé sur la pile
- Stack cleanup est réalisé par la fonction
- Nom de la fonction décoré avec @funcName@numberOfBytesOfStackRequired

Remarque

Permet un passage de paramètres plus efficace donc un appel de fonction plus performant

Convention d'appel

Code C++

```
int __fastcall AddByFastCall( int _param1, int _param2, int _param3)
{
        return _param1 + _param2 + _param3;
}

void TestConvention()
{
        int a = 1;
        int b = 2;
        int c = 3;
        int res = AddByFastCall(a,b,c);
}
```

Convention d'appel

Code ASM

```
__param2$ = -8                          ; size = 4
__param1$ = -4                          ; size = 4
__param3$ = 8                           ; size = 4
?AddByFastCall@@YIHHHH@Z PROC           ; AddByFastCall
; __param1$ = ecx / __param2$ = edx
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        mov     DWORD PTR __param2$[ebp], edx
        mov     DWORD PTR __param1$[ebp], ecx
        mov     eax, DWORD PTR __param1$[ebp]
        add     eax, DWORD PTR __param2$[ebp]
        add     eax, DWORD PTR __param3$[ebp]
        mov     esp, ebp
        pop     ebp
        ret     4

; TestConvention
_res$2561 = -24                         ; size = 4
_c$ = -12                               ; size = 4
_b$ = -8                                ; size = 4
_a$ = -4                                ; size = 4
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        mov     edx, DWORD PTR _b$[ebp]
        mov     ecx, DWORD PTR _a$[ebp]
        call    ?AddByFastCall@@YIHHHH@Z    ; AddByFastCall
        mov     DWORD PTR _res$2561[ebp], eax
```

Passage de paramètre

Passage des paramètres d'une fonction peut se faire de différentes façons

- Passage par référence
- Passage par valeur
- Passage par pointeur

Règles générales : types non primitifs

- Si non modification : toujours passer les références constantes
- Si modification : passer par référence simple

Règles générales : types primitifs

- Si non modification : passer par valeur
- Si modification : passer par référence

Remarque

Si paramètre peut ne par exister : passage par pointeur

$\rightarrow$ Quel impact sur le code généré ?

Passage par valeur

Code C++

```
#include <cstdio>

struct BigObj{
        BigObj(int _value):Data0(_value),Data1(_value), Data2(_value), Data3(_value), Data4(_value) {}
        int Data0, Data1, Data2, Data3, Data4;
};

void ByValue(int _p1, BigObj _p2)
{
        int res = _p1 + _p2.Data0 + _p2.Data1 + _p2.Data2 + _p2.Data3 + _p2.Data4;
}

void TestParameterTransfert()
{
        int a;
        BigObj b(0);
        ByValue(a,b);
}
```

Passage par valeur

Code C++

```
PUBLIC ?ByValue@@YAXHUBigObj@@@Z                    ; ByValue
EXTRN  __imp__printf:PROC
_TEXT  SEGMENT
_res$ = -4                                          ; size = 4
__p1$ = 8                                           ; size = 4
__p2$ = 12                                          ; size = 20
?ByValue@@YAXHUBigObj@@@Z PROC                      ; ByValue
       push   ebp
       mov    ebp, esp
       push   ecx
       mov    eax, DWORD PTR __p1$[ebp]
       add    eax, DWORD PTR __p2$[ebp]
       add    eax, DWORD PTR __p2$[ebp+4]
       add    eax, DWORD PTR __p2$[ebp+8]
       add    eax, DWORD PTR __p2$[ebp+12]
       add    eax, DWORD PTR __p2$[ebp+16]
       mov    DWORD PTR _res$[ebp], eax
       mov    ecx, DWORD PTR _res$[ebp]
       push   ecx
       push   OFFSET $SG3921
       call   DWORD PTR __imp__printf
       add    esp, 8
       mov    esp, ebp
       pop    ebp
       ret    0
?ByValue@@YAXHUBigObj@@@Z ENDP                      ; ByValue
_TEXT  ENDS
```

Passage par valeur

Code C++

```
; TestParameterTransfert
_b$ = -24                                   ; size = 20
_a$ = -4                                    ; size = 4
        mov     DWORD PTR _b$[ebp], 0
        mov     DWORD PTR _b$[ebp+4], 0
        mov     DWORD PTR _b$[ebp+8], 0
        mov     DWORD PTR _b$[ebp+12], 0
        mov     DWORD PTR _b$[ebp+16], 0

        add     esp, -12                    ; fffffff4H
        mov     eax, esp
        mov     ecx, DWORD PTR _b$[ebp]
        mov     DWORD PTR [eax], ecx
        mov     edx, DWORD PTR _b$[ebp+4]
        mov     DWORD PTR [eax+4], edx
        mov     ecx, DWORD PTR _b$[ebp+8]
        mov     DWORD PTR [eax+8], ecx
        mov     edx, DWORD PTR _b$[ebp+12]
        mov     DWORD PTR [eax+12], edx
        mov     ecx, DWORD PTR _b$[ebp+16]
        mov     DWORD PTR [eax+16], ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        call    ?ByValue@@YAXHUBigObj@@@Z   ; ByValue
```

Passage par valeur constante

Code C++

```
#include <cstdio>

struct BigObj{
        BigObj(int _value):Data0(_value),Data1(_value), Data2(_value), Data3(_value), Data4(_value) {}
        int Data0, Data1, Data2, Data3, Data4;
};

void ByConstValue(const int _p1, const BigObj _p2)
{
        int res = _p1 + _p2.Data0 + _p2.Data1 + _p2.Data2 + _p2.Data3 + _p2.Data4;
}

void TestParameterTransfert()
{
        int a;
        BigObj b(0);
        ByConstValue(a,b);
}
```

Passage par valeur constante

Code C++

```
PUBLIC ?ByConstValue@@YAXHUBigObj@@@Z                ; ByConstValue
_TEXT   SEGMENT
_res$ = -4                                           ; size = 4
__p1$ = 8                                            ; size = 4
__p2$ = 12                                           ; size = 20
?ByConstValue@@YAXHUBigObj@@@Z PROC                  ; ByConstValue
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     eax, DWORD PTR __p1$[ebp]
        add     eax, DWORD PTR __p2$[ebp]
        add     eax, DWORD PTR __p2$[ebp+4]
        add     eax, DWORD PTR __p2$[ebp+8]
        add     eax, DWORD PTR __p2$[ebp+12]
        add     eax, DWORD PTR __p2$[ebp+16]
        mov     DWORD PTR _res$[ebp], eax
        mov     ecx, DWORD PTR _res$[ebp]
        push    ecx
        push    OFFSET $SG3927
        call    DWORD PTR __imp__printf
        add     esp, 8
        mov     esp, ebp
        pop     ebp
        ret     0
?ByConstValue@@YAXHUBigObj@@@Z ENDP                  ; ByConstValue
_TEXT   ENDS
```

Passage par valeur constante

Code C++

```
; TestParameterTransfert
_TEXT   SEGMENT
_b$ = -24                                       ; size = 20
_a$ = -4                                        ; size = 4
        mov     DWORD PTR _b$[ebp], 0
        mov     DWORD PTR _b$[ebp+4], 0
        mov     DWORD PTR _b$[ebp+8], 0
        mov     DWORD PTR _b$[ebp+12], 0
        mov     DWORD PTR _b$[ebp+16], 0
        add     esp, 4
        mov     eax, esp
        mov     ecx, DWORD PTR _b$[ebp]
        mov     DWORD PTR [eax], ecx
        mov     edx, DWORD PTR _b$[ebp+4]
        mov     DWORD PTR [eax+4], edx
        mov     ecx, DWORD PTR _b$[ebp+8]
        mov     DWORD PTR [eax+8], ecx
        mov     edx, DWORD PTR _b$[ebp+12]
        mov     DWORD PTR [eax+12], edx
        mov     ecx, DWORD PTR _b$[ebp+16]
        mov     DWORD PTR [eax+16], ecx
        mov     edx, DWORD PTR _a$[ebp]
        push    edx
        call    ?ByConstValue@@YAXHUBigObj@@@Z      ; ByConstValue
        add     esp, 24                             ; 00000018H
```

Passage par référence

Code C++

```
#include <cstdio>

struct BigObj{
        BigObj(int _value):Data0(_value),Data1(_value), Data2(_value), Data3(_value), Data4(_value) {}
        int Data0, Data1, Data2, Data3, Data4;
};

void ByReference(int& _p1, BigObj& _p2)
{
        int res = _p1 + _p2.Data0 + _p2.Data1 + _p2.Data2 + _p2.Data3 + _p2.Data4;
}

void TestParameterTransfert()
{
        int a;
        BigObj b(0);
        ByReference(a,b);
}
```

Passage par référence

Code C++

```
PUBLIC ?ByReference@@YAXAAHAAUBigObj@@@Z          ; ByReference
; Function compile flags: /Odtp
_TEXT   SEGMENT
_res$ = -4                                        ; size = 4
__p1$ = 8                                         ; size = 4
__p2$ = 12                                        ; size = 4
?ByReference@@YAXAAHAAUBigObj@@@Z PROC            ; ByReference
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     eax, DWORD PTR __p1$[ebp]
        mov     ecx, DWORD PTR [eax]
        mov     edx, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [edx]
        mov     eax, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [eax+4]
        mov     edx, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [edx+8]
        mov     eax, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [eax+12]
        mov     edx, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [edx+16]
        mov     DWORD PTR _res$[ebp], ecx
        mov     eax, DWORD PTR _res$[ebp]
        push    eax
        push    OFFSET $SG3933
        call    DWORD PTR __imp__printf
        add     esp, 8
        mov     esp, ebp
        pop     ebp
        ret     0
?ByReference@@YAXAAHAAUBigObj@@@Z ENDP            ; ByReference
_TEXT   ENDS
```

Passage par référence

Code C++

```
; TestParameterTransfert
_b$ = -24                                       ; size = 20
_a$ = -4                                        ; size = 4
        mov     DWORD PTR _b$[ebp], 0
        mov     DWORD PTR _b$[ebp+4], 0
        mov     DWORD PTR _b$[ebp+8], 0
        mov     DWORD PTR _b$[ebp+12], 0
        mov     DWORD PTR _b$[ebp+16], 0

        lea     eax, DWORD PTR _b$[ebp]
        push    eax
        lea     ecx, DWORD PTR _a$[ebp]
        push    ecx
        call    ?ByReference@@YAXAAHAAUBigObj@@@Z    ; ByReference
        add     esp, 8
```

Passage par référence constante

Code C++

```
#include <cstdio>

struct BigObj{
        BigObj(int _value):Data0(_value),Data1(_value), Data2(_value), Data3(_value), Data4(_value) {}
        int Data0, Data1, Data2, Data3, Data4;
};

void ByConstReference(const int& _p1, const BigObj& _p2)
{
        int res = _p1 + _p2.Data0 + _p2.Data1 + _p2.Data2 + _p2.Data3 + _p2.Data4;
}

void TestParameterTransfert()
{
        int a;
        BigObj b(0);
        ByConstReference(a,b);
}
```

Passage par référence constante

Code C++

```
PUBLIC ?ByConstReference@@YAXABHABUBigObj@@@Z      ; ByConstReference
_TEXT   SEGMENT
_res$ = -4                                         ; size = 4
__p1$ = 8                                          ; size = 4
__p2$ = 12                                         ; size = 4
?ByConstReference@@YAXABHABUBigObj@@@Z PROC        ; ByConstReference
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     eax, DWORD PTR __p1$[ebp]
        mov     ecx, DWORD PTR [eax]
        mov     edx, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [edx]
        mov     eax, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [eax+4]
        mov     edx, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [edx+8]
        mov     eax, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [eax+12]
        mov     edx, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [edx+16]
        mov     DWORD PTR _res$[ebp], ecx
        mov     eax, DWORD PTR _res$[ebp]
        push    eax
        push    OFFSET $SG3939
        call    DWORD PTR __imp__printf
        add     esp, 8
        mov     esp, ebp
        pop     ebp
        ret     0
?ByConstReference@@YAXABHABUBigObj@@@Z ENDP        ; ByConstReference
_TEXT   ENDS
```

Passage par référence constante

Code C++

```
; TestParameterTransfert
_b$ = -24                                               ; size = 20
_a$ = -4                                                ; size = 4
        mov     DWORD PTR _b$[ebp], 0
        mov     DWORD PTR _b$[ebp+4], 0
        mov     DWORD PTR _b$[ebp+8], 0
        mov     DWORD PTR _b$[ebp+12], 0
        mov     DWORD PTR _b$[ebp+16], 0

        lea     edx, DWORD PTR _b$[ebp]
        push    edx
        lea     eax, DWORD PTR _a$[ebp]
        push    eax
        call    ?ByConstReference@@YAXABHABUBigObj@@@Z ; ByConstReference
        add     esp, 8
```

Passage par pointeur

Code C++

```
#include <cstdio>

struct BigObj{
        BigObj(int _value):Data0(_value),Data1(_value), Data2(_value), Data3(_value), Data4(_value) {}
        int Data0, Data1, Data2, Data3, Data4;
};

void ByPointer(int* _p1, BigObj* _p2)
{
        int res = *_p1 + _p2->Data0 + _p2->Data1 + _p2->Data2 + _p2->Data3 + _p2->Data4;
}

void TestParameterTransfert()
{
        int a;
        BigObj b(0);
        ByPointer(&a,&b);
}
```

Passage par pointeur

Code C++

```
PUBLIC ?ByPointer@@YAXPAHPAUBigObj@@@Z                ; ByPointer
_TEXT   SEGMENT
_res$ = -4                                            ; size = 4
__p1$ = 8                                             ; size = 4
__p2$ = 12                                            ; size = 4
?ByPointer@@YAXPAHPAUBigObj@@@Z PROC                  ; ByPointer
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     eax, DWORD PTR __p1$[ebp]
        mov     ecx, DWORD PTR [eax]
        mov     edx, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [edx]
        mov     eax, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [eax+4]
        mov     edx, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [edx+8]
        mov     eax, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [eax+12]
        mov     edx, DWORD PTR __p2$[ebp]
        add     ecx, DWORD PTR [edx+16]
        mov     DWORD PTR _res$[ebp], ecx
        mov     eax, DWORD PTR _res$[ebp]
        push    eax
        push    OFFSET $SG3945
        call    DWORD PTR __imp__printf
        add     esp, 8
        mov     esp, ebp
        pop     ebp
        ret     0
?ByPointer@@YAXPAHPAUBigObj@@@Z ENDP                  ; ByPointer
_TEXT   ENDS
```

Passage par pointeur

Code C++

```
; TestParameterTransfert
_b$ = -24                                       ; size = 20
_a$ = -4                                        ; size = 4
        mov     DWORD PTR _b$[ebp], 0
        mov     DWORD PTR _b$[ebp+4], 0
        mov     DWORD PTR _b$[ebp+8], 0
        mov     DWORD PTR _b$[ebp+12], 0
        mov     DWORD PTR _b$[ebp+16], 0

        lea     ecx, DWORD PTR _b$[ebp]
        push    ecx
        lea     edx, DWORD PTR _a$[ebp]
        push    edx
        call    ?ByPointer@@YAXPAHPAUBigObj@@@Z     ; ByPointer
        add     esp, 8
```

Valeur de retour

Rappels sur les formes de retour possibles :

- Obligation de retour par valeur si l'allocation a été réalisée sur la pile
- Possibilité de retour par référence uniquement si l'élément existe après le retour de la fonction (objet alloué sur le tas, donnée membre d'un objet

Problème

Création de temporaires entre la fonction et l'initialisation de l'objet au retour de la fonction

Valeur de retour

Optimisation de retour

Optimisation de la valeur de retour réalisée par le compilateur

Code C++

```
#include <cstdio>
struct BigObj {
        int Data0, Data1, Data2, Data3, Data4;
};

BigObj CreateBigObject(){
        BigObj res;
        res.Data0 = 0;
        res.Data1 = 1;
        res.Data2 = 2;
        res.Data3 = 3;
        res.Data4 = 4;
        return res;
}

void TestReturnOptim()
{
        BigObj res = CreateBigObject();
}
```

Valeur de retour

Exemple : Sans activation des optimisations

Code ASM

```
PUBLIC ?CreateBigObject@@YA?AUBigObj@@XZ          ; CreateBigObject
_TEXT   SEGMENT
_res$ = -20                                       ; size = 20
$T3773 = 8                                        ; size = 4
?CreateBigObject@@YA?AUBigObj@@XZ PROC            ; CreateBigObject
        push    ebp
        mov     ebp, esp
        sub     esp, 20                           ; 00000014H
        mov     DWORD PTR _res$[ebp], 0
        mov     DWORD PTR _res$[ebp+4], 1
        mov     DWORD PTR _res$[ebp+8], 2
        mov     DWORD PTR _res$[ebp+12], 3
        mov     DWORD PTR _res$[ebp+16], 4
        mov     eax, DWORD PTR $T3773[ebp]
        mov     ecx, DWORD PTR _res$[ebp]
        mov     DWORD PTR [eax], ecx
        mov     edx, DWORD PTR _res$[ebp+4]
        mov     DWORD PTR [eax+4], edx
        mov     ecx, DWORD PTR _res$[ebp+8]
        mov     DWORD PTR [eax+8], ecx
        mov     edx, DWORD PTR _res$[ebp+12]
        mov     DWORD PTR [eax+12], edx
        mov     ecx, DWORD PTR _res$[ebp+16]
        mov     DWORD PTR [eax+16], ecx
        mov     eax, DWORD PTR $T3773[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
?CreateBigObject@@YA?AUBigObj@@XZ ENDP            ; CreateBigObject
_TEXT   ENDS
```

Valeur de retour

Code ASM

```
; TestReturnOptim
$T3776 = -64                                    ; size = 20
$T3775 = -40                                    ; size = 20
_res$ = -20                                     ; size = 20
        lea     eax, DWORD PTR $T3776[ebp]
        push    eax
        call    ?CreateBigObject@@YA?AUBigObj@@XZ   ; CreateBigObject
        add     esp, 4
        mov     ecx, DWORD PTR [eax]
        mov     DWORD PTR $T3775[ebp], ecx
        mov     edx, DWORD PTR [eax+4]
        mov     DWORD PTR $T3775[ebp+4], edx
        mov     ecx, DWORD PTR [eax+8]
        mov     DWORD PTR $T3775[ebp+8], ecx
        mov     edx, DWORD PTR [eax+12]
        mov     DWORD PTR $T3775[ebp+12], edx
        mov     eax, DWORD PTR [eax+16]
        mov     DWORD PTR $T3775[ebp+16], eax
        mov     ecx, DWORD PTR $T3775[ebp]
        mov     DWORD PTR _res$[ebp], ecx
        mov     edx, DWORD PTR $T3775[ebp+4]
        mov     DWORD PTR _res$[ebp+4], edx
        mov     eax, DWORD PTR $T3775[ebp+8]
        mov     DWORD PTR _res$[ebp+8], eax
        mov     ecx, DWORD PTR $T3775[ebp+12]
        mov     DWORD PTR _res$[ebp+12], ecx
        mov     edx, DWORD PTR $T3775[ebp+16]
        mov     DWORD PTR _res$[ebp+16], edx
```

Valeur de retour

Exemple : Avec activation des optimisations

Code ASM

```
PUBLIC ?CreateBigObject@@YA?AUBigObj@@XZ          ; CreateBigObject
_TEXT   SEGMENT
$T3773 = 8                                        ; size = 4
?CreateBigObject@@YA?AUBigObj@@XZ PROC            ; CreateBigObject
        mov     eax, DWORD PTR $T3773[esp-4]
        mov     DWORD PTR [eax], 0
        mov     DWORD PTR [eax+4], 1
        mov     DWORD PTR [eax+8], 2
        mov     DWORD PTR [eax+12], 3
        mov     DWORD PTR [eax+16], 4
        ret     0
?CreateBigObject@@YA?AUBigObj@@XZ ENDP            ; CreateBigObject
_TEXT   ENDS

; TestReturnOptim
_TEXT   SEGMENT
_res$ = -40                                       ; size = 20
$T3778 = -20                                      ; size = 20
        lea     eax, DWORD PTR $T3778[esp+40]
        push    eax
        call    ?CreateBigObject@@YA?AUBigObj@@XZ ; CreateBigObject
        mov     edx, DWORD PTR [eax+4]
        mov     ecx, DWORD PTR [eax]
        mov     DWORD PTR _res$[esp+48], edx
        mov     edx, DWORD PTR [eax+8]
        mov     DWORD PTR _res$[esp+52], edx
        mov     edx, DWORD PTR [eax+12]
        mov     eax, DWORD PTR [eax+16]
```

Inline

Définition

Instruction indiquant au compilateur de recopier le code de la fonction plutôt que d'effectuer son appel

Principaux intérêts

- Supprime l'instruction d'appel de la fonction
- Supprime le passage des paramètres
- Évite les phases de prolog, épilog

$\longrightarrow$ Permet d'alléger le coût d'utilisation d'une fonction

Problème

Surcharge le code et donc peut causer des effets de bord : défaut de cache d'instructions plus fréquent ...

Inline

Quand inliner vos fonctions ?

- Nombre d'instructions moins important que le nombre d'instructions nécessaire pour faire l'appel de la fonction
- Coût de *call* / *ret* plus important que la fonction elle-même
- Utilisation unique dans tout le projet, quelque soit la taille de la fonction

Plan

Rappels

Contexte

En C++ possibilité de faire de l'héritage entre les classes selon plusieurs modalités :

- Héritage simple
- Héritage multiple

Principaux intérêt : Factorisation de code, Abstraction de comportement, ...

En ASM

Impacts sur le code ASM généré

- Lourdeur ?
- Transparent ?

Constructeur et héritage

Exemple : Construction d'une classe dérivée

En C++

```
#include <cstdio>

class Mother {
public:
        Mother(int _m):DataMother(_m){}
private:
        int DataMother;
};

class Child : public Mother{
public:
        Child(int _m, int _c):Mother(_m),DataChild(_c){}
private:
        int DataChild;
};

void TestInheritanceBuild(){
        Mother m(0);
        Child c(1,2);
}
```

Constructeur et héritage

En ASM

```
; TestInheritanceBuild
_m$ = -12                                   ; size = 4
_c$ = -8                                    ; size = 8
        mov     DWORD PTR _m$[ebp], 0
        mov     DWORD PTR _c$[ebp], 1
        mov     DWORD PTR _c$[ebp+4], 2
```

$\longrightarrow$ Aucun changement dans la structure de la classe

Polymorphisme

Héritage avec polymorphisme

Définition de méhode dites virtuelles :

- Utilisation du comportement des classes dérivées avec une interface d'ancêtre

$\longrightarrow$ Besoin de connaître le comportement à l'exécution : les **v-Table**

Définition : v-Table

- Tableau contenant des pointeurs sur les fonctions associées à une classe

Comment est stocké en mémoire cette **v-table** ?

- Une V-Table est définie par classe
- Chaque objet possède un pointeur sur cette vtable

Héritage avec polymorphisme

En C++

```
class Mother {
public :
        void Foo1() { ++DataMother; }
        virtual void Foo2()  { DataMother += 2; }
        virtual void Foo3()  { DataMother += 3; }
private :
        int DataMother;
};

class Child : public Mother{
public :
        virtual void Foo2()  { DataChild += 2; }
        virtual void Foo3()  { DataChild += 3; }
        void Foo4()  { DataChild += 4; }
private :
        int DataChild;
};

void CallVirtual(Mother& _obj){
        _obj.Foo2();
        _obj.Foo3();
}

void TestInheritance(){
        Mother m;
        m.Foo1();
        CallVirtual(m);
        Child c;
        c.Foo1();
        CallVirtual(c);
        c.Foo4();
}
```

Héritage avec polymorphisme

En ASM

```
; TestInheritance
_m$ = -20                                               ; size = 8
_c$ = -12                                               ; size = 12
        mov     DWORD PTR _m$[ebp], OFFSET ??_7Mother@@6B@
        mov     eax, DWORD PTR _m$[ebp+4]
        add     eax, 1
        mov     DWORD PTR _m$[ebp+4], eax
        lea     ecx, DWORD PTR _m$[ebp]
        push    ecx
        call    ?CallVirtual@@YAXAAVMother@@@Z          ; CallVirtual
        add     esp, 4
        mov     DWORD PTR _c$[ebp], OFFSET ??_7Mother@@6B@
        mov     DWORD PTR _c$[ebp], OFFSET ??_7Child@@6B@
        mov     edx, DWORD PTR _c$[ebp+4]
        add     edx, 1
        mov     DWORD PTR _c$[ebp+4], edx
        lea     eax, DWORD PTR _c$[ebp]
        push    eax
        call    ?CallVirtual@@YAXAAVMother@@@Z          ; CallVirtual
        add     esp, 4
        mov     ecx, DWORD PTR _c$[ebp+8]
        add     ecx, 4
        mov     DWORD PTR _c$[ebp+8], ecx
```

Héritage avec polymorphisme

En ASM

```
;       COMDAT ??_7Mother@@6B@
CONST   SEGMENT
??_7Mother@@6B@ DD FLAT:?Foo2@Mother@@UAEXXZ        ; Mother::`vftable
        DD      FLAT:?Foo3@Mother@@UAEXXZ
CONST   ENDS

;       COMDAT ??_7Child@@6B@
CONST   SEGMENT
??_7Child@@6B@ DD FLAT:?Foo2@Child@@UAEXXZ          ; Child::`vftable
        DD      FLAT:?Foo3@Child@@UAEXXZ
CONST   ENDS

PUBLIC  ??_7Child@@6B@                              ; Child::`vftable
PUBLIC  ??_7Mother@@6B@                             ; Mother::`vftable
PUBLIC  ?TestInheritance@@YAXXZ                     ; TestInheritance
PUBLIC  ?Foo2@Mother@@UAEXXZ                        ; Mother::Foo2
PUBLIC  ?Foo3@Mother@@UAEXXZ                        ; Mother::Foo3
PUBLIC  ?Foo2@Child@@UAEXXZ                         ; Child::Foo2
PUBLIC  ?Foo3@Child@@UAEXXZ                         ; Child::Foo3

; CallVirtual  ?CallVirtual@@YAXAAVMother@@@Z              ; CallVirtual
__obj$ = 8                                          ; size = 4
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR __obj$[ebp]
        mov     edx, DWORD PTR [eax]
        mov     ecx, DWORD PTR __obj$[ebp]
        mov     eax, DWORD PTR [edx]
        call    eax
        mov     ecx, DWORD PTR __obj$[ebp]
        mov     edx, DWORD PTR [ecx]
        mov     ecx, DWORD PTR __obj$[ebp]
        mov     eax, DWORD PTR [edx+4]
```

Héritage avec polymorphisme

En ASM

```
; Mother::Foo2 ?Foo2@Mother@@UAEXXZ
_this$ = -4                                   ; size = 4 ; _this$ = ecx
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _this$[ebp], ecx
        mov     eax, DWORD PTR _this$[ebp]
        mov     ecx, DWORD PTR [eax+4]
        add     ecx, 2
        mov     edx, DWORD PTR _this$[ebp]
        mov     DWORD PTR [edx+4], ecx
        mov     esp, ebp
        pop     ebp
        ret     0

; Mother::Foo3 COMDAT ?Foo3@Mother@@UAEXXZ
_this$ = -4                                   ; size = 4 ; _this$ = ecx
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _this$[ebp], ecx
        mov     eax, DWORD PTR _this$[ebp]
        mov     ecx, DWORD PTR [eax+4]
        add     ecx, 3
        mov     edx, DWORD PTR _this$[ebp]
        mov     DWORD PTR [edx+4], ecx
        mov     esp, ebp
        pop     ebp
        ret     0
```

Héritage avec polymorphisme

En ASM

```
; Child::Foo2, ?Foo2@Child@@UAEXXZ
_this$ = -4                             ; size = 4 ; _this$ = ecx
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _this$[ebp], ecx
        mov     eax, DWORD PTR _this$[ebp]
        mov     ecx, DWORD PTR [eax+8]
        add     ecx, 2
        mov     edx, DWORD PTR _this$[ebp]
        mov     DWORD PTR [edx+8], ecx
        mov     esp, ebp
        pop     ebp
        ret     0

; Child::Foo3, ?Foo3@Child@@UAEXXZ
_this$ = -4                             ; size = 4 ; _this$ = ecx
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _this$[ebp], ecx
        mov     eax, DWORD PTR _this$[ebp]
        mov     ecx, DWORD PTR [eax+8]
        add     ecx, 3
        mov     edx, DWORD PTR _this$[ebp]
        mov     DWORD PTR [edx+8], ecx
        mov     esp, ebp
        pop     ebp
        ret     0
```

Classe abstraite

Possibilité en C++ de définir des classes non **instanciables**

- Classe interface
- Classe abstraite

Problème

- Dans hiérarchie d'héritage importante : beaucoup de classes non instanciables

$\longrightarrow$ Définition de la v-table non utilisées : perte de place !

Solution

Pour ces classes, possibilité de supprimer la v-table : __decl(novirtual)

Suppression de v-table

En C++

```
class __declspec(novtable) Interface {
public:
        virtual void Foo1()=0 { int i; i += 1; };
        virtual void Foo2()=0 { int i; i += 2; };
};

class Object : public Interface{
public:
       Object():Interface() { DataChild = 0; }
        virtual void Foo1()  { DataChild += 1; }
        virtual void Foo2()  { DataChild += 2; }
private:
        int DataChild;
};

void TestFastInheritance(){
       Object o;
       o.Foo1();
       o.Foo2();
}
```

Suppression de v-table

En ASM

```
;Object::Foo1 COMDAT ?Foo1@Object@@UAEXXZ
_this$ = -4                                   ; size = 4 ; _this$ = ecx
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _this$[ebp], ecx
        mov     eax, DWORD PTR _this$[ebp]
        mov     ecx, DWORD PTR [eax+4]
        add     ecx, 1
        mov     edx, DWORD PTR _this$[ebp]
        mov     DWORD PTR [edx+4], ecx
        mov     esp, ebp
        pop     ebp
        ret     0

;Object::Foo2   COMDAT ?Foo2@Object@@UAEXXZ
_this$ = -4                                   ; size = 4 ; _this$ = ecx
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _this$[ebp], ecx
        mov     eax, DWORD PTR _this$[ebp]
        mov     ecx, DWORD PTR [eax+4]
        add     ecx, 2
        mov     edx, DWORD PTR _this$[ebp]
        mov     DWORD PTR [edx+4], ecx
        mov     esp, ebp
        pop     ebp
        ret     0
```

Suppression de v-table

En ASM

```
PUBLIC ??_7Object@@6B@                          ; Object::`vftable
PUBLIC ?TestFastInheritance@@YAXXZ              ; TestFastInheritance
PUBLIC ?Foo1@Object@@UAEXXZ                     ; Object::Foo1
PUBLIC ?Foo2@Object@@UAEXXZ                     ; Object::Foo2

;       COMDAT ??_7Object@@6B@
CONST   SEGMENT
??_7Object@@6B@ DD FLAT:?Foo1@Object@@UAEXXZ    ; Object::`vftable
        DD      FLAT:?Foo2@Object@@UAEXXZ
CONST   ENDS

; TestFastInheritance
_TEXT   SEGMENT
_o$ = -8                                        ; size = 8
        mov     DWORD PTR _o$[ebp], OFFSET ??_7Object@@6B@
        mov     DWORD PTR _o$[ebp+4], 0
        mov     eax, DWORD PTR _o$[ebp+4]
        add     eax, 1
        mov     DWORD PTR _o$[ebp+4], eax
        mov     ecx, DWORD PTR _o$[ebp+4]
        add     ecx, 2
        mov     DWORD PTR _o$[ebp+4], ecx
```

Plan

Rappels opérateur en C++

Context

Possibilité en C++ de surcharger les opérateurs pour des types non primitifs

Exemple : Surcharge de l'opérateur +

Exemple

```
struct A { A(int _val=0):Data(_val){}; int Data; };

A operator+(const A& _obj1, const A& _obj2)
{
        A temp;
        temp.Data = _obj1.Data + _obj2.Data;
        return temp;
}

void TestOperatorAdd()
{
        A a(1);
        A b(2);
        A c = a + b;
}
```

Rappels opérateur en C++

Possibilité de sucharger nombreux opérateurs

- $+, -, *, /, =$
- $++, --$
- $*, ->$
- [], ()
- ...

Avantage : Permet d'avoir un code plus concis et donc plus lisible

À la compilation que se passe-t-il ?

Remplacement de tous les opérateurs par leur fonction originelle

- A c = a + b ; ⇒ A c(operator+(a,b)) ;

Traduction opérateur en ASM

Exemple : Traduction en ASM du code de l'opérateur d'addition

Code C++

```
struct A { A(int _val=0):Data(_val){}; int Data; };

A operator+(const A& _obj1, const A& _obj2)
{
        A temp;
        temp.Data = _obj1.Data + _obj2.Data;
        return temp;
}

void TestOperatorAdd()
{
        A a(1);
        A b(2);
        A c = a + b;
}
```

Tradiction opérateur en ASM

Code ASM

```
; operator+ ??H@YA?AUA@@ABU0@0@Z
_temp$ = -4                                   ; size = 4
___$ReturnUdt$ = 8                            ; size = 4
__obj1$ = 12                                  ; size = 4
__obj2$ = 16                                  ; size = 4
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _temp$[ebp], 0
        mov     eax, DWORD PTR __obj1$[ebp]
        mov     ecx, DWORD PTR [eax]
        mov     edx, DWORD PTR __obj2$[ebp]
        add     ecx, DWORD PTR [edx]
        mov     DWORD PTR _temp$[ebp], ecx
        mov     eax, DWORD PTR ___$ReturnUdt$[ebp]
        mov     ecx, DWORD PTR _temp$[ebp]
        mov     DWORD PTR [eax], ecx
        mov     eax, DWORD PTR ___$ReturnUdt$[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
```

Tradiction opérateur en ASM

Code ASM

```
; TestOperatorAdd
_c$ = -12                                   ; size = 4
_b$ = -8                                    ; size = 4
_a$ = -4                                    ; size = 4
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        lea     eax, DWORD PTR _b$[ebp]
        push    eax
        lea     ecx, DWORD PTR _a$[ebp]
        push    ecx
        lea     edx, DWORD PTR _c$[ebp]
        push    edx
        call    ??H@YA?AUA@@ABU0@0@Z        ; operator+
        add     esp, 12                     ; 0000000cH
        mov     eax, DWORD PTR _c$[ebp]
        push    eax
        push    OFFSET $SG3925
        call    DWORD PTR __imp__printf
        add     esp, 8
```

Les temporaires

Problème

L'utilisation intensive fait apparaitre théoriquement beaucoup de temporaires

Code

```
struct B { B(int _val=0):Data(_val){}; int Data; };

B operator+(const B& _obj1, const B& _obj2)
{
        B temp;
        temp.Data = _obj1.Data + _obj2.Data;
        return temp;
}

void TestOperatorTemp()
{
        B a(1);
        B b(2);
        B c(3);
        B d = a + (b + c);
}
```

Question : combien de temporaires créés pour effectuer le calcul de 'd' ?

Comparaison de temporaires

Code non optimisé

```
; operator+ ??H@YA?AUB@@ABU0@0@Z
_temp$ = -4                                   ; size = 4
___$ReturnUdt$ = 8                            ; size = 4
__obj1$ = 12                                  ; size = 4
__obj2$ = 16                                  ; size = 4
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _temp$[ebp], 0
        mov     eax, DWORD PTR __obj1$[ebp]
        mov     ecx, DWORD PTR [eax]
        mov     edx, DWORD PTR __obj2$[ebp]
        add     ecx, DWORD PTR [edx]
        mov     DWORD PTR _temp$[ebp], ecx
        mov     eax, DWORD PTR ___$ReturnUdt$[ebp]
        mov     ecx, DWORD PTR _temp$[ebp]
        mov     DWORD PTR [eax], ecx
        mov     eax, DWORD PTR ___$ReturnUdt$[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
```

Comparaison de temporaires

Code non optimisé

```
; TestOperatorTemp
$T3946 = -20                                    ; size = 4
_c$ = -16                                       ; size = 4
_d$ = -12                                       ; size = 4
_b$ = -8                                        ; size = 4
_a$ = -4                                        ; size = 4
        mov     DWORD PTR _a$[ebp], 1
        mov     DWORD PTR _b$[ebp], 2
        mov     DWORD PTR _c$[ebp], 3
        lea     eax, DWORD PTR _c$[ebp]
        push    eax
        lea     ecx, DWORD PTR _b$[ebp]
        push    ecx
        lea     edx, DWORD PTR $T3946[ebp]
        push    edx
        call    ??H@YA?AUB@@ABU0@0@Z            ; operator+
        add     esp, 12                         ; 0000000cH
        push    eax
        lea     eax, DWORD PTR _a$[ebp]
        push    eax
        lea     ecx, DWORD PTR _d$[ebp]
        push    ecx
        call    ??H@YA?AUB@@ABU0@0@Z            ; operator+
        add     esp, 12                         ; 0000000cH
        mov     edx, DWORD PTR _d$[ebp]
        push    edx
        push    OFFSET $SG3927
        call    DWORD PTR __imp__printf
        add     esp, 8
```

Comparaison de temporaires

Code optimisé

```
;B::B   COMDAT ??0B@@QAE@H@Z
__val$ = 8                                          ; size = 4 ; _this$ = ecx
       mov     eax, ecx
       mov     ecx, DWORD PTR __val$[esp-4]
       mov     DWORD PTR [eax], ecx
       ret     4

; operator+    ??H@YA?AUB@@ABU0@0@Z
___$ReturnUdt$ = 8                                  ; size = 4
__obj1$ = 12                                        ; size = 4
__obj2$ = 16                                        ; size = 4
       mov     eax, DWORD PTR __obj1$[esp-4]
       mov     ecx, DWORD PTR [eax]
       mov     edx, DWORD PTR __obj2$[esp-4]
       add     ecx, DWORD PTR [edx]
       mov     eax, DWORD PTR ___$ReturnUdt$[esp-4]
       mov     DWORD PTR [eax], ecx
       ret     0
```

Comparaison de temporaires

Code optimisé

```
; TestOperatorTemp
_c$ = -16                                   ; size = 4
_d$ = -12                                   ; size = 4
_b$ = -12                                   ; size = 4
_a$ = -8                                    ; size = 4
$T3950 = -4                                 ; size = 4
        lea     eax, DWORD PTR _c$[esp+16]
        push    eax
        lea     ecx, DWORD PTR _b$[esp+20]
        push    ecx
        lea     edx, DWORD PTR $T3950[esp+24]
        push    edx
        mov     DWORD PTR _a$[esp+28], 1
        mov     DWORD PTR _b$[esp+28], 2
        mov     DWORD PTR _c$[esp+28], 3
        call    ??H@YA?AUB@@ABU0@0@Z        ; operator+
        push    eax
        lea     eax, DWORD PTR _a$[esp+32]
        push    eax
        lea     ecx, DWORD PTR _d$[esp+36]
        push    ecx
        call    ??H@YA?AUB@@ABU0@0@Z        ; operator+
```

Comparaison de temporaires

Existe-t-il un moyen pour supprimer les temporaires ?

Les principales solutions :

- Utilisation des expression-template
- Utilisation des opérateurs 'self' $(+=, -=, *=, ...)$

Opérateurs incrémentation et décrémentation

Faut-il préfèrer ++i plutôt que i++ ?

Concrètement voyons le code ASM pour les types primitifs et non-primitifs

Exemple : Incrémentation pour un type primitif

Code C++

```
int a = 1;
int c = ++a;
int d = a++
```

Opérateurs incrémentation et décrémentation

Code ASM

```
; TestIncrementOperator
_tmp$3962 = -92                          ; size = 20
_c$3934 = -72                            ; size = 20
_d$3935 = -52                            ; size = 20
_a$3933 = -32                            ; size = 20
_c$3928 = -12                            ; size = 4
_d$3929 = -8                             ; size = 4
_a$3927 = -4                             ; size = 4
        ; init
        mov     DWORD PTR _a$3927[ebp], 1
        ; c = ++a
        mov     eax, DWORD PTR _a$3927[ebp]
        add     eax, 1
        mov     DWORD PTR _a$3927[ebp], eax
        mov     ecx, DWORD PTR _a$3927[ebp]
        mov     DWORD PTR _c$3928[ebp], ecx
        ; d = a++
        mov     edx, DWORD PTR _a$3927[ebp]
        mov     DWORD PTR _d$3929[ebp], edx
        mov     eax, DWORD PTR _a$3927[ebp]
        add     eax, 1
        mov     DWORD PTR _a$3927[ebp], eax
```

Opérateurs incrémentation et décrémentation

Exemple : Incrémentation pour un type non-primitif

Code C++

```
BigObj a(0);
BigObj c = ++a;
BigObj d = a++;
```

Opérateurs incrémentation et décrémentation

Code ASM

```
; TestIncrementOperator
; BigObj : int Data0, Data1, Data2, Data3, Data4 (operator+ do nothing !)
_tmp$3962 = -92                                    ; size = 20
_c$3934 = -72                                      ; size = 20
_d$3935 = -52                                      ; size = 20
_a$3933 = -32                                      ; size = 20
_c$3928 = -12                                      ; size = 4
_d$3929 = -8                                       ; size = 4
_a$3927 = -4                                       ; size = 4
; B a(0)
        mov     DWORD PTR _a$3933[ebp], 0
; B c = ++a (10 instr.)
        mov     ecx, DWORD PTR _a$3933[ebp]
        mov     DWORD PTR _c$3934[ebp], ecx
        mov     edx, DWORD PTR _a$3933[ebp+4]
        mov     DWORD PTR _c$3934[ebp+4], edx
        mov     eax, DWORD PTR _a$3933[ebp+8]
        mov     DWORD PTR _c$3934[ebp+8], eax
        mov     ecx, DWORD PTR _a$3933[ebp+12]
        mov     DWORD PTR _c$3934[ebp+12], ecx
        mov     edx, DWORD PTR _a$3933[ebp+16]
        mov     DWORD PTR _c$3934[ebp+16], edx
; B d = a++ (20 instr.)
        mov     eax, DWORD PTR _a$3933[ebp]
        mov     DWORD PTR _tmp$3962[ebp], eax
        mov     ecx, DWORD PTR _a$3933[ebp+4]
        mov     DWORD PTR _tmp$3962[ebp+4], ecx
        mov     edx, DWORD PTR _a$3933[ebp+8]
        mov     DWORD PTR _tmp$3962[ebp+8], edx
        mov     eax, DWORD PTR _a$3933[ebp+12]
        mov     DWORD PTR _tmp$3962[ebp+12], eax
        mov     ecx, DWORD PTR _a$3933[ebp+16]
        mov     DWORD PTR _tmp$3962[ebp+16], ecx
```

Opérateurs incrémentation et décrémentation

Conclusion

- **Type primitif** : coût strictement identique et le compilateur met ce qui l'arrange
- **Type non primitif** : coût surcoût pour la post incrémentation dû à la création d'un temporaire !

$\Rightarrow$ Non négligeable pour les objets dont la construction est lourde !

Plan

Généralités

Définition

- Zone mémoire de taille réduite à faible latence
- Décomposée hiérarchiquement en plusieurs couches

Permet de faire tampon entre le processeur et la RAM

- Amortit les coûts de lecture dans la RAM
- Permet de réduire les trous dans le flux d'exécution (stall pipeline)

Problème

Si une portion de code est mal conçue : difficulté de prévoir le flot d'exécution
$\longrightarrow$ Risque de provoquer des défauts de cache : ralentissement de l'exécution

Outils

Les outils pour diagnostiquer les défauts de cache

- Sous Linux : valgrind / cachegrind
- Sous Windows : profiler de Visual Studio, ... ?

Objectifs

- Trouver les points sensibles du programme
- Permettre d'évaluer les techniques mises en place pour éviter les défauts de cache

Conseils

But : garder une certaine cohérence dans les données manipulées afin d'invalider le moins possible le cache de donnée

Conseils en vrac

- Essayer de conserver vos données alignées sur un multiple de 4 octets (pour un processeur 32bits)
- Partionner vos données **hot** et **cold** de façon à grouper en mémoire les données selon leur utilisation
- Dans les tableaux multidimensionnels tener compte de l'ordre de parcours pour organiser vos données (row/colums ou columns/rows)
- Sur code SSE utiliser les instructions de prefetch
- Réduiser la taille de vos boucles

Conseils

Attention

Ces conseils sont données pour effectuer une passe d'optimisation après identification des problèmes de cache dans une portion du code

- Pas d'application pas systématiquement sur des portions de code non problématiques
- L'optimisation prématurée est source de tous les maux !