GNU C++ Renovation Project

Phase 1.3

Brendan Kehoe, Jason Merrill,
Mike Stump, Michael Tiemann

Edited March, 1994 by Roland Pesch (pesch@cygnus.com)

1 Introduction

As you may remember, GNU C++ was the first native-code C++ compiler available under Unix (December 1987). In November 1988, it was judged superior to the AT&T compiler in a Unix World review. In 1990 it won a Sun Observer "Best-Of" award. But now, with new requirements coming out of the ANSI C++ committee and a growing backlog of bugs, it's clear that GNU C++ needs an overhaul.

The C++ language has been under development since 1982. It has evolved significantly since its original incarnation (C with Classes), addressing many commercial needs and incorporating many lessons learned as more and more people started using "object-oriented" programming techniques. In 1989, the first X3J16 committee meeting was held in Washington DC; in the interest of users, C++ was going to be standardized.

As C++ has become more popular, more demands have been placed on its compilers. Some compilers are up to the demands, others are not. GNU C++ was used to prototype several features which have since been incorporated into the standard, most notably exception handling. While GNU C++ has been an excellent experimental vehicle, it did not have the resources that AT&T, Borland, or Microsoft have at their disposal.

We believe that GNU C++ is an important compiler, providing users with many of the features that have made GNU C so popular: fast compilation, good error messages, innovative features, and full sources that may be freely redistributed. The purpose of this overhaul, dubbed the GNU *C++ Renovation Project*, is to take advantage of the functionality that GNU C++ offers today, to strengthen its base technology, and put it in a position to remain—as other GNU software currently is—the technical leader in the field.

This release represents the latest phase of work in strengthening the compiler on a variety of points. It includes many months of work concentrated on fixing many of the more egregious bugs that presented themselves in the compiler recently. In the coming months, we hope to continue expanding and enhancing the quality and dependability of the industry's only freely redistributable C++ compiler.

2 Changes in Behavior in GNU C++

The GNU C++ compiler continues to improve and change. A major goal of our work has been to continue to bring the compiler into compliance with the draft ANSI C++ standard, and with *The Annotated C++ Reference Manual* (the ARM). This section outlines most of the user-noticeable changes that might be encountered during the normal course of use.

2.1 Summary of Changes in Phase 1.3

The bulk of this note discusses the cumulative effects of the GNU C++ Renovation Project to date. The work during its most recent phase (1.3) had these major effects:

- The standard compiler driver `g++` is now the faster compiled version, rather than a shell script.
- Nested types work much better; notably, nesting is no longer restricted to nine levels.
- Better ARM conformance on member access control.
- The compiler now always generates default assignment operators (‘`operator =`’), copy constructors (‘`X::X(X&)`’), and default constructors (‘`X::X()`’) whenever they are required.
- The new draft ANSI standard keyword `mutable` is supported.
- ‘`-fansi-overloading`’ is the default, to comply better with the ARM (at some cost in compatibility to earlier versions of GNU C++).
- More informative error messages.
- System include files are automatically treated as if they were wrapped in ‘`extern "C" { }`’.
- The new option ‘`-falt-external-templates`’ provides alternate template instantiation semantics.
- Operator declarations are now checked more strictly.
- You can now use template type arguments in the template parameter list.
- You can call the destructor for any type.
- The compiler source code is better organized.
- You can specify where to instantiate template definitions explicitly.

Much of the work in Phase 1.3 went to elimination of known bugs, as well as the major items above.

During the span of Phase 1.3, there were also two changes associated with the compiler that, while not specifically part of the C++ Renovation project, may be of interest:

- `gcov`, a code coverage tool for GNU CC, is now available from Cygnus Support. (`gcov` is free software, but the FSF has not yet accepted it.) See section “`gcov`: a Test Coverage Program” in *Using GNU CC*, for more information (in Cygnus releases of that manual).
- GNU C++ now supports *signatures*, a language extension to provide more flexibility in abstract type definitions. See section “Type Abstraction using Signatures” in *Using GNU CC*.

2.2 Major Changes

This release includes four wholesale rewrites of certain areas of compiler functionality:

1. Argument matching. GNU C++ is more compliant with the rules described in Chapter 13, "Overloading", of the ARM. This behavior is the default, though you can specify it explicitly with '`-fansi-overloading`'. For compatibility with earlier releases of GNU C++, specify '`-fno-ansi-overloading`'; this makes the compiler behave as it used to with respect to argument matching and name overloading.
2. Default constructors/destructors. Section 12.8 of the ARM, "Copying Class Objects", and Section 12.1, "Constructors", state that a compiler must declare such default functions if the user does not specify them. GNU C++ now declares, and generates when necessary, the defaults for constructors and destructors you might omit. In particular, assignment operators ('`operator =`') behave the same way whether you define them, or whether the compiler generates them by default; taking the address of the default '`operator =`' is now guaranteed to work. Default copy constructors ('`X::X(X&)`') now function correctly, rather than calling the copy assignment operator for the base class. Finally, constructors ('`X::X()`'), as well as assignment operators and copy constructors, are now available whenever they are required.
3. Binary incompatibility. There are no new binary incompatibilities in Phase 1.3, but Phase 1.2 introduced two binary incompatibilities with earlier releases. First, the functionality of '`operator new`' and '`operator delete`' changed. Name encoding ("mangling") of virtual table names changed as well. Libraries built with versions of the compiler earlier than Phase 1.2 must be compiled with the new compiler. (This includes the Cygnus Q2 progressive release and the FSF 2.4.5 release.)
4. New `g++` driver. A new binary `g++` compiler driver replaces the shell script. The new driver executes faster.

2.3 New features

- The compiler warns when a class contains only private constructors or destructors, and has no friends. At the request of some of our customers, we have added a new option, '`-Wctor-dtor-privacy`' (on by default), and its negation, '`-Wno-ctor-dtor-privacy`', to control the emission of this warning. If, for example, you are working towards making your code compile warning-free, you can use '`-Wall -Wno-ctor-dtor-privacy`' to find the most common warnings.
- There is now a mechanism which controls exactly when templates are expanded, so that you can reduce memory usage and program size and also instantiate them exactly once. You can control this mechanism with the option '`-fexternal-templates`' and its corresponding negation '`-fno-external-templates`'. Without this feature, space consumed by template instantiations can grow unacceptably in large-scale projects with many different source files. The default is '`-fno-external-templates`'.

  You do not need to use the '`-fexternal-templates`' option when compiling a file that does not define and instantiate templates used in other files, even if those files *are* compiled with '`-fexternal-templates`'. The only side effect is an increase in object size for each file that was compiled without '`-fexternal-templates`'.

When your code is compiled with '`-fexternal-templates`', all template instantiations are external; this requires that the templates be under the control of '`#pragma interface`' and '`#pragma implementation`'. All instantiations that will be needed should be in the implementation file; you can do this with a `typedef` that references the instantiation needed. Conversely, when you compile using the option '`-fno-external-templates`', all template instantiations are explicitly internal.

'`-fexternal-templates`' also allows you to finally separate class template function definitions from their declarations, thus speeding up compilation times for every file that includes the template declaration. Now you can have tens or even hundreds of lines in template declarations, and thousands or tens of thousands of lines in template definitions, with the definitions only going through the compiler once instead of once for each source file. It is important to note that you must remember to externally instantiate *all* templates that are used from template declarations in interface files. If you forget to do this, unresolved externals will occur.

In the example below, the object file generated ('`example.o`') will contain the global instantiation for '`Stack<int>`'. If other types of '`Stack`' are needed, they can be added to '`example.cc`' or placed in a new file, in the same spirit as '`example.cc`'.

`foo.h`:

```
#pragma interface "foo.h"
template<class T>
class Stack {
  static int statc;
  static T statc2;
  Stack() { }
  virtual ~Stack() { }
  int bar();
};
```

`example.cc`:

```
#pragma implementation "foo.h"
#include "foo.h"

typedef Stack<int> t;
int Stack<int>::statc;
int Stack<int>::statc2;
int Stack<int>::bar() { }
```

Note that using '`-fexternal-templates`' does not reduce memory usage from completely different instantiations ('`Stack<Name>`' vs. '`Stack<Net_Connection>`'), but only collapses different occurrences of '`Stack<Name>`' so that only one '`Stack<Name>`' is generated.

'`-falt-external-templates`' selects a slight variation in the semantics described above (incidentally, you need not specify both options; '`-falt-external-templates`' implies '`-fexternal-templates`').

With '`-fexternal-templates`', the compiler emits a definition in the implementation file that includes the header definition, *even if* instantiation is triggered from a *different* implementation file (e.g. with a template that uses another template).

With '`-falt-external-templates`', the definition always goes in the implementation file that triggers instantiation.

For instance, with these two header files—

'`a.h`':

```
#pragma interface
template <class T> class A { ... };
```

'`b.h`':

```
#pragma interface
class B { ... };
void f (A<B>);
```

Under '`-fexternal-templates`', the definition of '`A<B>`' ends up in the implementation file that includes '`a.h`'. Under '`-falt-external-templates`', the same definition ends up in the implementation file that includes '`b.h`'.

- You can control explicitly where a template is instantiated, without having to *use* the template to get an instantiation.

  To instantiate a class template explicitly, write '`template class` *name*`<paramvals>`', where *paramvals* is a list of values for the template parameters. For example, you might write

  ```
  template class A<int>
  ```

  Similarly, to instantiate a function template explicitly, write '`template` *fnsign*' where *fnsign* is the particular function signature you need. For example, you might write

  ```
  template void foo (int, int)
  ```

  This syntax for explicit template instantiation agrees with recent extensions to the draft ANSI standard.

- The compiler's actions on ANSI-related warnings and errors have been further enhanced. The '`-pedantic-errors`' option produces error messages in a number of new situations: using `return` in a non-`void` function (one returning a value); declaring a local variable that shadows a parameter (e.g., the function takes an argument '`a`', and has a local variable '`a`'); and use of the '`asm`' keyword. Finally, the compiler by default now issues a warning when converting from an `int` to an enumerated type. This is likely to cause many new warnings in code that hadn't triggered them before. For example, when you compile this code,

  ```
  enum boolean { false, true };
  void
  f ()
  {
    boolean x;

    x = 1; //assigning an int to an enum now triggers a warning
  }
  ```

  you should see the warning "`anachronistic conversion from integer type to enumeral type 'boolean'`". Instead of assigning the value 1, assign the original enumerated value '`true`'.

2.4 Enhancements and bug fixes

- You can now use nested types in a template parameter list, even if the nested type is defined within the same class that attempts to use the template. For example, given a template `list`, the following now works:

```
struct glyph {
  ...
  struct stroke { ... };
  list<stroke> l;
  ...
}
```

- Function pointers now work in template parameter lists. For example, you might want to instantiate a parameterized `list` class in terms of a pointer to a function like this:

```
list<int (*)(int, void *)> fnlist;
```

- Nested types are now handled correctly. In particular, there is no longer a limit to how deeply you can nest type definitions.
- GNU C++ now conforms to the specifications in Chapter 11 of the ARM, "Member Access Control".
- The ANSI C++ committee has introduced a new keyword `mutable`. GNU C++ supports it. Use `mutable` to specify that some particular members of a `const` class are *not* constant. For example, you can use this to include a cache in a data structure that otherwise represents a read-only database.
- Error messages now explicitly specify the declaration, type, or expression that contains an error.
- To avoid copying and editing all system include files during GNU C++ installation, the compiler now automatically recognizes system include files as C language definitions, as if they were wrapped in '`extern "C" { ... }`'.
- The compiler checks operator declarations more strictly. For example, you may no longer declare an '`operator +`' with three arguments.
- You can now use template type arguments in the same template parameter list where the type argument is specified (as well as in the template body). For example, you may write

```
template <class T, T t> class A { ... };
```

- Destructors are now available for all types, even built-in ones; for example, you can call '`int::~int`'. (Destructors for types like `int` do not actually do anything, but their existence provides a level of generality that permits smooth template expansion in more cases.)
- Enumerated types declared inside a class are now handled correctly.
- An argument list for a function may not use an initializer list for its default value. For example, '`void foo ( T x = { 1, 2 } )`' is not permitted.
- A significant amount of work went into improving the ability of the compiler to act accurately on multiple inheritance and virtual functions. Virtual function dispatch has been enhanced as well.

- The warning concerning a virtual inheritance environment with a non-virtual destructor has been disabled, since it is not clear that such a warning is warranted.
- Until exception handling is fully implemented in the Reno-2 release, use of the identifiers '`catch`', '`throw`', or '`try`' results in the warning:

```
t.C:1: warning: 'catch', 'throw', and 'try'
       are all C++ reserved words
```

- When giving a warning or error concerning initialization of a member in a class, the compiler gives the name of the member if it has one.
- Detecting friendship between classes is more accurately checked.
- The syntaxes of '`#pragma implementation "file.h"`' and '`#pragma interface`' are now more strictly controlled. The compiler notices (and warns) when any text follows '`file.h`' in the implementation pragma, or follows the word '`interface`'. Any such text is otherwise ignored.
- Trying to declare a template on a variable or type is now considered an error, not an unimplemented feature.
- When an error occurs involving a template, the compiler attempts to tell you at which point of instantiation the error occurred, in addition to noting the line in the template declaration which had the actual error.
- The symbol names for function templates in the resulting assembly file are now encoded according to the arguments, rather than just being emitted as, for example, two definitions of a function '`foo`'.
- Template member functions that are declared `static` no longer receive a `this` pointer.
- Case labels are no longer allowed to have commas to make up their expressions.
- Warnings concerning the shift count of a left or right shift now tell you if it was a '`left`' or '`right`' shift.
- The compiler now warns when a decimal constant is so large that it becomes `unsigned`.
- Union initializers which are raw constructors are now handled properly.
- The compiler no longer gives incorrect errors when initializing a union with an empty initializer list.
- Anonymous unions are now correctly used when nested inside a class.
- Anonymous unions declared as static class members are now handled properly.
- The compiler now notices when a field in a class is declared both as a type and a non-type.
- The compiler now warns when a user-defined function shadows a built-in function, rather than emitting an error.
- A conflict between two function declarations now produces an error regardless of their language context.
- Duplicate definitions of variables with '`extern "C"`' linkage are no longer considered in error. (Note in C++ linkage—the default—you may not have more than one definition of a variable.)
- Referencing a label that is not defined in any function is now an error.

- The syntax for pointers to methods has been improved; there are still some minor bugs, but a number of cases should now be accepted by the compiler.
- In error messages, arguments are now numbered starting at 1, instead of 0. Therefore, in the function '`void foo (int a, int b)`', the argument '`a`' is argument 1, and '`b`' is argument 2. There is no longer an argument 0.
- The tag for an enumerator, rather than its value, used as a default argument is now shown in all error messages. For example, '`void foo (enum x (= true))`' is shown instead of '`void foo (enum x (= 1))`'.
- The '`__asm__`' keyword is now accepted by the C++ front-end.
- Expressions of the form '`foo->~Class()`' are now handled properly.
- The compiler now gives better warnings for situations which result in integer overflows (e.g., in storage sizes, enumerators, unary expressions, etc).
- `unsigned` bitfields are now promoted to `signed int` if the field isn't as wide as an `int`.
- Declaration and usage of prefix and postfix '`operator ++`' and '`operator --`' are now handled correctly. For example,

```
class foo
{
public:
  operator ++ ();
  operator ++ (int);
  operator -- ();
  operator -- (int);
};

void
f (foo *f)
{
  f++;          // call f->operator++(int)
  ++f;          // call f->operator++()
  f--;          // call f->operator++(int)
  --f;          // call f->operator++()
}
```

- In accordance with ARM section 10.1.1, ambiguities and dominance are now handled properly. The rules described in section 10.1.1 are now fully implemented.

2.5 Problems with debugging

Two problems remain with regard to debugging:

- Debugging of anonymous structures on the IBM RS/6000 host is incorrect.
- Symbol table size is overly large due to redundant symbol information; this can make `gdb` coredump under certain circumstances. This problem is not host-specific.

3 Plans for Reno-2

The overall goal for the second phase of the GNU C++ Renovation Project is to bring GNU C++ to a new level of reliability, quality, and competitiveness. As particular elements of this strategy, we intend to:

0. Fully implement ANSI exception handling.
1. With the exception handling, add Runtime Type Identification (RTTI), if the ANSI committee adopts it into the standard.
2. Bring the compiler into closer compliance with the ARM and the draft ANSI standard, and document what points in the ARM we do not yet comply, or agree, with.
3. Add further support for the DWARF debugging format.
4. Finish the work to make the compiler compliant with ARM Section 12.6.2, initializing base classes in declaration order, rather than in the order that you specify them in a *mem-initializer* list.
5. Perform a full coverage analysis on the compiler, and weed out unused code, for a gain in performance and a reduction in the size of the compiler.
6. Further improve the multiple inheritance implementation in the compiler to make it cleaner and more complete.

As always, we encourage you to make suggestions and ask questions about GNU C++ as a whole, so we can be sure that the end of this project will bring a compiler that everyone will find essential for C++ and will meet the needs of the world's C++ community.

4 The Template Implementation

The C++ template[1] facility, which effectively allows use of variables for types in declarations, is one of the newest features of the language.

GNU C++ is one of the first compilers to implement many of the template facilities currently defined by the ANSI committee.

Nevertheless, the template implementation is not yet complete. This chapter maps the current limitations of the GNU C++ template implementation.

4.1 Limitations for function and class templates

These limitations apply to any use of templates (function templates or class templates) with GNU C++:

Template definitions must be visible
: When you compile code with templates, the template definitions must come first (before the compiler needs to expand them), and template definitions you use must be visible in the current scope.

Individual initializers needed for static data
: Templates for static data in template classes do not work. See Section 4.3 [Limitations for class templates], page 14.

4.2 Limitations for function templates

Function templates are implemented for the most part. The compiler can correctly determine template parameter values, and will delay instantiation of a function that uses templates until the requisite type information is available.

The following limitations remain:

- Narrowed specification: function declarations should not prevent template expansion. When you declare a function, GNU C++ interprets the declaration as an indication that you will provide a definition for that function. Therefore, GNU C++ does not use a template expansion if there is also an applicable declaration. GNU C++ only expands the template when there is no such declaration.

  The specification in Bjarne Stroustrup's *The C++ Programming Language, Second Edition* is narrower, and the GNU C++ implementation is now clearly incorrect. With this new specification, a declaration that corresponds to an instantiation of a function template only affects whether conversions are needed to use that version of the function. It should no longer prevent expansion of the template definition.

  For example, this code fragment must be treated differently:

```
template <class X> X min (X& x1, X& x2) { ... }
int min (int, int);
...
int i; short s;
```

[1] Class templates are also known as *parameterized types*.

```
min (i, s); // should call min(int,int)
            // derived from template
...
```

- The compiler does not yet understand function signatures where types are nested within template parameters. For example, a function like the following produces a syntax error on the closing ')' of the definition of the function `f`:

```
template <class T> class A { public: T x; class Y {}; };
template <class X> int f (A<X>::Y y) { ... }
```

- If you declare an `inline` function using templates, the compiler can only inline the code *after* the first time you use that function with whatever particular type signature the template was instantiated.

  Removing this limitation is akin to supporting nested function definitions in GNU C++; the limitation will probably remain until the more general problem of nested functions is solved.

- All the *method* templates (templates for member functions) for a class must be visible to the compiler when the class template is instantiated.

4.3 Limitations for class templates

Unfortunately, individual initializations of this sort are likely to be considered errors eventually; since they're needed now, you might want to flag places where you use them with comments to mark the need for a future transition.

- Member functions in template classes may not have results of nested type; GNU C++ signals a syntax error on the attempt. The following example illustrates this problem with an `enum` type `alph`:

```
template <class T> class list {
  ...
  enum alph {a,b,c};
  alph bar();
  ...
};

template <class T>
list<int>::alph list<int>::bar()  // Syntax error here
{
...
}
```

- A parsing bug makes it difficult to use preprocessor conditionals within templates. For example, in this code:

```
template <class T>
class list {
  ...
#ifdef SYSWRONG
  T x;
#endif
```

```
  ...
}
```

The preprocessor output leaves sourcefile line number information (lines like '`# 6 "foo.cc"`' when it expands the `#ifdef` block. These lines confuse the compiler while parsing templates, giving a syntax error.

If you cannot avoid preprocessor conditionals in templates, you can suppress the line number information using the '`-P`' preprocessor option (but this will make debugging more difficult), by compiling the affected modules like this:

```
g++ -P foo.cc -o foo
```

- Parsing errors are reported when templates are first *instantiated*—not on the template definition itself. In particular, if you do not instantiate a template definition at all, the compiler never reports any parsing errors that may be in the template definition.

4.4 Debugging information for templates

Debugging information for templates works for some object code formats, but not others. It works for stabs[2] (used primarily in A.OUT object code, but also in the Solaris 2 version of ELF), and the MIPS version of COFF debugging format.

DWARF support is currently minimal, and requires further development.

[2] Except that insufficient debugging information for methods of template classes is generated in stabs.

5 GNU C++ Conformance to ANSI C++

These changes in the GNU C++ compiler were made to comply more closely with the ANSI base document, *The Annotated C++ Reference Manual* (the ARM). Further reducing the divergences from ANSI C++ is a continued goal of the GNU C++ Renovation Project.

Section 3.4, *Start and Termination*. It is now invalid to take the address of the function '`main()`'.

Section 4.8, *Pointers to Members*. The compiler produces an error for trying to convert between a pointer to a member and the type '`void *`'.

Section 5.2.5, *Increment and Decrement*. It is an error to use the increment and decrement operators on an enumerated type.

Section 5.3.2, *Sizeof*. Doing `sizeof` on a function is now an error.

Section 5.3.4, *Delete*. The syntax of a *cast-expression* is now more strictly controlled.

Section 7.1.1, *Storage Class Specifiers*. Using the `static` and `extern` specifiers can now only be applied to names of objects, functions, and anonymous unions.

Section 7.1.1, *Storage Class Specifiers*. The compiler no longer complains about taking the address of a variable which has been declared to have `register` storage.

Section 7.1.2, *Function Specifiers*. The compiler produces an error when the `inline` or `virtual` specifiers are used on anything other than a function.

Section 8.3, *Function Definitions*. It is now an error to shadow a parameter name with a local variable; in the past, the compiler only gave a warning in such a situation.

Section 8.4.1, *Aggregates*. The rules concerning declaration of an aggregate are now all checked in the GNU C++ compiler; they include having no private or protected members and no base classes.

Section 8.4.3, *References*. Declaring an array of references is now forbidden. Initializing a reference with an initializer list is also considered an error.

Section 9.5, *Unions*. Global anonymous unions must be declared `static`.

Section 11.4, *Friends*. Declaring a member to be a friend of a type that has not yet been defined is an error.

Section 12.1, *Constructors*. The compiler generates a default copy constructor for a class if no constructor has been declared.

Section 12.6.2, *Special Member Functions*. When using a *mem-initializer* list, the compiler will now initialize class members in declaration order, not in the order in which you specify them. Also, the compiler enforces the rule that non-static `const` and reference members must be initialized with a *mem-initializer* list when their class does not have a constructor.

Section 12.8, *Copying Class Objects*. The compiler generates default copy constructors correctly, and supplies default assignment operators compatible with user-defined ones.

Section 13.4, *Overloaded Operators*. An overloaded operator may no longer have default arguments.

Section 13.4.4, *Function Call*. An overloaded '`operator ()`' must be a non-static member function.

Section 13.4.5, *Subscripting*. An overloaded '`operator []`' must be a non-static member function.

Section 13.4.6, *Class Member Access*. An overloaded '`operator ->`' must be a non-static member function.

Section 13.4.7, *Increment and Decrement*. The compiler will now make sure a postfix '`operator ++`' or '`operator --`' has an `int` as its second argument.

6 Name Encoding in GNU C++

In order to support its strong typing rules and the ability to provide function overloading, the C++ programming language *encodes* information about functions and objects, so that conflicts across object files can be detected during linking.[1] These rules tend to be unique to each individual implementation of C++.

The scheme detailed in the commentary for 7.2.1 of *The Annotated Reference Manual* offers a description of a possible implementation which happens to closely resemble the `cfront` compiler. The design used in GNU C++ differs from this model in a number of ways:

- In addition to the basic types `void`, `char`, `short`, `int`, `long`, `float`, `double`, and `long double`, GNU C++ supports two additional types: `wchar_t`, the wide character type, and `long long` (if the host supports it). The encodings for these are '`w`' and '`x`' respectively.
- According to the ARM, qualified names (e.g., '`foo::bar::baz`') are encoded with a leading '`Q`'. Followed by the number of qualifications (in this case, three) and the respective names, this might be encoded as '`Q33foo3bar3baz`'. GNU C++ adds a leading underscore to the list, producing '`_Q33foo3bar3baz`'.
- The operator '`*=`' is encoded as '`__aml`', not '`__amu`', to match the normal '`*`' operator, which is encoded as '`__ml`'.
- In addition to the normal operators, GNU C++ also offers the minimum and maximum operators '`>?`' and '`<?`', encoded as '`__mx`' and '`__mn`', and the conditional operator '`?:`', encoded as '`__cn`'.
- Constructors are encoded as simply '`__`*name*', where *name* is the encoded name (e.g., `3foo` for the `foo` class constructor). Destructors are encoded as two leading underscores separated by either a period or a dollar sign, depending on the capabilities of the local host, followed by the encoded name. For example, the destructor '`foo::~foo`' is encoded as '`_$_3foo`'.
- Virtual tables are encoded with a prefix of '`_vt`', rather than '`__vtbl`'. The names of their classes are separated by dollar signs (or periods), and not encoded as normal: the virtual table for `foo` is '`__vt$foo`', and the table for `foo::bar` is named '`__vt$foo$bar`'.
- Static members are encoded as a leading underscore, followed by the encoded name of the class in which they appear, a separating dollar sign or period, and finally the unencoded name of the variable. For example, if the class `foo` contains a static member '`bar`', its encoding would be '`_3foo$bar`'.
- GNU C++ is not as aggressive as other compilers when it comes to always generating '`Fv`' for functions with no arguments. In particular, the compiler does not add the sequence to conversion operators. The function '`foo::bar()`' is encoded as '`bar__3foo`', not '`bar__3fooFv`'.
- The argument list for methods is not prefixed by a leading '`F`'; it is considered implied.
- GNU C++ approaches the task of saving space in encodings differently from that noted in the ARM. It does use the '`T`*n*' and '`N`*xy*' codes to signify copying the *n*th argument's

[1] This encoding is also sometimes called, whimsically enough, *mangling*; the corresponding decoding is sometimes called *demangling*.

type, and making the next x arguments be the type of the yth argument, respectively. However, the values for n and y begin at zero with GNU C++, whereas the ARM describes them as starting at one. For the function '`foo (bartype, bartype)`', GNU C++ uses '`foo__7bartypeT0`', while compilers following the ARM example generate '`foo__7bartypeT1`'.

- GNU C++ does not bother using the space-saving methods for types whose encoding is a single character (like an integer, encoded as '`i`'). This is useful in the most common cases (two `int`s would result in using three letters, instead of just '`ii`').

Table of Contents