Sharing Between Translation Units in C++ Program Databases

Samuel C. Kendall
Sun Microsystems Laboratories, Inc.
2 Elizabeth Drive
Chelmsford, MA 01824
sam.kendall@east.sun.com

Glenn Allin
CenterLine Software, Inc.
10 Fawcett Street
Cambridge, MA 02138
glenn@centerline.com

ABSTRACT

A C++ program database represents information about C++ code, typically to enable program browsing or debugging. Such databases can grow very large. The growth is fundamentally due to the *translation unit* (*TU*) program structure C++ inherited from C: a naively designed database will consist largely of representations of redundant or unused code from header files.

This paper measures the effect of some techniques for shrinking this naively designed database: the *elision* of unused entities from a TU, and the *sharing* or *linking* (generically, the *combination*) of redundant entities across TUs. We also measure the overhead imposed by the *segregation* of class types with external linkage from those with internal linkage.

We define and measure these techniques for our own database, which was designed with very specific requirements. We also discuss techniques and organizations used in other program databases to save space: sharing at header file granularity; ruthless simplification of the database; and lazy loading of data into the database. Finally, we note the potential problems associated with independently implemented translators feeding into the same database.

1. Introduction

A C++ program database represents information about C++ source or object code, typically to enable program browsing or debugging. Such databases, if they do not take steps to avoid it, can grow very large. This growth is fundamentally due to the *translation unit* (*TU*) program structure C++ inherited from C: a program is a set of TUs, and declarations are shared between several TUs by placing them in header files and textually including the header files (by means of `#include` directives) in each TU. In C++ programs, broadly speaking, most of the source code a translator sees comes from header files. Much of this code is redundant, because a given header file is included in many TUs, and often much of the included code goes unused in any given TU that includes it. A naively designed database will consist largely of representations of this redundant and unused code, wasting time (the time needed to store the extra code) as well as space.

This paper measures the effect of two techniques for shrinking this naively designed database: the *elision* of unused entities from a TU, and the *sharing* or *linking* (generically, the *combination)* of redundant entities across TUs. We also measure the overhead imposed by the *segregation* of class types with external linkage from those with internal linkage; segregation turns out to be necessary to maintain semantic consistency.

To measure these techniques we employed a modified version of the ObjectCenter component debugging engine, which we call “our database”. The component debugging engine is part of CenterLine Software’s ObjectCenter C++ programming environment [Wyant]. Although the techniques we discuss are general, the results of our measurements are of course specific to our database. To place our measurements in context and to allow the reader to judge if they might apply to some other database, we explain the organization of our database in section 2.

Sections 3, 4, and 5 define elision, combination, and segregation respectively, and discuss these techniques. Section 6 measures the effect of these techniques on three medium-sized C++ program fragments and draws conclusions from the measurements. Section 7 discusses related work, including alternate techniques and database organizations that lead to smaller program databases. Section 8 concludes the paper.

Much of this paper is about class types. Many of the points about class types, we should note, also apply to enumeration types; for brevity we rarely mention enumerations explicitly.

2. Our Database

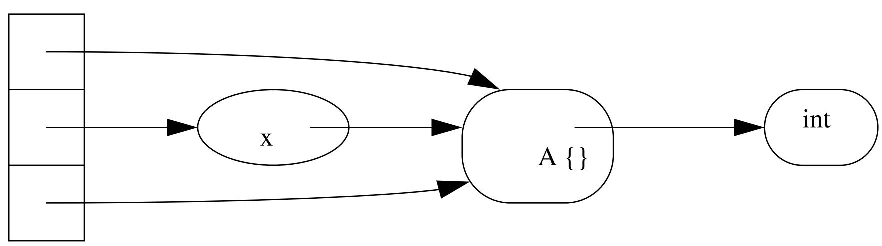
The ObjectCenter component development engine database supports browsing, debugging, interactive translation (to interpreter bytecodes) and execution of source code fragments, mixtures of object code and interpreted code, incremental TU linking, and mixtures of C and C++. The database is kept entirely in virtual memory. Tools access the database through a message-passing interface.

For simplicity of explanation, we do not discuss the entire database in detail. The portion of the database we discuss is sufficient to support browsing, interactive translation of source code fragments, incremental TU linking, and C++ only. Most of our code examples are confined to a subset of C++ containing only top-level variables, functions, and classes. We do measure (in section 6) a larger portion of the database than we discuss in detail.

Our C++ program database represents a program as a set of translation units (TUs); each TU is a sequence of declarations, and each declaration refers to an entity such as a class type, a variable, or a function. For example:

```
struct A;
extern A x;
struct A { int i; };
```

We represent the database that results from this TU as a diagram:



Boxes represent declarations. Ovals represent variables. Rounded rectangles represent types. Arrows represent relationships between entities. The “{}” after “A” indicates that A is defined.

To keep the diagram simple, we have omitted much: the name of `A::i`; that the first declaration of A is a forward declaration, while the second is a definition; that `x` is only extern-declared as opposed to defined. Also missing are tables that allow fast lookup of entities by name or by other attribute. These things, and others, are in our database. But the diagram captures the essentials we need for our discussion.

Note that if there is more than one declaration or definition for an entity in a TU, there is only one representation for that entity in the representation of the TU. For example, there is only one representation of `A` above, even though there are two declarations of that class. Representing declarations rather than declared entities can make sharing between TUs easier, as we discuss in section 7.2. Our database has the “one representation per TU” organization because that is what our C++ front end produces and consumes. CenterLine’s design philosophy has been to alter the C++ front end (which is supplied by a third party) as little as possible.

In the following sections we view our database as a directed graph; when we refer to “a graph” in this paper, we mean a single TU, or a database, considered as a directed graph. Nodes have attributes, and edges connect nodes. We do not give a complete list of the different kind of nodes, attributes, and edges; but the following abridged list should give an idea of what we mean:

A TU is a sequence of declarations. Each declaration has a Boolean attribute “is-a-definition” and an edge to the entity it declares. There are many different kinds of declared entities, among them top-level variables and class types.

Each top-level variable has a name attribute (a character string), a Boolean attribute “is-a-definition”, a linkage attribute (“external” or “internal”), and an edge to a type.

Each class type has a name attribute, a Boolean attribute “is-a-definition”, a Boolean attribute “is-a-structure” (if false, the class is a union), a linkage attribute (“external”, “internal”, or “none”), a sequence of edges to base nodes, a sequence of edges to friend classes and functions, and a sequence of edges to members. A base node has an access control attribute (“private”, “protected” or “public”), a Boolean attribute “is-virtual”, and an edge to a class (the base class).

There are many kinds of members. Data members, for example, have a name attribute, an access control attribute, a Boolean attribute “is-static”, and an edge to a type.

Our diagrams of databases are simplified versions of these directed graphs, with only the most nodes, attributes, and edges pictured.

A TU is added to a database in the following manner. First, the source code for the TU is translated into a stand-alone graph. Second, this graph is added to the graph which is the database. We call these phases *TU translation* and *TU combination,* respectively; they correspond to compilation and linking in a traditional system.

The next three sections describe elision, combination, and segregation. The scenarios we measure are determined by whether each of these techniques is invoked during TU combination (actually, elision is invoked just before TU combination). Algorithms to implement these techniques are beyond the scope of this paper, but we have taken pains to accurately define the terms (elision, sharing, linking, and segregation) that capture the effect of our algorithms.

3. Elision

Informally, elision means “omit everything from a TU that is not used”. To define elision more precisely, we define first a source-to-source transformation on TUs:

> *Elision transformation*: the omission of all non-defining declarations, and the omission or demotion to forward declarations of all class definitions and inline function definitions, unless the omission or demotion would render the code ill-formed or would remove a member of a class.

For example:

```
TU before elision trans.          TU after elision trans.

struct A { int i; };              struct A;
struct B { A* ap; };              struct B { A* ap; };
struct D : B {};                  struct D : B {};
extern A a;
void f();
D d;                              D d;
```

From bottom to top: `d` is kept because it is a definition. `f` and `a` are discarded because they are unused. `D`’s definition is kept because `d` depends on it, and `B`’s definition is kept because `D` depends on it. `A`’s definition is demoted to a forward declaration. `A` is not eliminated entirely because `B`’s definition depends on its forward declaration.

Elision, then, is the discarding of part of a graph as though the elision transformation had been applied to the source code. Elision is a filter on graphs, applied just before TU combination.

We define elision in terms of a source-to-source transformation as a way of preserving invariants the violation of which would discomfit a database client. For example, a more aggressive elision algorithm might elide the body of `B` above, even though `B` is a base class of `D`; and we could enable our database to store the odd result, a class (`D`) with a "base class" edge to an undefined class (`B`). But class browsing tools, many written before elision was implemented, will be surprised to find a base class with no definition; some of these tools may crash when confronted with the undefined base class.

A happy property of elision is that an entity is completely missing from a database only if it is used in none of the TUs. For example, suppose a particular type is declared in ten TUs, but only used in two of them. Even though that type may be elided from eight TUs, it will still be present in two of the TUs, and thus it will be in the database and available to the user.

In our experience elision is acceptable for a debugging database. Occasionally the declaration of an entity the debugger user wants to access has been elided; but usually this is not the case, due to that happy property.

For a browsing database, however, elision may not be acceptable [Grass]. Cia++ [Grass & Chen], for example, can be used to discover unused code or needless header file inclusions. Elision would destroy the data needed to make these discoveries.

There are no C++ language issues with elision—elision was carefully defined to avoid them—except that one must be careful to compute the linkage of a type before performing elision. In this example, the elision transformation changes class `A`'s linkage from external to internal:

```
// TU before elision trans.   // TU after elision trans.

struct A { int i; };          struct A { int i; };
void f(A);
static A a;                   static A a;
```

Our implementation incorrectly computes the linkage of a class after elision. We expected this bug to generate many classes with falsely internal linkage, but in practice this did not occur very often. In most cases where all externally linked uses of a class were elided, the class itself was elided or at least demoted to a forward declaration. Linkage is discussed in section 5.

4. Combination

Combination is the merging of nodes in the graph in order to reduce the number of nodes. Note that "combination" is different than "TU combination"; TU combination (the inserting of the graph for a TU into a database) may be performed with combination (the merging of individual entities) or without it.

The word "sharing" is used loosely in the title of this paper. More precisely, this paper is about *combining* representations of C++ language entities between TUs. Sharing is one mode of combination.

We define two modes of combination:

> *Sharing*: two entities are sharable if their attributes (as discussed in section 2) are equal and corresponding edges go to sharable nodes.
>
> *Linking*: two entities are linkable if their attributes are equal and corresponding edges go to linkable nodes; or if both are class types with the same tag, one is a definition, and the other is a forward declaration.

The difference between sharing and linking is that linking will combine a class forward declaration with a class definition of the same name, while sharing will not. For example:

```
TU1             TU2                     TU3

struct A;       struct A { int i; };    struct A { int i; };
```

Figures 1, 2, and 3 are the diagrams for these TUs with, respectively, no combination, sharing, and linking.

Figure 1: No combination

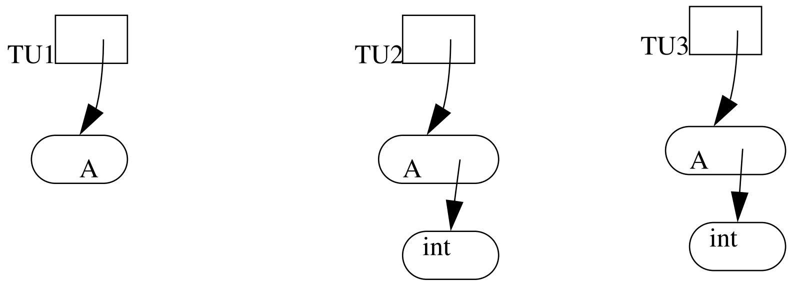


Figure 2: Sharing

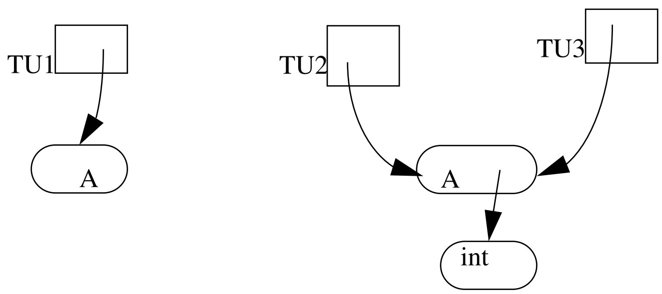
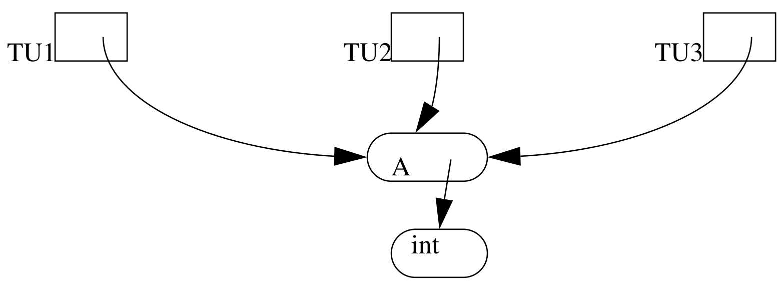


Figure 3: Linking



If we look closer at these definitions of sharing and linking we find two problems. The first problem is that a graph with cycles—for example, a class containing a pointer to itself—can result in infinite regress when one tries to apply these definitions. The corrected definitions are more cumbersome and can be found in the Appendix.

The second problem is that, although we define the relations sharable and linkable, we do not say which sharable types are shared, nor which linkable types are linked; we have not really defined sharing and linking. There is an easy solution for sharable types: share all types that are sharable. But for linkable types this is impossible. For example, consider the following three TUs:

```
TU1                TU2                TU3

struct B;          struct B {         struct B {
                         int i;             char c;
                   };                 };
struct C {                            struct C {
      B* bp;                                B* bp;
};                                    };
```

The first `B` is linkable with the second `B` and the third `B`—but it can only be linked with one of them. Furthermore, the two `C`'s, which are linkable, can only be linked if the first and third `B`'s are linked, not if the first and second B's are linked.

In our database linking is performed eagerly during TU combination, and a linking decision is never "rethought" after it is made. For example, if the TUs above were added to our database in numerical order, the result would be two defined B's and two defined C's; but if the TUs were added in the order TU1, TU3, TU2, the result would be two defined B's and only one defined C.

Note that the state of the database with linking depends on the order in which TUs are added. This is an unfortunate property, because it can make bugs more difficult to reproduce. Sharing does not have this problem.

In the following section we explore situations where there are different class definitions with the same tag.

5. Segregation

In C++, classes come in three varieties:

- *external classes* (their tags have *external linkage*);
- *internal classes* (their tags have *internal linkage*); and
- *local classes* (local to a function; their tags have *no linkage*).

A class in file scope is external if and only if another external entity (ultimately, a variable, function, static data member, or non-inline member function) depends on it. For example:

TU1

```
struct A { int i; };           // This A has external linkage
void f(A);                     // because f depends on it.
```

TU2

```
struct A { int i; };           // This A has internal linkage.
void g() {
      struct A { int i; };     // This A has no linkage.
}
```

The fundamental difference between external and internal classes is that C++ requires all definitions of an external class with a given tag to be identical; but there is no such restriction for internal classes.

We probably want to *segregate* these three kinds of types, not to combine one kind with another. For example, although the three definitions of A above appear sharable or linkable, if we are segregating according to linkage then they must remain separate nodes.

The variants we measure in section 6 are:

- "No segregation": no distinction is made between internally linked and externally linked classes.
- "Segregation": internally linked and externally linked classes are segregated. The two kinds of classes are combined independently of each other, and can have different combination modes.

In both variants local classes are never combined with each other, nor with internal or external classes. In the programs we measure there are very few local classes.

C++ language issues have an impact on segregation and the combination modes for internal and external classes. Most importantly, this example gives rise to a semantic error unless we segregate:

```
TU1                 TU2                        TU3

struct A;           struct A {                 struct A {
                          char c;                    int i;
                    };                         };
                    //  internal class
extern A x;                                    extern A x;
```

The `A`'s in TU1 and TU3 are external; the `A` in TU2 is internal. If these TUs are TU-combined in numerical order, and we do not segregate, then the `A`'s in TU1 and TU2 will be linked, leading to the variable `x` incorrectly seeming to have two conflicting types. If we segregate, then the right thing happens: the `A`'s in TU1 and TU3 are linked, and there is no problem with the type of `x`.

Segregation is also necessary if we are to diagnose a violation of the one definition rule (discussed below) for external classes; and segregation is of course necessary if a we want a browser to distinguish internal from external classes.

Clearly we would prefer to segregate. But given that we want to segregate, we must decide how (if at all) to combine each kind of class. We consider external classes, then internal classes; for each, the answer hinges on a C++ language issue.

External classes fall under the *one-definition rule* (*ODR*).[1] The ODR states that all external definitions of a tag must be the same. Since all external definitions of a tag are the same, they, together with all external forward declarations of the tag, can be linked to form a single definition. This implies that we should link external classes.

We may even be tempted to think that we can link external classes using a simplified, easier-to-implement algorithm that does not tolerate differing definitions of the same tag. If we encounter a non-linkable definition, so the argument goes, we can give a fatal error message and reject the TU from the database. Unfortunately, no C++ implementation that we know of enforces the ODR, and so we expect that many programs violate it accidentally. One of our example programs, cxxcomp, violates the ODR deliberately.[2] As a result, a database that aims to handle existing code cannot restrict itself to programs that conform to the ODR, and a linking algorithm must handle differing definitions of the same external tag.

Internal classes might be said to fall under the “many-definition rule”. Even if internal class definitions in separate TUs are identical, they still define different types. One can construct a program to verify this using run-time type information (RTTI) [Stroustrup & Lenkov], a set of features recently added to C++ that allow explicit manipulation, and testing for equality, of representations for the types of expressions. In fact RTTI (and exceptions) require a C++ compiler to generate a database of types, called the typeinfo database, to be accessible by each program at run-time. Much of what this paper says applies to the design of a typeinfo database. In the next section, scenario D was included because of its possible use in a typeinfo database.

Two corner cases, tagless classes and nested classes, complete our survey of issues related to segregation.

Tagless classes are a conundrum in C++, since “C++ relies on name equivalence, *not* on structural equivalence of types” [Ellis & Stroustrup]. The ANSI/ISO C++ committee is currently debating the status of tagless classes. Our implementation sidesteps the issue by generating a unique pseudo-tag for each tagless class, then allowing structurally equivalent tagless classes to be combined between TUs in spite of their differing pseudo-tags. Linkage for tagless classes is computed just as for tagged classes.

Linkage for nested classes is also ambiguous in C++. Does external linkage propagate from a class to a class nested within it? From a nested classes to its containing class? For our implementation the answer is “yes” and “no”, respectively, but the “no” could be argued with.

---

1. The ODR is found only in implicit form in [Ellis & Stroustrup]. The need for it is generally accepted in the ANSI/ISO C++ committee, but its precise definition is still being debated.
2. The cxxcomp sources violate the ODR as follows: many member function declarations are conditionally compiled out in most, but not all, TUs. This speeds up compilation of the cxxcomp sources.

6. Measurements

We measured three medium-sized C++ program fragments loaded into our database, in each case trying all combinations of the parameters elision, segregation, and combination mode for which we could imagine a use. We call a particular combination of the parameters a *scenario*. A scenario is named with a letter, indicating the segregation and combination modes (see table 1), followed by a "+" or "–" that indicates, respectively, no elision or elision. (The "–" is intended to suggest that elision "subtracts" entities.) For example, the scenario E– is elision, segregation, linking of classes with external linkage, and sharing of classes with internal linkage.

Table 1: Scenarios Measured

Scenario Letter	Segregation	Combination Mode	
		External Classes	Internal Classes
A	Yes	No combination	No combination
B	Yes	Share	Share
C	No	Share	
D	Yes	Link	No combination
E	Yes	Link	Share
F	Yes	Link	Link
G	No	Link	

Scenarios A (that is, A+ and A–) are the "naive database", with no combination.

Skipping ahead for a moment, scenarios G, linking with no segregation, are what the current release of ObjectCenter implements. Scenarios B through F are the alternatives we considered.

Scenarios B and C are sharing with, respectively, segregation and no segregation. Because sharing is much simpler to implement than linking, it is a viable implementation choice. We were interested to see how much larger the database would be with sharing than with linking.

Scenarios D, E, and F all involve linking of external classes. For internal classes they involve no combination, sharing, and linking, respectively. D is the most straightforward implementation for a database that must be completely accurate about types, such as a typeinfo database, since an internal class in one TU has no relationship to an internal class in another TU (even if they look identical). E allows a smaller, but less straightforward implementation of a typeinfo database: each reference to a type must be annotated with an identifier for the TU the type is in. The TU annotation allows two internal classes whose representations are shared to be distinguished. A database with fewer accuracy constraints may choose to implement E without the TU annotation. Finally, scenarios F produce databases even smaller than E, but at the price of a further inaccuracy: the linking of an incomplete internal class in one TU with an arbitrary complete internal class with the same name in some other TU.

The program fragments we measure are shown in tables 2 and 3.

The measurements do not consider the additional libraries normally linked with these programs—for example, the cbrowse measurements do not include the OI class library—but the measurements *do* consider the included header files from those libraries.

The line counts reveal that we have chosen three programs with different organizations. Most of the lines of cxxcomp are in the top-level source files; cbrowse is mostly header files; iv is about half and half. Note that

Table 2: Programs Measured

Name	Header Files	TUs	Description
iv	247	53	Some of the “doc” editor, and some of the InterViews 3.1 library itself.
cbrowse	289	10	Some of the ObjectCenter user interface. This program uses the OI class library [Aitken], which has many large header files.
cxxcomp	80	42	A proprietary C++ compiler.

Table 3: Line Counts on Programs Measured

Program	Raw		Cooked[a]		After pre-processing, cooked
	Header files	Top-level sources	Header files	Top-level sources	
iv	19661	15434	12013	12310	58280
cbrowse	45994	4615	31109	3743	182586
cxxcomp	18723	69627	14877	54434	169890

a. “Cooked” means excluding blank lines and comments.

even for cxxcomp, most of the lines a translator sees after header file expansion (see the “after preprocessing, cooked” column) come from header files.

Two other differences in these programs are important: cxxcomp is a complete program (except for the C++ standard library), while the other two are fragments; and in cbrowse essentially the same (large) set of header files are included in each top-level source, while the other two programs are less uniform in their inclusion.

We measure two quantities in our database. Our first quantity is the *number of defined classes* in the database. We rely on this one measurement because it is a reasonable predictor of the database size, at least for our database; and because it is an abstract number, mostly independent of the details of our database.

It is not surprising that the defined class count is a good predictor of database size: the representation of a defined class in our database is larger than the representation of nearly any other language entity.

Our second quantity is the *size of the C++-related database* in kilobytes. We are measuring the data structures produced by our C++ front end to represent functions (though not their executable code), variables, templates, and types in the database. We do not measure representations of identifiers and macros, nor do we measure executable code and other back end data structures, although these are also in our database.

Figures 5 and 6 plot our two quantities against the fourteen scenarios. Figure 5 plots the number of defined classes (vertical axis) for each scenario, and figure 6 plots the total database size (vertical axis) for each scenario. Low points are better (fewer classes or a smaller database).

Each graph has two sets of points. The first set are for the “+” (no elision) scenarios and are represented by a “+”, though this mark often looks like a vertical tic. The second set are for the “–” (elision) scenarios and are represented by a square. The lines connecting the points in a set show trends: a slope from upper left to lower right is an improvement, the steeper the better.

Several features are apparent in these graphs.

**Figure 5: scenarios vs.
number of defined classes**

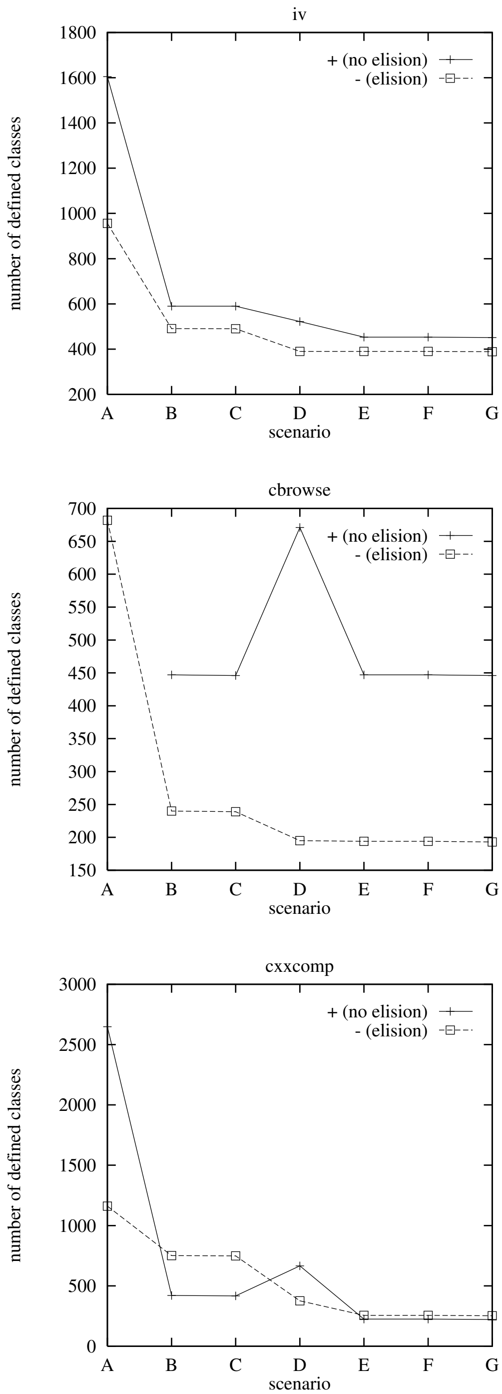
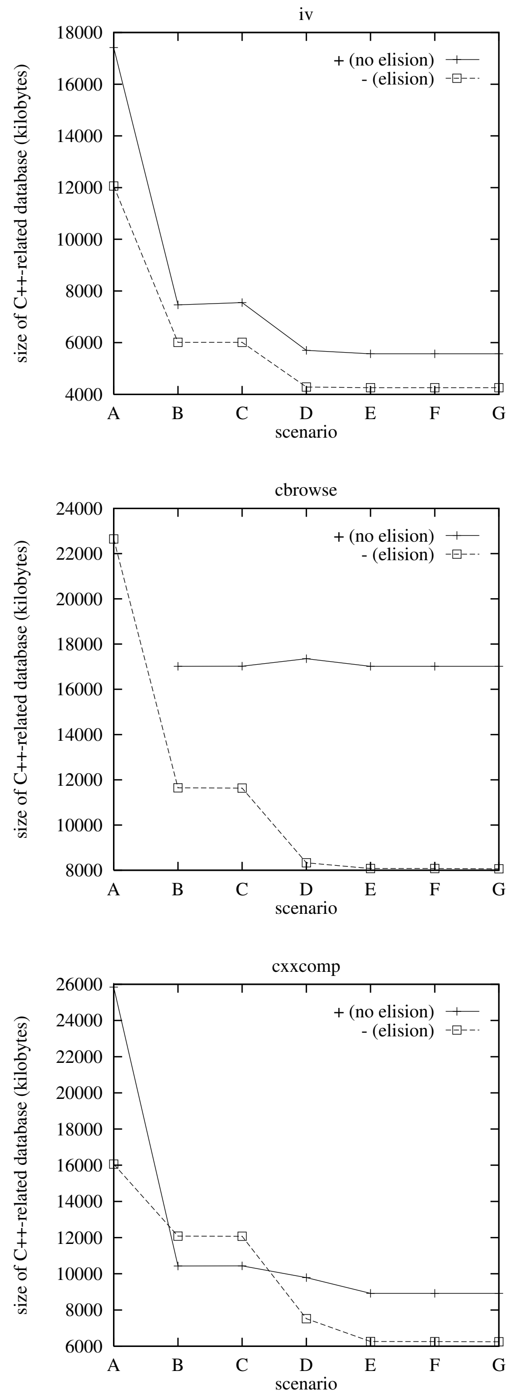


**Figure 6: scenarios vs.
size of C++-related database (KB)**



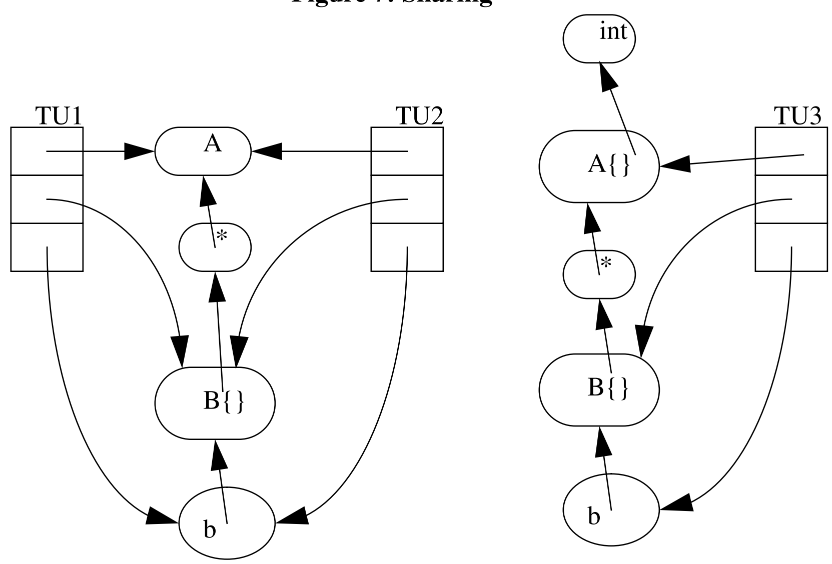
- The improvement is dramatic between no combination and sharing (scenarios A vs. B).
- Between sharing and linking (scenarios B vs. F, or C vs. G), the improvement is less dramatic but still significant. The following example suggests why this is so:

```
TU1               TU2               TU3

struct A;         struct A;         struct A {
                                          int i;
                                    };
struct B {        struct B {        struct B {
      A* p;             A* p;             A* p;
};                };                };
extern B b;       extern B b;       extern B b;
```

  No combination produces three nodes for `B`. Linking combines them all and produces one node for `B`. Sharing (scenario B+ for this example) produces two nodes for `B`; it is diagrammed in figure 7. Because

Figure 7: Sharing



  `A` cannot be shared between TU3 and the other TUs, `B` (which depends on `A`) cannot be shared either. We call this a *cascading combination failure*. In actual programs classes are often heavily interdependent; as a result, when one class fails to share (or link) the cascade can be many tens of classes.

- There is no data for the naive (A+) scenario of cbrowse, because that scenario exceeded a normally reasonable fixed limit in our database.
- Again for cbrowse, in the absence of elision there was no improvement from sharing to linking (B+ vs. F+). This is because the TUs of cbrowse are homogeneous; they all include the same header files, and so they all declare the same classes in the same way. Without some difference between TUs, sharing does not fail.
- Elision is uniformly better than no elision for no-combination (scenario A) and for linking (scenarios E, F, and G).[1] This is not surprising, since elision deletes classes.

---

1. A minor bug in our implementation of elision makes elision look very slightly worse than no-elision for cxxcomp scenarios E, F, and G in "number of defined classes". In reality elision should look very slightly better for those scenarios. No other result was significantly affected by this bug.

- Elision sometimes does not interact well with sharing. This is evident in cxxcomp scenarios B and C. The reason is that elision can demote a class definition to a forward declaration in one TU but not others, creating an opportunity for cascading combination failure. In general, sharing prefers homogeneity, and elision destroys homogeneity. One might expect to observe this interaction with cbrowse, since its TUs are so homogeneous; but so many defined classes go unused in cbrowse that their removal by elision (which reduces the number of classes) overwhelms the bad interaction (which increases the number of classes).
- For cxxcomp scenarios E, F, and G, "number of defined classes" appears not to favor elision over no elision. Yet elision is sharply favored in "size of C++-related database". This disparity seems to be due to elision's effect on entities other than classes, notably function and variable declarations.
- The improvement (slope) is steeper between scenarios C– and D– in plotting database size than in plotting number of defined classes. We believe this is due to a subtle effect that we have also observed independently: cascading combination failure hits large, complicated classes more often than small, simple classes. As a result sharing (which has many cascading combination failures) produces a larger database than the number of defined classes would lead one to believe; linking (which has far fewer cascading combination failures) does not.
- In these programs, at least, internal and local classes are insignificant. In none of our programs are there more than two local classes or enumerations. As for internally linked types, they seem significant only in scenario D+ for cbrowse and cxxcomp, where their numbers are inflated by the lack of elision and combination. But even in those scenarios the inflation is insignificant when we consider the size of the database. This disparity between number of defined classes and database size is because most internal classes are simple, and so have small representations.

7. Related Work

7.1 Sharing at Header File Granularity

Most modern programming languages comparable to C++—for example, Ada and the Modula family—are organized around modules. An "import" construct takes the place of `#include`. An imported module, unlike an included header file, has the same meaning in all contexts. As a result, modules provide a natural structuring unit for a program database. For C++, header files can be used as such a unit, but the database must be prepared to see multiple meanings for one header file.

The combination we have talked about so far has all been at *fine granularity*; that is, individual variables, types, etc. are combined (or not). Some databases, however, share at *header file granularit*y: all the declarations in a header file are shared (or not) as a unit. These include some versions of the Sun C debugging information database [Linton] [Menapace], the DEC C++ debugging information database [Hamby], the Sun C++ source browser database [Pelegri-Llopart], and the Rational C++ Analyzer [Wilcox] [Wilcox94], part of Rational Rose/C++ product.

Elision essentially prevents sharing at header file granularity, for a different set of entities will likely be elided from a given header file in each TU that includes it.

Since a difference in any one declaration in a header file prevents sharing, great attention must be paid to minimizing such differences. Following is a list of other items that prevent sharing at header file granularity. Most of these factors are inherent in the C and C++ languages.

- Conditional compilation. Here are two common examples, the first surrounding the other:

```
#ifndef UNIQUE_TO_THIS_HEADER
#define UNIQUE_TO_THIS_HEADER

#if DEBUG
/* Declarations... */
```

```
#endif
#endif /* UNIQUE_TO_THIS_HEADER */
```

These very common idioms often cause one instance of a header file to contain a different sequence of declarations than another.

- Redundant declarations. Many translators do not emit a record on any but the first of a sequence of redundant declarations. If several header files contain a declaration `struct A;` and those header files are included in various orders, some inclusions of a given header will seem to declare `A` and some will not.
- Implicit declarations, e.g.:

  ```
  struct A *p;        // Implicit declaration of A.
  ```

  Most translators do not emit a record on an implicit declaration unless it is the first declaration. This reduces sharing for the same reason that redundant declarations do.
- Automatic generation of source code. Some language processors generate C or C++ source code. These processors often do not emit header files at all, thus defeating any header file-based sharing scheme.
- Static variables. If the description of a static variable includes its address (as in some debugging information formats), then no two inclusions of a header file will be sharable.
- Inline functions. When bodies are generated for these, as is common, they can cause the same sharing problem as static variables.
- Generated names for unnamed types, anonymous unions (which are not types), and function parameters. Some C++ compilers generate a unique identifier for an unnamed entity. A naive name-generation algorithm will generate different names for the same entity in different TUs.

The last two problems are unique to C++. The others affect both C and C++. Nevertheless sharing at header file granularity has proved useful for many databases.

7.2 Database of Declarations

Our database is oriented around such entities as types, variables, and functions—the entities that declarations declare. It is these entities we have elided, combined, and measured. An alternative organization taken by many databases, particularly databases intended to support browsing, is to store only the declarations. Such an organization is very flexible, and it is more amenable to sharing.

One way to categorize program databases is how close they are to the program source, or to the machine code (figure 8). Sharing is easier when a database is closer to source code. For example, in a database of

Figure 8: Spectrum of C++ Program Databases

source code	tokenized source code	database of declarations	our database	object code

tokenized source code sharing is trivial: just share the tokenized representations of header files. On the other end, object code is almost never shared between TUs.[1] In the middle, we have databases of declarations, for which sharing is relatively easy. For our database, as we have seen, sharing is possible but not optimal; we are forced to introduce linking, which is more complicated, to save additional space.

1. As an example of sharing object code between TUs, a linker might share function bodies that happen to be identical. Such redundant functions can be generated by template instantiation.

7.3 Multi-Program Database

For the most part we have assumed that a database contains at most one program. But some databases, such as the Sun Source Browser database, can contain many programs in one database. The essential restriction in designing such a database is that you may share entities, but you should probably not link them. The database may contain any number of definitions for an entity E; to link a forward declaration for E, one would have to pick one of those definitions. Any choice is semantically meaningless or even incorrect; the chosen definition may be from an entirely different program.

This is the same problem that confronted us for classes with internal linkage, but in a multi-program database this problem affects all named entities. Namespaces (a new C++ feature) may ameliorate this problem for newly written code.

7.4 Lazy Loading

Gdb, the GNU debugger, does not combine or elide. Instead, it loads debugging information from object file(s) into its in-memory database in a lazy fashion: the debugging information for a TU is only loaded when absolutely necessary. This trick is usually portrayed as a time-saver, but it is also a space-saver.

Gdb would probably benefit from combination, which is orthogonal to lazy loading. Elision is not possible, since there is not sufficient dependency information in object files.

7.5 Ruthless Simplification

Cia and cia++ are designed to handle large programs. We believe that the design achieves its scalability not so much by tricks of the kind we have presented—although both cia and cia++ seem to implement linking—as by ruthless simplification. They store information only about global entities and the relationships between them; the rationalization is that these relationships are the most important in examining the structure of large programs.

The opposite approach is that taken by Reprise [Rosenblum & Wolf], the Rational C++ Analyzer, and Alf [Murray]. These AST (abstract syntax tree) databases store nearly everything about the program. One of them, Reprise, has no combination between TUs. Rational has a particularly radical form of sharing at header file granularity, in which header files can be user-specified to behave as though they were importable modules. [Murray] hints that Alf will also prefer header files that have the same meaning in all contexts.

7.6 Multiple C++ Translators

One final point we feel is crucial, although it is not reflected in our measurements. In these examples, our database has only one C++ translator generating graphs. This is true for most other databases. Were a database to have multiple, independently implemented translators generating graphs, then combination and enforcement of the ODR would become problematic.

Because there are multiple implementors each interpreting an inevitably ambiguous specification of what output the database expects, small details in the representation of programs will differ, and any difference is enough to prevent sharing or linking. We have observed these differences in the various C compilers that generate STABS debugging information. Simple details that tend to differ are such things as representation of classes and enumerations with no members; and the distinction between types with identical machine lengths and formats, such as (on many systems) `char` and `signed char`, or `double` and `long double`.

Given that a C++ program database can be rougly as complicated as the language itself, it is difficult to imagine that these problems can be completely avoided. The database that will face these problems in the near future is the typeinfo database; luckily, the requirements on this database are simple enough that it should be possible to give it an unambiguous representation specification.

8. Conclusion

We state our conclusions as advice to designers of program databases on how to save space.

Our first set of advice is aimed at those designing databases with constraints similar to ours: no more than one representation for a given program entity in a given TU; combination must generate data structures designed for no combination, or for handling no more than a single TU. These databases might include a debugging information database (on disk or in memory) or a run-time typeinfo database.

- Segregation of internal from external classes is necessary for correctness, and imposes no significant space penalty.
- Most classes are external, so how external classes are combined is important. Future releases of ObjectCenter will probably continue to link external classes and to elide by default.
- The other reasonable alternative for external classes is sharing. In this case elision should probably not be available or at least should not be the default, since it does not mix well with sharing. This alternative is much easier to implement.
- Internal classes may be not-combined, shared, or linked. No-combination preserves program semantics most rigorously, but this kind of correctness is probably only important to a typeinfo database. Sharing preserves semantics except for type identity, and is the mode future ObjectCenter implementations will probably use for internal classes. Linking gives false definitions to undefined internal classes.

Our second set of advice is more general, aimed at those designing C++ program databases. We have no definite advice, but we can pass on strategies used in other databases:

- Sharing or linking of classes will almost certainly save space and time.
- Ruthlessly simplify and limit the information in the database, keeping only what is necessary for your purposes. This strategy, we believe, enables the cia++ database to handle large C++ programs.
- Represent only declarations, not declared entities. Sharing becomes easier and more profitable. This is the strategy used in browsing databases such as the Sun Source Browser and the Rational C++ Analyzer. If you are sharing at header file granularity, do not elide unused declarations.
- If your database is built by extracting data from another database, extract lazily. This strategy is used by gdb.
- If there are multiple, independently implemented translators feeding into your database, combination will probably suffer due to slight variations in output between the translators. Combat this problem by ensuring that the output of the translators is rigorously specified.

It is an interesting question whether an implementation should diagnose a violation of the one-definition-rule for classes with external linkage. It is clear that a debugging environment such as ObjectCenter should. However, we conjecture that many existing programs will violate this rule; it is important to allow them to execute.

Acknowledgments

We would like to thank those the many patient people who answered questions from the authors, particularly Geoff Wyant, Judy Grass, Robin Chen, Tom Wilcox, Ed Pellegri-Llopart, and David Chase.

Thanks are due to Geoff Wyant, Ann Wollrath, and Webb Stacy for their helpful comments on drafts of this paper. Particular thanks are due to Jim Waldo, whose suggestions and detailed comments on drafts were timely and very insightful.

Of course, any mistakes are the fault of the authors or of the computers they used to prepare the paper.

Trademarks

SPARCworks is a trademark of SPARC International, Inc., and is licensed exclusively to Sun Microsystems, Inc. ObjectCenter is a trademark of CenterLine Software, Inc. Rational and Rose/C++ are trademarks of Rational. Other product or service names mentioned herein are trademarks of their respective owners.

References

[Aitken] G. Aitken, "OI: A Model Extensible C++ Toolkit for the X Window System," *Proc. 4th Annual X Technical Conference*, January 1990.

[Chen] Y.-F. Chen, "The C Program Database and Its Applications", *Proc. USENIX Summer '89*, pp. 157-171.

[Chen94] Y.-F. Chen, personal communication, 1994.

[Ellis & Stroustrup] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Grass & Chen] J. E. Grass and Y.-F. Chen, "The C++ Information Abstractor", *Proc. USENIX C++ '90*, 1990, pp. 265-275.

[Grass] J.E. Grass, personal communication, 1994.

[Hamby] J. Hamby, personal communication, c. 1992.

[Linton] M. Linton, "The Evolution of Dbx", *Proc. USENIX Summer '90*, June 1990, pp. 211-220.

[Menapace] J. Menapace, J. Kingdon, and D. MacKenzie, "The 'stabs' debugging information format". In file gdb-4.12/gdb/doc/stabs.texinfo in the distribution of GDB version 4.12, available from numerous ftp servers such as prep.ai.mit.edu.

[Murray] R. B. Murray, "A Statically Typed Abstract Representation for C++ Programs", *Proc. USENIX C++ '92*, 1992, pp. 83-97.

[Pelegri-Llopart] E. Pelegri-Llopart, personal communication, 1994.

[Rosenblum & Wolf] D. Rosenblum and A. Wolf, "Representing Semantically Analyzed C++ Code with Reprise", *Proc. USENIX C++ '91*, 1991, pp. 119-134.

[Rovner] P. Rovner, *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language*. Xerox PARC technical report CSL-84-7, July 1984.

[Stroustrup & Lenkov] B. Stroustrup and D. Lenkov, "Run-Time Type Identification for C++ (Revised yet again)", doc. no. X3J16/92-0121 WG21/N0198, ANSI/ISO C++ Committee Mailing, 1992.

[Wilcox] T. R. Wilcox, USENIX C++ Advanced Topics Workshop presentation, 1992.

[Wilcox94] T. R. Wilcox, personal communication, 1994.

[Wyant] G. Wyant, J. Sherman, D. Reed, and S. Chapman, "An Object-Oriented Program Development Environment for C++", *Conf. Proc. of C++ At Work 1991*.

Appendix: Corrected Definitions of Sharable and Linkable

The corrected definition of sharable is as follows:

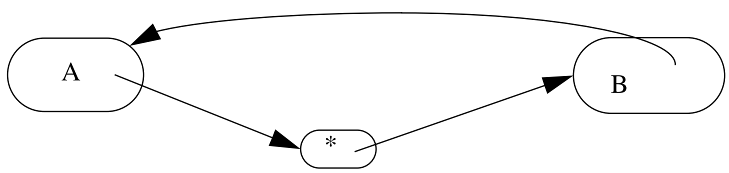
> Two types T and U are *sharable* if every pair of paths of equal, finite length, beginning at T and U and proceeding at each step along corresponding edges, terminate at entities whose attributes are equal, and which possess only edges corresponding to edges in the other.

The definition of linkable is corrected in a similar fashion. Note that paths of zero length are part of the definitions, and are necessary for types that are *not* involved in a cycle.

For example, two translation units containing

```
struct B;
struct A { B* p; };
struct B : A {};
```

result in two copies of the following type graph:



When we try to apply the original definition of sharable to the A's, we find that the definition of whether the A's are sharable depends on itself. When we apply the corrected definition, we find that although we have to examine an infinite number of (pairs of) paths:

> [A], [A, B*], [A, B*, B], [A, B*, B, A], ...

the two A's are indeed shareable.

The infinity can be removed, though doing so is not necessary and makes the corrected definitions even more cumbersome.