ソフトウェア工学 96−19
（1994. 1. 21）

C++プログラム・データベース構築

安田 和† 三ツ井 欽一† 中村 宏明† Shahram Javey‡
†日本アイ・ビー・エム（株） 東京基礎研究所
‡IBM Canada

Abstract

ブラウザーなどのプログラミング・ツールの多くは対象のプログラムに関する情報を必要とするが、これは通常ソース・コードから獲得され、ツールから利用しやすい形で保管される。我々はC++ブラウザーを開発した際、ソース・コードから獲得したプログラム情報をプログラム・データベースに渡すためのインターフェースとして、pdb(program database) stream interfaceを考案した。pdb streamはプログラム情報を小さな意味単位の列に変換したものであり、その入出力はオブジェクト間のメッセージ通信を使って行なわれる。pdb stream interfaceは、情報の供給側にとって負担が少なく、拡張性にすぐれている。本論文では、このpdb stream interfaceをその使用経験とともに解説する。

Constructing a C++ Program Database

K. Yasuda†, K. Mitsui†, H. Nakamura†, and S. Javey‡
†IBM Research, Tokyo Research Laboratory
1623-14, Shimo-tsuruma, Yamato-shi, Kanagawa-ken, 242, Japan
‡IBM Canada

Abstract

Most programming tools such as program browsers require information about the target program, which is usually retrieved from the source codes and stored in the form convenient for the tools. When we developed a C++ browser, we designed an interface called *pdb stream* to transfer the program information from the information retriever to the program database. The pdb stream is a sequence of messages, each of which conveys a piece of information about the program. This paper describes the concepts of the pdb stream interface and the experiences using it.

1 はじめに

C++におけるプログラム開発を支援するためのツールには、プログラム・ブラウザー、デバッガー、各種のスタティック・アナライザーやダイナミック・アナライザーなどさまざまなものがあるが、いずれのツールにも共通するのは、対象となるプログラムについての知識を各ツールが知っている必要がある点である。この知識には、プログラムに現れるシンボルやその型についての情報、クロス・リファレンス情報、ソース・ファイルやモジュール間の関係に関する情報などが含まれる。こういった情報を管理するためのプログラム・データベースを構築するには、プログラムのソース・コードを解析することによって情報を獲得し、それをツールから利用できるような適切な表現に変換・格納する必要がある。本論文は、われわれが C++ ブラウザーを開発する際に採用した C++ プログラム・データベース構築方法を、とくにその情報供給源とのインターフェースを中心に解説するものである。

プログラムに関する知識を、プログラムのソース・コードから獲得するにはいくつかの方法があげられる。ひとつは、Sniff[1] の fuzzy parser にみられるように、そのツール専用に作られた簡易版のパーザーを利用することである。こういったパーザーは、実行が高速ではあるが、厳密なスコープ管理やマクロの処理などは行なうことができないため、誤った情報を出力する危険がある。

もうひとつの方法は、正確な構文解析を行なうパーザーを作り、各種ツールから利用できる汎用の形式で情報を出力させることである。CIA++[2] では、ciafront とよばれる cfront に手を加えたパーザーによって、プログラムに現れる何種類かの関係を表にしてテキスト・ファイルに書き込んだものを、各種のツールが利用する形をとっている。Alf[3] は、C++プログラムのプログラムの意味に注目して、パーザーが取り出した結果を、オブジェクトのツリーの形で表現したものである。こういった専用のパーザーは、かなりの正確さをもちあわせてはいるが、コンパイラと別のツールとして維持・管理するのは負担が大きい。

われわれは、プログラム情報の正確さを重視するとともに、C++ ブラウザーの開発が C++ コンパイラの開発と並行した事情も手伝って、コンパイラ自身がプログラム情報を出力する方法を採用した。

一方、対象となるデータベースは、

- C++ プログラムにあらわれる複雑な関係を自然に表現できること、
- そういった関係をたどるような、さまざま問い合わせを高速に行なえること、

などの条件を満たす必要があり、C++で開発を行なうこともあってオブジェクト指向データベースを仮定し、Alf 同様プログラムを C++ オブジェクトのツリー形式で表現することにした。ただし、今回のブラウザーでは persisent なオブジェクト指向データベースを使用できる条件が整わなかったため、いったんコンパイラがファイルの形でデータを出力し、それを C++ ブラウザーが読み込んでヒープ上にデータベース・オブジェクトを生成する方式をとることとした。その上で、コンパイラが出力する情報の形式を設計する際に、次のような点を考慮した。

コンパイラ側の負担　情報を出力することによるコンパイラのオーバーヘッドを最小にすること、例えばコンパイル時に保持する情報量をなるべく少なくすることが求められた。その上、言語仕様やコンパイラに変更があったときに容易に対応できることが必要である。そのためには、情報の単位をできるだけ細かくすることや、コンパイラがソース・プログラムを処理する片端から、それまでにわかった情報を出力できるようにすることが、大きな要件となった。

拡張性　コンパイラによって出力される情報は、とりあえずは C++ ブラウザーが単独で利用するものであるが、将来的にはデバッガーやスタティック・アナライザなど各種のツールによっても利用されることが見込まれた。したがって、出力データも大枠では他のツールと共有できるものであり、かつ必要な情報が増えた場合なども拡張が容易な形式でなければならない。また、当座はファイルを情報媒体として使うことになったとはいえ、将来 persistent なオブジェクト指向データベースを利用することになった場合も、できるだけ

プログラムの変更が必要ないように設計することが求められた。

効率　一般にプログラム・データベースではデータが巨大になりがちである。今回は、情報の媒体としてファイルを使用するため、その大きさをできるだけ小さくする必要があった。また、そのファイルを読み込んで、ヒープ上に C++ オブジェクト・データベースを構築するときの実行効率も問題となった。

以上の点を考慮して、我々は pdb stream と呼ばれるストリーム形式を情報の媒体とし、それをオブジェクトへのメッセージ通信によって受け渡しする方式を考案した。pdb stream は、プログラム上の情報を小さな意味単位に分解し、それを pdb instruction と呼ばれる命令の列として、ソース・プログラム上での出現順に従って出力するものである。

本論文は、2節でわれわれの使ったデータベースのスキーマについて簡単に解説したあと、3節でこの pdb stream の概要を、4節ではこの pdb stream からオブジェクト・データベースを構築する方法、およびそのとき問題になった点や利点について説明し、最後に今後の課題を述べてまとめとする。

2　Database Schema の概略

われわれの C++ ブラウザーのプログラム・データベースでは、プログラムに現れる基本概念（ファイル・文・シンボルなど）が C++ オブジェクトによって表現される。そのオブジェクトがポインタによって結ばれ、ネットワークを形作ることによって、プログラム上の関係が表現されている。本節ではデータベースのスキーマを解説するが、ページ数の関係上、詳細には触れない。

C++ プログラミングにおいては、クラスのインターフェースを記述したヘッダー・ファイルは、多数のインプリメンテーション・ファイルからインクルードされるのが普通であり、かつこれらのインプリメンテーション・ファイルは個別にコンパイルされる。したがって、コンパイラが情報を出力するのもコンパイル単位ごとになる。このとき、複数のコンパイル単位からなるプログラムについて 1 個のデータベースを構築する場合には、同一のヘッダー・ファイルから来た情報を同定し、情報の重複をなくすことが問題となる。つまり、C++ プログラム・データベースを設計する上では、情報の共有の単位とその同一性の判定基準を決めることが必要となる。

ここで注意すべきなのは、C++においてコンパイルやインクルードの単位は確かにファイルであるが、C++の意味論上では、ファイルの区切りはただの空白文字にすぎないことである。例えば、一つのクラス定義、あるいは一つの式さえ、いくつものファイルにまたがって書かれてもかまわない。また同一のファイルでも、たとえばあるマクロ名が定義されているかどうかなど、文脈によって意味が変わることがある。

われわれのデーターベース設計にあたっては、C++ におけるプログラムの意味を正確に表現するため、情報共有の単位として「トップレベル構造」を採用した。トップレベル構造は、ファイル・スコープにある C++ ステートメントおよびすべてのプリプロセッサ命令を表す概念である。

トップレベル構造は、そのソース・ファイル上での開始位置が同一で、その中で参照している外部シンボル名がすべて一致するとき同一であるとされる。

個々のトップレベル構造は、その種類に応じたクラス TopLevelConstruct の派生クラスのオブジェクトとして表現される。TopLevelConstruct の他、データベースを構成するオブジェクトのクラスには以下のような種類がある。

CompilationUnit　CompilationUnit オブジェクトは、一つのコンパイル単位に関する情報を表すオブジェクトのツリー構造のルートとなる。

File　ソース・ファイルを表す。インプリメンテーション・ファイルおよびヘッダー・ファイルの両方を含む。パス名とタイムスタンプが互いに一致するファイルは同一とみなされる。ファイル間のインクルードの関係は、#include 命令を表す TopLevelConstruct オブジェクトによって表現される。

Statement　通常の C++ ステートメントを表す。ステートメントの種類は Statement クラスの派生クラス

によって表現される。

Xref あるシンボルのソース・ファイル上の1点における参照関係を表現している。シンボルの参照のされ方、つまりシンボルが定義・宣言・変更されているといった場合の違いを、Xrefクラスの派生クラスを使って表す。

Symbol プログラム中に現れるシンボルを表す。シンボルには、クラス等の型名、関数・変数・マクロ名・ラベル名などがあるが、これらはその種類に応じてSymbolクラスの派生クラスのオブジェクトとして表現される。外部リンクを持つ同じ名前のシンボルは、同一のトップレベル構造で定義され、かつ、クラスのメンバーである場合は親クラスが同一であるとき、さらに、型があるシンボルの場合は型が同一であるときに同一であると判定され、複数コンパイル単位間で同一のオブジェクトが共有される。

Type 変数や関数などのシンボルの型に関する情報を表す。後に効率上の問題が生じたため、このクラスは廃止されたが、その経緯については、4.2節でふれる。

図2は、次の3個のファイルからなるデータベース中のオブジェクトの関係を簡単にあらわしたものである。

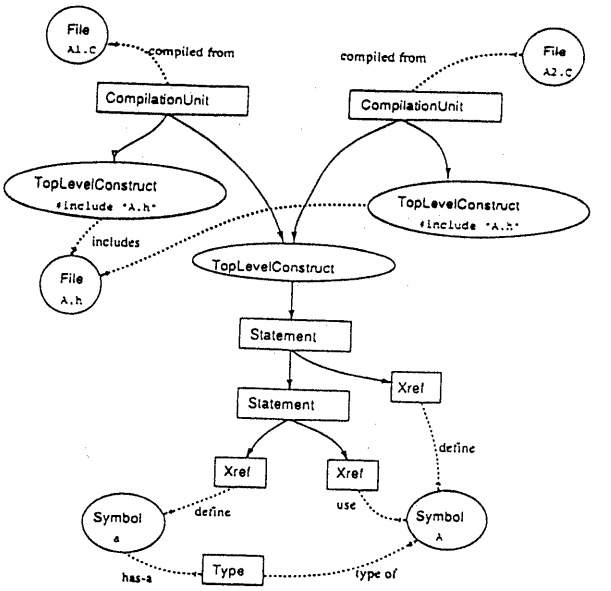
```
// A1.C
#include "A.h"

// A2.C
#include "A.h"

// A.h
class A { A* a; };
```

図の実線は、CompilationUnitオブジェクトをルートとするプログラムのツリー構造を示すものであり、点線はその他の諸関係を表す。これらの関係はすべて双方向ポインターによって実現されている。

図2データベース中のオブジェクトの関係



3 PDB Stream

pdb streamは、C++のプログラム上の情報を、小さな意味単位に分解し、そのそれぞれをpdb instructionと呼ばれる命令に対応づけ、ソース・プログラム上での出現順に並べたストリームとして、ツール間の通信に用いるインターフェースである。

pdb instructionは、おおよそ次の種類に分類される。

Symbol instruction プログラムに新しくシンボルが導入されたとき、シンボルの名前とその位置とを与える。シンボルの種類によって、SYM_CLASS, SYM_FUNCTION などがある。一度symbol instructionによって導入されたシンボルには、コンパイル単位内での通し番号(SymId)が割り当てられ、以後はそのSymIdによって参照される。

Attribute instruction 直前のsymbol instructionによって導入されたシンボルの属性を与える。クラス・

メンバーのアクセス属性やstatic属性などが含まれる。

Type instruction 直前の symbol instruction によって導入されたシンボルの型情報を与える。type instruction には TYPE_INT など基本的な型を表すものと、TYPE_POINTER のようにある型から別の型を構成するものなどがある。例えば A がクラス名であるとき、`const A* v;` で定義された変数の型は、

```
TYPE_POINTER
TYPE_CONST
TYPE_NAMED A's SymId
```

のように 3 つの type instruction からなるストリームによって表される。

Statement instruction ステートメントの種類、開始位置および終了位置を与える。ステートメントの開始位置は BGN_STATEMENT によって示され、終了位置は種類によって END_WHILE, END_IF などによって示される。ステートメントの種類がそのステートメントの終了までわからないのは、C++のコンパイラの制約によっている。

Xref instruction シンボルが参照されたときに、その位置と SymId とを与える。シンボルの参照のされかたによって、XREF_DEFINE, XREF_LVALUE などの種類がある。

CPP instruction プリプロセッサ命令の種類、開始位置および終了位置を与える。#include では、インクルードされるファイルについての情報も与える。

以下のプログラムに対する pdb stream を例として図 3に示す。

```
class A;
class B {
  public:
    A* b();
};
A* B::b() {
}
```

図 3 pdb stream の例

```
BGN_STATEMENT:  coord(1,1)
  SYM_CLASS: "A" (id=1) coord(1,7)
END_DECL:  coord(1,8)

BGN_STATEMENT:  coord(2,1)
  SYM_CLASS: "B" (id=2) coord(2,7)
  BGN_MEMBER_LIST:  coord(2,9)
    BGN_STATEMENT:  coord(4,2)
    | XREF_REF: sym=A (id=1) coord(4,2)
    | SYM_FUNCTION: "b" (id=3) coord(4,5)
    | TYPE_FUNCTION
    |   TYPE_POINTER
    |   TYPE_NAMED: Type="A" (id=1)
    |   TYPE_VOID
    | TYPE_FUNCTION_END
    | ATTR_MEMBER_FUNC: public
    END_MEMBER_DECL:  coord(4,11)
  END_DEF:  coord(5,1)
END_DECL:  coord(5,2)

BGN_STATEMENT:  coord(6,1)
  XREF_REF: sym=B (id=2) coord(6,4)
  XREF_REF: sym=A (id=1) coord(6,1)
  XREF_DECL: sym=b (id=3) coord(6,7)
END_ANSI_DECLARATOR:  coord(6,9)
BGN_FUNCTION_DEF:  coord(6,11)
  BGN_STATEMENT:  coord(7,2)
  END_RETURN: coord(7,13)
END_DEF:  coord(8,1)
```

```
  return 0;
}
```

4 Implementation

4.1 PdbStream クラス

pdb stream は、オブジェクト間のメッセージ通信を利用して受け渡しされる。これは C++における virtual メソッド・コールによって実現されている。個々の pdb instruction は、抽象クラス PdbStream の virtual メソッドに一対一で対応している。pdb stream を出力する側では、PdbStream のメソッドを呼び出すことによって、対応する pdb instruction が一つ出力されたことになる。一方、pdb stream を受けとる側はすべて、抽象クラス PdbStream の派生クラスとして実現される。

例えば、TYPE_INT instruction を、PdbStream の派生クラスのオブジェクト receiver に渡すには、PdbStream のメソッド typeInt() を

```
receiver.typeInt()
```

のように呼んでやればよい。

われわれの C++ ブラウザーでは、コンパイラは pdb stream をいったんファイルに書き出す。そのためには、PdbStream の派生クラス PdbFileWriter のオブジェクトに対してメッセージを送ればよい。PdbFileWriter オブジェクトは、受けとった pdb instruction に対応するコードをファイルに書き込む。

このファイルを再びメッセージ通信による pdb stream に変換するには、ファイルを読み込んでその中のコードに対応する PdbStream メソッドを呼ぶ簡単なツール (PdbFileReader) を利用する。PdbFileReader の出力する際のインターフェースは、コンパイラの場合と全く同じである。ファイルの読み書きに使われるコードは、PdbFileWriter と PdbFileReader 間のみで共有される。

pdb stream からプログラム・データベースを構築する場合には、PdbStream の派生クラスである DatabaseWriter のオブジェクト (DatabaseWriter) にメッセージを送ればよい。

このように抽象クラスを用いることによって、コンパイラや PdbFileWriter など情報を出力する側のプログラムには、その情報をファイルに書き込む場合でも、直接データベースに書き込む場合でも全く変更を加えないですむ。同様に、情報を受けとる側も、自分にメッセージを送ってくるツールがコンパイラであるか、PdbFileReader であるかは関知しなくてもよい。

現在、PdbFileWriter, DatabaseWriter の他に、pdb stream の pritty printer が PdbStream の派生クラスとして実現されている。図 3はこの pritty printer の出力である。

図 4.1は以上で説明した pdb stream によるツール間通信を図式化したものである。

4.2 DatabaseWriter

DatabaseWriter は pdb stream を受けとってプログラム・データベースを構築する仮想機械である。DatabaseWriter のインプリメンテーションを考える際には、DatabaseWriter がメッセージを受けとるだけの受身一方のオブジェクトであること、また、1個の pdb instruction は、必ずしもプログラム・データベースのオブジェクト 1個に対応しないことが重要となる。

例えば、1個のシンボルを表すオブジェクトを生成するには、1個の symbol instruction の他、いくつかの attributre instruction および type instruction が必要となる。したがって DatabaseWriter は、これらの pdb instruction に対応するメソッドが呼ばれると、その都度その情報を内部のキャッシュ領域あるいはスタックに保存する。そして、十分な情報がたくわえられたときはじめて、データベースに同一のシンボルを表すオブジェクトがないかどうかを調べ、なければオブジェクトを生成して登録する。

トップレベル構造に関する情報は、さらに多くの pdb instruction によって与えられるが、あるトップレベル構造がデータベースにすでにあるものと同一か否かは、その定義上、これにかかわるすべての pdb instruction を受けとってからでなければ判定できない。そのため DatabaseWriter はその間の情報をすべて内部にとっておき、最終的に不要になればそれを捨てることが必要となる。こういった DatabaseWriter の性質から、その実行中に大量のオブジェクトが生成され、不要になると壊されることになった。

例えば、われわれのデータベース・スキーマおよび DatabaseWriter のソース・ファイル (合計約 22000 行) を対象に用いた場合では、約 147 万個のオブジェクトが作られ、そのうち約 139 万個が結局不要になって消されており、最終的にデータベースに残るオブジェクトは 1 割にも満たない。これが実行効率に大きく影響を及ぼしたため、なんらかの工夫が必要となった。以後の実験はすべて、同じ例を用いて RS/6000 AIX 3.2 上で行なった。

Performance Tuning

Fast & fuzzy mode　複数のトップレベル構造が同一であるためには、そのトップレベル構造で参照している外部シンボルがすべて一致する必要があるが、実用上は特定のプログラム・データベース内において、同

図 4.1 pdb stream を使ったツール間通信

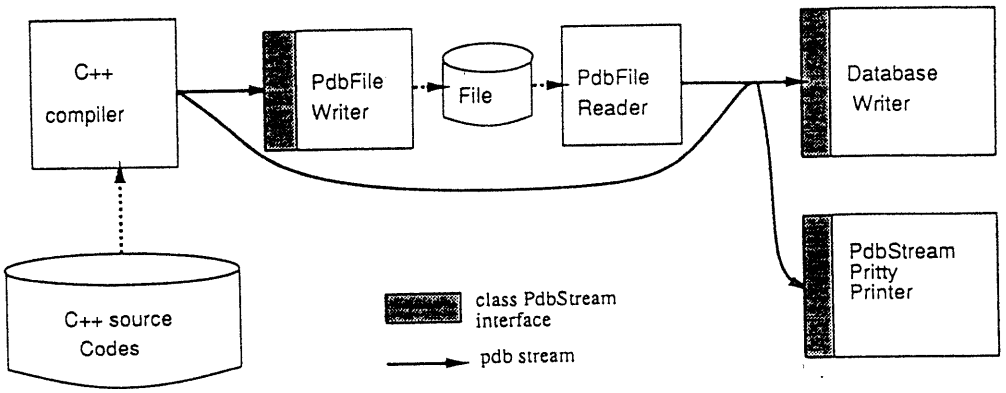


表 4.1 2 通りのモードでの実行効率

mode	no. of objects(K)		execution
	created	deleted	time(s.)
slow & strict	1472	1389	410
fast & fuzzy	585	505	253

一ファイルの同じ位置にあるトップレベル構造は同一の意味をもつと仮定してよい場合がほとんどである。そこで、正確であるが遅い従来の方式 (slow & strict) に加えて、同一位置から始まるトップレベル構造がすでにデータベースにある場合は、それ以後の pdb instruction を無視するモード (fast & fuzzy) を付け加えることにした。この結果、表 4.1 に示すように、生成・消去されるオブジェクトの個数が減り、実行効率も向上した。

Type 情報の扱い fast & fuzzy モードにおいても、依然として生成されたオブジェクトの 80 %以上が消されてしまっている。そこでそのクラス別の内訳をしらべてみたのが表 4.2 である。

この表から、最も大量に作られ、かつ消されているのが Type オブジェクト、次に Symbol オブジェクトであることがわかる。これは、3節でも見たようにシンボルの型情報が細分化した形で与えられるのに対し、DatabaseWriter がその情報をたくわえるのに、ひとつひとつオブジェクトを生成していたためである。そこで、実行効率をあげるため、Type オブジェクトを作ることを放棄し、数値コード化してみた。また Symbol オブジェクトについても生成をできる限り遅らせるようプログラムを書き換えた。その結果が表 4.3 である。実行時間は slow & strict モードで約 40 %、fast & fuzzy モードで約 50 %も短縮された。

表 4.2 オブジェクトの内訳 (fast&fuzzy の場合)

(数が少ないものは省いたため合計は TOTAL と一致しない)

kinds of objects	no. of objects(K)	
	created	deleted
Type	274	261
Symbol	199	190
Xref	68	32
Statement	25	10
TOTAL	585	505

表 4.3 Type オブジェクトを作らない場合の実行効率

mode	no. of objects(K)		execution
	created	deleted	time(s.)
slow & strict	1009	920	249
fast & fuzzy	103	17	134

4.3 拡張性

プログラムの動作をブラウザーを使って精密に理解する場合や、デバッガーなどに使うデータベースでは情報の正確さや詳しさが必要となるが、その一方で、開発途中のまだコンパイルできないプログラムから、例えばクラス階層図だけでも表示できればよいという要求もある。

そこで、われわれは C++プログラムの簡易パーザーを作って、pdb stream インターフェースを使って、DatabaseWriter につなげてみた。このとき DatabaseWriter の書き換えは全く不要であった。この簡易パーザーは、マクロ、テンプレートに関する処理を全く行なわず、シンボルの型情報も取り扱わないなど、はなはだ不完全なものではあるが、先ほどまでの例に対するクラス階層図を 10 秒程で表示できた。

このことからも、この pdb stream によるインターフェースが、ツールからの独立性・拡張性に優れているといえるだろう。またブラウザーの開発中、C++の言語仕様の複雑さから、当初与えられていた pdb instruction set の不備が相当数みつかったが、それらの追加・修正はたいへん容易であったことを記しておきたい。

5 まとめと今後の課題

ブラウザーなどプログラム開発支援環境で使用するためのプログラム・データベースに、ソース・プログラムからの情報を供給するためのインターフェースとして、pdb stream およびそのメッセージ通信によるインターフェースを提案した。これは、コンパイラが必要な情報を出力するときの負担をできるだけ少なくすることを主眼に設計されたが、簡易パーザーなどからも使えるものとなった。また pdb instruction set の拡張・修正が容易であったことなどから、prototyping をともなうツール開発においては得に有効だと思われる。

この pdb stream からデータベースを構築するときには、オブジェクトの生成・消滅が頻繁になるため、実行効率が問題となり、あまり細かな単位でオブジェクトをつくらないようにするなどの工夫が必要となった。

現在のシステムではデータベースをヒープ上に作っているが、一般にプログラム・データベースは巨大なものであり、われわれの使った例でも十数メガバイトに達し、頻繁なページ・フォールトによるパフォーマンスへの悪影響も無視できなくなっている。今後は persistent なオブジェクト指向データベースへの移行も考えているが、その際にはさらに実行効率に注意して、クラスタリングなどを行なう必要があるだろう。また、ヘッダー・ファイルのプレコンパイルなどにも対応できるよう拡張を考える必要もあるだろう。

謝辞 本研究は、日本アイ・ビー・エム (株) 東京基礎研究所における C++ ブラウザー開発の一環として行なわれたものである。数々の助言を与えてくれた久世和資氏をはじめとする同僚諸氏に感謝したい。

References

[1] Walter R.Bischofberger: Sniff - A Pragmatic Approach to a C++ Programming Environment *Proceedings of USENIX C++ Conference*(1992), 67-81.

[2] Judith E. Grass The C++ Information Abstractor *Proceedings of the USENIX C++ Conference*(1990), 265-275.

[3] Robert B. Murray: A Statically Typed Abstract Representation for C++ Programs *Proceedings of the USENIX C++ Conference*(1992), 83-97.