

Les méthodes inline

Philippe Prados

pp@philippe.prados.name

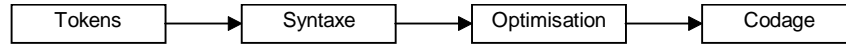


***Préservez l'environnement,
n'imprimez pas ce document***

Avant propos

Ce document décrit comment le compilateur optimise le code à l'aide des méthodes inline.

Comment le compilateur fait pour optimiser les appels d'une méthode `inline` ? Cela dépend, mais les principes généraux peuvent être décrits. Un compilateur analyse en plusieurs phases les sources. Dans un premier temps, une analyse lexicale permet d'identifier les différents éléments (*tokens*) du programme. Ensuite, une analyse syntaxique relie les éléments entre eux pour décrire les différentes étapes d'exécution. La troisième analyse simplifie l'arbre syntaxique. La dernière le traduit en code assembleur. En réalité, le compilateur exécute beaucoup plus d'étapes que celles décrites ici. Pour simplifier le discours, nous ne nous occuperons que de celles-ci.



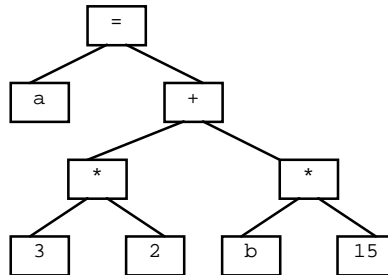
Pour éclairer ces étapes, nous allons prendre un exemple et décortiquer les premières phases du compilateur.

```
a=3*2+b*15;
```

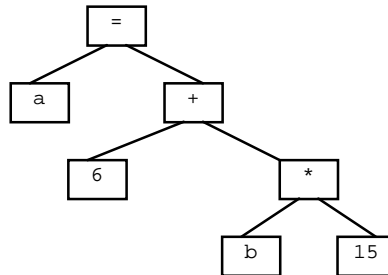
1. Identifications des *tokens*. Le compilateur détecte les *tokens* suivant :

```
'a'; '='; '3'; '*'; '2'; '+'; 'b'; '*'; '15'; ';' 
```

2. Création de l'arbre syntaxique. En faisant abstraction du point virgule, l'arbre construit est le suivant :



3. Simplification de l'arbre. Le compilateur calcule toutes les constantes. L'arbre devient :



L'expression aurait pu être écrite comme ceci :

```
a=6+b*15;
```

le compilateur serait arrivé au même résultat. Ensuite, il traduit cet arbre par le langage machine spécifique à chaque unité centrale.

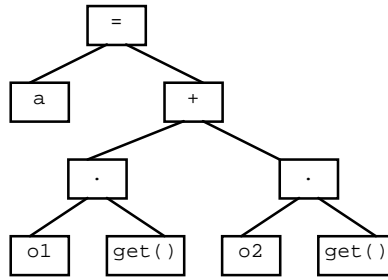
Comment cela se passe-t-il avec les méthodes `inline` ? Nous allons utiliser un autre exemple un peu plus compliqué que nous allons traduire en arbre.

```

class CObj
{ int _j;
  int _i;
  public:
    CObj(int i) : _i(i) {}
    int get() const;
};
int CObj::get() const { return _i; }

//...
{ CObj o1=4;
  CObj o2=7;
  int a;
  a=o1.get()+o2.get();// expression
}
  
```

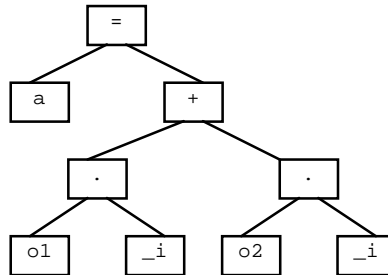
L'expression est traduite par un arbre équivalent à celui-ci :



Pour chaque appel de méthode le compilateur génère :

- la sauvegarde dans la pile de l'adresse `this`,
- l'ajout des paramètres de la méthode dans la pile,
- l'appel de la méthode
- la suppression des paramètres et de `this` de la pile.

Même si la méthode ne comporte aucun traitement, le compilateur va générer toutes les étapes indiquées ci-dessus. Pour notre exemple, le compilateur va générer deux appels à la méthode `get()`. Si cette méthode est déclarée en `inline`, l'arbre d'appel est différent.



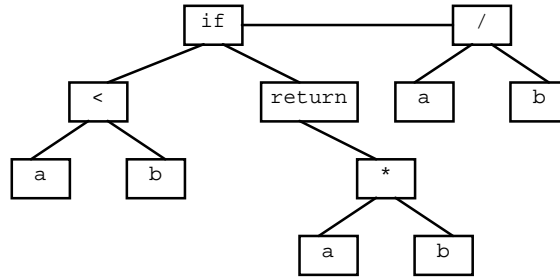
Il n'y a plus d'appel de méthode. Le compilateur utilise directement les adresses de `o1` et `o2` pour consulter leurs paramètres `_i`. L'appelant aura autant de copies de celui-ci que d'appels. Le programme sera plus long, mais le nombre d'instructions exécutées sera inférieur. Le coût de l'appel d'une fonction, quel que soit son corps est non nul. Si le corps de l'instruction a un coût inférieur à la procédure d'appel, il est préférable de déclarer la méthode en `inline`. Dans notre exemple, déclarer la méthode `get()` avec l'attribut `inline` permet d'accélérer le programme et de réduire sa taille ! Le coût d'appel d'une méthode ne peut pas être connu à priori. Ce coût est différent suivant les micro-processeurs et les compilateurs. En général, une méthode « courte » de deux ou trois lignes peut être déclarée en `inline`.

D'autre part, nous avons vu plus haut que le compilateur simplifie l'arbre syntaxique en pré-calculant les constantes. Un appel d'une méthode avec une constante en paramètre peut également augmenter les chances d'optimisation si la méthode est `inline`. En effet, le compilateur va générer chaque appel par une nouvelle version de la méthode, mais en remplaçant le paramètre constant par sa valeur. L'arbre syntaxique aura ainsi plus de chance d'être optimisé. Toute méthode est traduite en un arbre syntaxique. Cet arbre est recopié à chaque appel en y remplaçant les paramètres. Ensuite le compilateur optimise cet arbre.

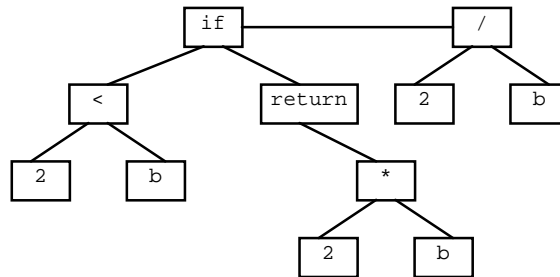
```

inline int calcul(int x,int y)
{ if (x<y) return x*y;
  return x/y;
}
//...
void f(int a,int b)
{ calcul(a,b);           // Expr 1
  calcul(2,b);          // Expr 2
  calcul(3,4);          // Expr 3
  calcul(4,2);          // Expr 4
  calcul(a,calcul(3,4)); // Expr 5
}
  
```

Pour les cinq appels de la fonction `calcul`, les arbres d'appels deviennent :



Expression 1 : `calcul(a,b);`



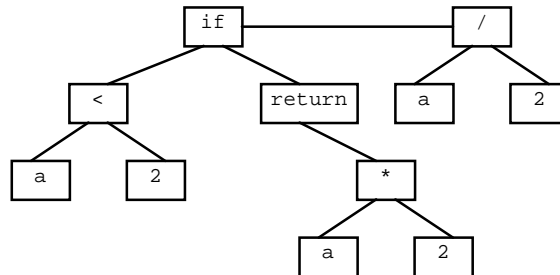
Expression 2 : `calcul(2,b);`

12

Expression 3 : `calcul(3,4);`

2

Expression 4 : `calcul(4,2);`



Expression 5 : `calcul(a,calcul(3,4));`

On constate que l'arbre peut être complètement différent suivant les paramètres reçus en entrée. Si une méthode ou une fonction reçoit une constante, et que celle-ci peut avoir un grand effet simplificateur sur l'arbre syntaxique, il est judicieux de déclarer la méthode en `inline`.

Certains compilateurs refusent de traduire une fonction en `inline` si celle-ci possède des boucles ou est trop importante pour pouvoir être gardée en mémoire par le compilateur. Dans ce cas, il fait abstraction de cet attribut, et les appels sont générés de façon classique. D'autres compilateurs, au contraire, n'imposent aucune limite à la déclaration d'une méthode `inline`. Dans ce cas de figure, une méthode très importante, appelée rarement et avec une constante particulière pour chaque appel, peut être un bon candidat à la génération `inline`. Le temps de compilation sera très important et le programme imposant en mémoire, mais l'exécution sera très efficace. Il n'est pas rare que le compilateur traduise une succession d'appels imbriqués de fonctions `inline`, par la simple lecture d'un attribut. Vous serez certainement surpris par les possibilités d'optimisations offertes par ces méthodes. Les compilateurs peuvent énormément réduire la taille de l'arbre syntaxique. Il ne faut pas se priver de rédiger des méthodes toutes petites pour accéder à un attribut car elles seront toutes diluées dans l'arbre syntaxique final.

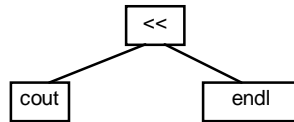
Par exemple, les flux du C++ utilisent des « manipulateurs ». Ceux-ci sont implantés à l'aide de pointeurs de fonction. Lors d'un appel de :

```
cout << endl;
```

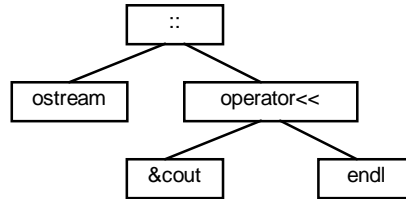
le compilateur utilise l'opérateur `<<()` recevant un pointeur de fonction. Ce pointeur est valorisé par l'adresse de `endl`. Cet opérateur appelle simplement la fonction reçue en paramètre.

```
inline ostream& ostream::operator <<(ostream& (*manip)(ostream&))
{ return ((*manip)(*this)); }
```

L'arbre syntaxique de cet appel est le suivant :



Il est ensuite modifié comme ceci :



ce qui correspond à une écriture équivalente à :

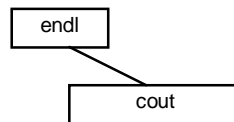
```
cout.operator <<(endl);
```

ou bien, en indiquant explicitement le paramètre this :

```
ostream::operator <<(&cout,endl); // Ce n'est pas du C++
```

&cout est le pointeur this de l'opérateur <<() et endl est le paramètre.

Comme le code de l'opérateur est inline, l'arbre devient après simplification :



L'appel de endl est effectué directement. L'opérateur <<() à disparu de l'arbre. Si l'expression avait été :

```
endl(cout);
```

L'arbre syntaxique aurait été le même. Ce n'est pas fini ! Comme le manipulateur endl est lui aussi inline, le code de celui-ci est généré *in situ* lors de la compilation de l'expression. Contrairement aux fonctions printf du C, l'utilisation des flux du C++ est *compilée* grâce à l'existence des méthodes inline. Le code le plus efficace est généré pour chaque utilisation. Si vous appelez l'opérateur ci-dessus avec un code comme ci-après :

```
void f(ostream& (*manip)(ostream&))
{
    cout << manip;      // expr
}
void main()
{
    f(endl);
}
```

la compilation de l'expression est différente. En effet, le compilateur crée une version non inline de la fonction endl(), puis fournit l'adresse de celle-ci à f(). L'expression appelle directement la fonction endl par l'intermédiaire du paramètre manip. Le code endl n'est pas généré en inline dans l'expression contrairement à l'exemple précédent. Par contre, si la fonction f() est inline, l'appel de endl est dilué entièrement dans l'appelant.

Lorsque vous rédigez votre programme, essayez de prévoir l'arbre syntaxique compris par le compilateur. Certains experts rédigent leurs programmes en fonction des possibilités d'optimisations. Par exemple, un code comme celui-ci :

```
enum Direction {Nord,Ouest,Sud,Est};
Direction const Directions[]={Nord,Ouest,Sud,Est};

// ----- Autre fichier -----
inline void recherche(Direction x)
{
    //...
}

void main()
{
    for (int i=0;i<4;++i)
    {
        recherche(Directions[i]);
    }
}
```

```

}
}

```

utilise un tableau de constantes pour manipuler l'ensemble des quatre points cardinaux. La boucle `i` appelle la fonction `recherche` pour chacune des constantes. Une seule génération de cette fonction est effectuée dans le corps de la boucle. Cette traduction utilise une variable pour le paramètre de recherche. Le compilateur ne peut pas pré-calculer les expressions de la fonction `recherche` lors de sa génération `inline`. Si le programme est rédigé différemment :

```

enum Direction {Nord,Ouest,Sud,Est};
Direction const Directions[]={Nord,Ouest,Sud,Est};

// ----- Autre fichier -----
inline void recherche(Direction x)
{
    //...
}

void main()
{
    recherche(Nord);
    recherche(Ouest);
    recherche(Sud);
    recherche(Est);
}

```

le compilateur va pouvoir spécialiser la fonction `recherche` pour chacune des constantes d'orientation. Le choix de la rédaction de `main()` est ici dicté par les possibilités d'optimisation du compilateur. Le corps de la fonction `main()` sera beaucoup plus gros, car quatre générations de `recherche()` seront effectuées. Par contre, l'exécutable sera beaucoup plus rapide. Vous pouvez également modifier les signatures de vos méthodes pour augmenter les chances d'utilisation de constantes. Par exemple, un programme de jeux voulant parcourir un damier suivant les directions horizontales et verticales peut utiliser une méthode recevant dans les paramètres le coefficient à ajouter à la coordonnée X, et le coefficient pour la coordonnée Y au lieu de recevoir la direction voulue.

```

const int MaxX=10;
const int MaxY=10;
struct
{ int offsetx;
  int offsety;
} Offset[]={1,0},{0,1};

enum TDirection {Horizontal, Vertical};

void calcul(TDirection dir)
{ int offx=Offset[dir].offsetx;
  int offy=Offset[dir].offsety;
  for (int x=0;x<MaxX;x+=offx)
  { for (int y=0;y<MaxY;y+=offy)
    { //...
    }
  }
}

void main()
{ calcul(Horizontal);
  calcul(Vertical);
}

```

peut être modifié comme cela :

```

const int MaxX=10;
const int MaxY=10;
inline void calcul(int offx,int offy)
{ for (int x=0;x<MaxX;x+=offx)
  { for (int y=0;y<MaxY;y+=offy)
    { //...
    }
  }
}

void main()
{ calcul(1,0);
  calcul(0,1);
}

```

Ce code est moins élégant que le précédent, mais le compilateur pourra l'optimiser au mieux. La signature de `calcul()` est modifiée pour pouvoir bénéficier d'une meilleure optimisation.

Pour corriger votre programme, il ne faut surtout pas que les méthodes appelées soient diluées dans l'arbre d'appel. Vous ne retrouveriez plus vos petits... C'est pour cela que tous les compilateurs C++ offrent une option permettant de rejeter tous les attributs `inline`. Le compilateur se comportera comme si ces attributs n'avaient jamais existé. Cela vous permet de suivre pas à pas votre programme. Ensuite lors de la phase d'intégration, en changeant les paramètres de compilation, vous pourrez générer une version nettement plus rapide et souvent moins exigeante en taille mémoire !

Voici quelques principes vous permettant d'imaginer le comportement du compilateur lors de la phase d'optimisation :

- la manipulation d'une variable globale est plus rapide que la manipulation d'une variable locale ou d'un paramètre,
- la manipulation d'une constante est plus rapide que la manipulation d'une variable,
- le compilateur effectue tous les calculs sur les constantes,
- un appel de fonction ou de méthode est moins rapide que l'exécution directe de celle-ci dans l'appelant,
- un appel de méthode virtuelle est moins rapide qu'un appel de fonction classique.