Techniques for Scientific C++

Todd Veldhuizen

2015 年 11 月 30 日

1 Introduction

1.1 C++ Compilers

1.1.1 Placating sub-standard compilers

全てのコンパイラは、ISO 標準規格/ANSI 標準規格の異なる部分をサポートしている。多くのプラットフォーム上で C++ のライブラリやプログラムを保証するのは頭の痛い問題である。

２つの最も対応の難しい分野は、テンプレート (template) と標準ライブラリ (STL) である。

```
#include <cmath>

namespace blitz {
    double pow(double x, double y)
    {
        return BZ_MATHFN_SCOPE(pow)(x,y);
    }
};
```

このコードでは、blitz の名前空間で pow 関数を作成している。

以下のような複数プラットフォームへの対応が必要な場合を考える。

- あるコンパイラには <cmath> ヘッダがないので、<math.h> に置き換えなければならない。
- あるコンパイラは、<cmath> ヘッダを持っているが、std 名前空間に C math 内の関数を持っていないかもしれない。
- 名前空間自体をサポートしないコンパイラもある。

以上のような状況での解決策の一案のコードが以下である。

```
#ifdef BZ_MATH_FN_IN_NAMESPACE_STD
    #include <cmath>
#else
    #include <math.h>
#endif

#ifdef BZ_NAMESPACES
double pow(double x, double y)
{
    return BZ_MATHFN_SCOPE(pow)(x,y);
}
#endif
```

このように対応するのは好まれない。Blitz++ を用いることで対処が少し楽になる。

複数プラットフォーム対応の問題を避けるほかの方法としては、gcc コンパイラをサポートするようにコー

ドを書くことである。gcc コンパイラは標準準拠なほとんどのコンパイラよりも良く、急速に STL の実装が改善されている。最適化能力は KAI C++ やベンダー独自のコンパイラよりも劣る。しかし、gcc コンパイラは、out of cache（おそらく CPU 内部のキャッシュメモリではなく、外部のメモリにアクセスしにいくということ [1]）という大きな問題に対しては、とてもよく対処する。

1.1.2 The compiler landscape

多くの C++ コンパイラは Edison Design Group (EDG) の作成したフロントエンドを使用している。このフロントエンドは、とても頑強で信頼に足る標準規格 (the standard) と一致しており、よいエラーメッセージを発行する。おそらくどんなプラットフォームでも EDG ベースのコンパイラが存在する。SGI C++, KAI C++, DEC cxx, Cray C++. Intel C++, Portland Group C++, Tera C++ といったコンパイラが EDG ベースのものである。

しかし、主流の C++ コンパイラ（Borland/Inprise、Microsoft 等）は驚くべきほどに C++ 標準規格よりも遅れている。（顧客が C++ 標準規格に興味ないため）

もし、標準準拠に近いコンパイラを使用したいなら、Intel C++（Visual C++ へのプラグインとして使用可能）か、Metrowerk の CodeWarrior がよい選択肢となる。

1.1.3 C++-specific optimization

ほとんどのコンパイラが C++ の最適化の点で悲しいほどに貧弱である。伝統的な最適化のテクニックは、nested aggregates や一時オブジェクトを用いるように修正される必要がある。例外は、KAI C++ である。KAI C++ は一流の最適化能力を持つ C++ コンパイラである。ほかの例外は、SGI C++ コンパイラである。ある程度よい仕事する。一時オブジェクトの除去の点といった事柄のために KAI C++ よりもあまり良くないと現在はされる）

KAI は最適化技術についての米国特許を持っている。同様の最適化を実装することについてコンパイラベンダを失望させた。しかし、その最適化エンジンのライセンス使用許可を少なくとも 1 つのほかのコンパイラベンダーに与えている。

1.2 Compile times

Blitz++ やほかのテンプレートに注力する C++ ライブラリに関する最もよく受ける批判の一つは、コンパイル時間である。続く節でなぜ C++ プログラムをコンパイルするのにそんなに長い時間がかかるのかを説明していく。

1.2.1 Headers

ヘッダーは問題の大きな部分を占めている。C++ はあまり効率的なモジュールシステムを持っていない。結果として、各コンパイルユニットに対して、膨大な量のヘッダーをパースして、分析しなければならない。例えば、<iostream> をインクルードする際には、コンパイラによっては何万行ものコードを include しなければならない。

コンパイル時には全てのテンプレートの実装を利用できる必要があるとの理由から、テンプレートライブラリは特にコンパイルに時間がかかる。そのため、関数宣言の代わりにヘッダーに関数の定義を書く。

例えば、Blitz++ ライブラリから examples/array.cpp を例として gcc を使って筆者の環境でコンパイルし

たところ 37 秒かかった。もし、このファイル外の全てのコードを取り除き、blitz/array.h だけ include するようにしてら、コンパイルに 22 秒かかった。言い換えると、コンパイル時間の 60% がヘッダーの処理に費やされているのである！

プリコンパイルされたヘッダは助けになりうる場合もある。しかし、問題をより悪くする場合もある。プリコンパイルされたヘッダーを Blitz++ で変換すると、置き換えたヘッダーよりも読み込みに時間のかかる 64Mb ものダンプが生成される。

このことからの学びは、可能な限りヘッダーを小さくするということである。ライブラリ開発の際には、ライブラリ全体をインクルードする一つのヘッダーではなく、別々にインクルードされなければならない、多くの小さなヘッダーを提供することである。

1.2.2 Prelinking

テンプレートはコンパイラに対して深刻な問題を引き起こす。いつそれらはインスタンス化されるべきなのかという問題である。2 つの一般的なアプローチがある。

■使用する全てのインスタンスを生成 各コンパイルユニットに対して、使われる全てのテンプレートインスタンスを生成する方法である。この方法は、テンプレートインスタンスの重複を生む。例えばもし、foo1.cpp と foo2.cpp の両方が list¡int¿を使用している場合、foo1.o ともう一つは foo2.o の内部で list¡int¿は 2 回インスタンス化される。リンカによっては自動的に重複しているインスタンスを処分する。

■Prelinking どれもインスタンス化しない。その代わり、リンク時にどのテンプレートインスタンスが必要になるか決まるまで待って、戻って、それらを生成する。これを prelinking という。

ちなみに、prelinking は複数回プログラム全体をコンパイルし直す必要がある。

モジュールが 3 回コンパイルし直されるのが典型的である。想像できるように、とても大きなプログラムだと、prelinking はコンパイル時間を長くしてしまう。

EDG フロントエンドを使用するコンパイラは-tused や-ptused というオプションを使用することで prelinking をさせないようにすることができる。

1.2.3 The program database approach - Visual Age C++

IBM の Visual Age C++ コンパイラは、分割コンパイルの問題に対して、興味深いアプローチを採用している。全てのコードを大きなデータベースにキープしておくのである。プログラムの完全なソースコードがコンパイル時に使用可能なため（一度に一つの module の変わりに）、VAC++ は prelinking をする必要や分割コンパイルの問題を心配する必要がない。また、変更のあった関数のみコンパイルし直され、変更された定義に依存する。このアプローチは、テンプレートに注力する C++ ライブラリにとってコンパイルにかかる時間を劇的に減らす可能性を持っている。

1.2.4 Quadratic（2 次の） / Cubic（3 次の） template algorithms

Blitz++ のようなライブラリは、テンプレートを拡張的に使用している。あまりスケールしないテンプレートの実装におけるアルゴリズムを採用しているコンパイラもある。

例えば以下のコードについて考えてみよう。

```
1 template<int N>
2 void foo()
3 {
```

```
4       foo<N−1>();
5   }
6
7   template<>
8   void foo<0>()
9   {
10  }
```

もし、foo<N> をインスタンス化すると、foo<N-1> もインスタンス化され、続いて foo<N-2> もというように foo<0> まで続く。これはどのくらい時間がかかるだろうか？インスタンス化された関数の数は N だから、コンパイル時間は N の線形時間になるはずだろうか？

そうではないコンパイラも存在する。単純にいえば、多くのコンパイラはテンプレートのインスタンスのリンクされたリストをただ保持している。template がインスタンス化されるか否かを決めるのに、このリストに対して線形探索が行われる。コンパイル時間は $O(N^2)$。これは各テンプレートに対してインスタンスが多くないときには問題にならないが、Blitz++ のようなライブラリはいつもテンプレートクラスのインスタンスを何千も生成しているようなときには問題となる。

同様な問題がエイリアス解析[*1]といった最適化のアルゴリズムに対しても発生している。C++ の主流コンパイラで $O(N^2)$ や $O(N^3)$ のアルゴリズムがあることを確認した。

1.3 Static Polymorphism（静的ポリモーフィズム）

1.3.1 Are virtual functions evil?

C++ の世界では、ポリモーフィズムは通常仮想関数のことを指す。この種のポリモーフィズムは、便利なデザインツールである。しかし、仮想関数は大きなパフォーマンス上のペナルティを抱えうる。

- 追加のメモリアクセス
- 仮想関数の効率の良さは、予期せぬブランチを発生させ、パイプラインを急停止させうる。（しかし注目すべきは、この問題を避けるためのブランチキャッシュを持つアーキテクチャが存在することである。）[*2][*3]
- 現在の C++ コンパイラは（筆者の知っている限りでは）仮想関数呼び出しに関する最適化ができない。仮想関数は、命令スケジューリング[*4]、データフロー解析[*5]、ループ展開などを阻害する。

ループ展開の例

```
1   for (int i = 0; i < 1000; ++i) {
2       ++test;
3   }
```

```
1   for (int i = 0; i < 1000/5; ++i) {
2       ++test;
3       ++test;
```

[*1] エイリアス解析（英: Alias analysis）：コンパイラ理論における手法の一つで、ある記憶域が複数の箇所からアクセスされるかどうかを判定する方法である。二つのポインタが同じ記憶域を参照している場合、「エイリアスしている」とみなされる。

[*2] dispatch：タスク指名（コンピュータが何か仕事や処理単位を行う場合に、その仕事等に処理するための時間を割り当てる作業）

[*3] pipeline：CPU 動作の文脈では、高速化を目的とする命令サイクルの先回り制御方式を指す。

[*4] 命令スケジューリング（めいれいスケジューリング、英: Instruction scheduling）：計算機科学のおけるコンパイラ最適化方法の一つで、命令レベルの並列性を高め、命令パイプラインを備えた計算機の性能を向上させる。

[*5] データフロー解析（英: Data-flow analysis）：プログラム内の様々な位置で、取りうる値の集合に関する情報を収集する技法である。制御フローグラフ (CFG) を使って変数の値が伝播するかどうかなどの情報を集め、利用する。このようにして集められた情報はコンパイラが最適化に利用する。

```
4       ++test;
5       ++test;
6       ++test;
7   }
```

仮想関数の呼び出しを直接的なタスク指名 (dispatch) に置き換える研究はあるが、この研究はまだどのコンパイラにも実装されていない。その原因の 1 つは、仮想関数のタスク指名を取り除くのには whole program analysis（プログラム全体の解析）が必要となることである。コンパイル時の解析に使用可能な全てのプログラムを保持している必要がある。ほぼ全ての C++ 実装は、各モジュール（compilation unit）を別々にコンパイルしている。そう処理しない唯一の例外は Visual Age C++ である。

■仮想関数を使うべきでない場面

- 仮想関数内のコードが少ないとき（例えば、25flops[*6]よりも少ないとき）。
- 仮想関数を頻繁に使用するとき（例えばループ内で使用するとき）。これはプロファイラ[*7]の助けを用いて決めることができる。

呼び出される関数が大きいか、もしくはあまり頻繁に呼ばれない場合には、仮想関数の使用は強く推奨される。

■仮想関数の問題の例　しかし、科学計算のコードにおいて、仮想関数が有用な場面は、ループ内で小さなルーチンを含んでいることが多い。古典的なクラスの例は以下のようなものである。

```
1   class Matrix {
2   public:
3       virtual double operator()(int i, int j) = 0;
4   };
5
6   class SymmetricMatrix : public Matrix {
7   public:
8       virtual double operator()(int i, int j);
9   };
10
11  class UpperTriangularMatrix : public Matrix {
12  public:
13      virtual double operator()(int i, int j);
14  };
```

行列から要素を読み取るという、仮想関数の”operator()”へのタスク指名 (dispatch) は、あらゆる行列のアルゴリズムのパフォーマンスを損なう。

この解決策として、ここでは 2 つの方法を提案する。

1.3.2 Solution A: simple engines

アイデアは、動的ポリモーフィズム（仮想関数）を静的ポリモーフィズム（テンプレートパラメータ）で置き換えるということである。このテクニックを用いたコードが以下である。

[*6] FLOPS：コンピュータの処理速度をあらわす単位の 1 つで、1 秒間に実行できる浮動小数点数演算の回数。科学技術計算等における性能指標として用いられる事が多い

[*7] プロファイラ：動作中のプログラムがどの処理をどういった順序で実行したかを監視するプログラム。コンパイラやデバッガなどと共に、プログラミング言語の開発環境の一部として提供されることが多い。例えば、プログラムはまず A という関数を呼び出し、次いで B の計算を実行し…というように、逐一記録が残される。それぞれの処理にかかった時間などを監視できるものもある。ユーザはこうした記録を分析することで、プログラムが自分の意図通りに動作しているか、またプログラムのどの部分がボトルネックとなって処理に時間がかかっているのか、などを知ることができる。プログラムの障害を見つけるために用いられるよりも、プログラムの余計な部分を削るなどして高速化するために用いられることが多い。

```
class Symmetric {
    // Encapsulates storage info for symmetric matrices
};

class UpperTriangular {
    // Storage format info for upper tri matrices
};

template<class T_engine>
class Matrix {
private:
    T_engine engine;
};

// Example routine which takes any matrix structure
template<class T_engine>
double sum(Matrix<T_engine>& A);

// Example use ...
Matrix<Symmetric> A;
sum(A);
```

このアプローチでは、行列構成の変数の側面が engines Symmetric と UpperTriangular 内に隠されている。Matrix クラスは、engines をテンプレートパラメータとして受け取り、その engines をインスタンスに含めている。

SymmetricMatrix ではなくて、Matrix<Symmetric> というように、これまでユーザが慣れていた表記法とは異なる。また、Matrix では、多くの興味深い操作が engines に移譲されている。

最も大きな頭痛の種は、すべての Matrix のサブタイプが同じメンバ関数を持たねばならないことである。例えば、Matrix<Symmetric> に isSymmetricPositiveDefinite() メソッドを持たせようとすると、典型的な実装は以下のようになる。

```
template<class T_engine>
class Matrix {
public:
    // Delegate to the engine
    bool isSymmetricPositiveDefinite()
    {
        return engine.isSymmetricPositiveDefinite();
    }
private:
    T_engine engine;
};
class Symmetric {
public:
    bool isSymmetricPositiveDefinite()
    {
        ...
    }
};
```

しかし、このコードで Matrix<UpperTriangular> が isSymmetricPositiveDefinite() メソッドを持っていることに意味はない。上三角行列は対称になりえないためである。このアプローチでわかるのは、基底となるタイプ（この例では Matrix）は、その派生タイプに提供される全てのメソッドの集まりを保持し続ける必要があるということである。そのため、多くの例外を投げるだけのメソッドを各エンジンに保持し続けることになる。

```
class UpperTriangular {
public:
    bool isSymmetricPositiveDefinite()
    {
        throw makeError("Method isSymmetricPositiveDefinite() is "
        "not defined for UpperTriangular matrices");
```

```
7           return false;
8       }
9   };
```

別のアプローチ（推奨はされない）：これらのメソッドを含めないことで、ただユーザを見捨てるという方法。そしてユーザは、含まれなかったメソッドを使おうとする際に出てくる不可思議なテンプレートのインスタンス化エラーに対処しなければならない。

よりよい解決策が以下の B である。

1.3.3 Solution B: the Barton and Nackman Trick

このトリックは、しばしば"Barton and Nackman Trick"と呼ばれる。なぜなら彼らの本で使用しているから。Geoff Furnish は"Curiously Recursive Template Pattern"というそれをよく表す単語を生み出した。

```
1   // Base class takes a template parameter. This parameter
2   // is the type of the class which derives from it.
3   template<class T_leaftype>
4   class Matrix {
5   public:
6       T_leaftype& asLeaf()
7       {
8           return static_cast<T_leaftype&>(*this);
9       }
10
11      double operator()(int i, int j)
12      {
13          return asLeaf()(i,j);
14      } // delegate to leaf
15  };
16
17  class SymmetricMatrix : public Matrix<SymmetricMatrix> {
18  ...
19  };
20
21  class UpperTriMatrix : public Matrix<UpperTriMatrix> {
22  ...
23  };
24
25  // Example routine which takes any matrix structure
26  template<class T_leaftype>
27  double sum(Matrix<T_leaftype>& A);
28
29  // Example usage
30  SymmetricMatrix A;
31  sum(A);
```

この方法は、より標準的な継承階層のアプローチを可能にする。トリックは、基底クラスがテンプレートパラメータを受け取り、そのテンプレートパラメータは派生クラスの型となっている。このことは、オブジェクトの完全な型がコンパイル時に知られていることを保証する。仮想関数のタスク指名を必要としないのである。

メソッドは、派生クラス内で選択的に特殊化される。（すなわち、デフォルトの実装は基底クラス内にあるが、必要なときには派生クラスでオーバーライドされる。）前項のアプローチとは違って、派生クラスは派生クラスに特有のメソッドを保持しうる。

このアプローチでは、基底クラスは派生クラスに全てを委譲しなければならない。1 段よりも深い継承ツリーの作成には考えることが必要だが、可能である。

参考文献

[1] プログラミング :: 高速なプログラムを書く為に :: メモリ、http://myoga.web.fc2.com/prog/cpp/opti02.htm