Green Hills C++

User's Guide

• SOFTWARE, INC. •

Version 1.8.9

DISCLAIMER

GREEN HILLS SOFTWARE, INC. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Green Hills Software, Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Green Hills Software, Inc. to notify any person of such revision or changes.

Revision History

Revision	Release Date	Location of Revision(s)

PubID: L01B-C0499-89NG

CONTENTS

CONTENTS

.

PREFACE

ABOUT THIS MANUAL

This manual explains the Green Hills C++ language. Green Hills system-specific *Development Guide* provides details on using the compiler. It is a primary reference guide, providing language-specific information for the programmer. It assumes familiarity with commonly used software terminology, plus relevant programming languages and operating systems.

The platform for all examples is a Sun workstation running a UNIX environment. Differences on other systems are mentioned, where applicable.

The explanations and examples in this manual assume the Green Hills products are installed in the directory **/usr/green**. If this is not the case, substitute the correct directory. The C++ compiler driver is **gcx** in this manual. If this is not the case for your release, substitute the correct driver name.

TYPOGRAPHICAL CONVENTIONS

Convention	Example	Description
bold text	-noansi	name of program, command, directory, or file
bold characters in quotes	"**A**"	name to enter as shown, without quotes
courier	setenv TMPDIR	samples of code, or instructions to enter
italic text in a command line	**-o** *filename*	place-holder for user-supplied information
square brackets, []	**.macro** *name* [*list*]	encloses optional commands or terms
square brackets [] around boldface **default**	Specifies **char** as signed [**default**].	command or option is the default

For example, in the command description

gcx [*-cpu=processor*] *filename*

the command **gcx** should be entered as given, the *-cpu=processor* is optional with the appropriate CPU option replacing *processor*, and the appropriate file name replacing the word *filename*.

Chapter

1

C++ Language Features

This chapter provides information on the C++ compiler and language-specific C++ issues.

INTRODUCTION

The Green Hills Software C++ compiler accepts several dialects of C++ - ANSI Standard C++ (very close to full compliance): EC++ (Embedded C++), ESTL (Embedded C++ with templates and namespaces), Cfront 3.0, Cfront 2.1, and ARM compliant C++. The default dialect is Standard C++.

In ARM mode, the Green Hills C++ compiler accepts the C++ language as defined by *The Annotated C++ Reference Manual (ARM)* by Ellis and Stroustrup, Addison-Wesley, 1990, including templates, exceptions, and the anachronisms of Chapter 18. This is essentially the same language defined by the language reference for Cfront version 3.0.x, with the addition of exceptions.

The Green Hills C++ compiler also has a Cfront compatibility mode, which duplicates a number of "features" and bugs of Cfront. Complete compatibility is not guaranteed or intended; the mode allows programmers who have unwittingly used Cfront features to continue to compile existing code. Other options enable and disable anachronisms and strict standard-conformance checking.

The following features, not in the ARM, but in the J16/WG21 Working Paper are accepted:

- The dependent statement **if**, **while**, **do**-**while**, or **for** is a scope, and the restriction on such a statement's being a declaration is removed.
- The expression tested in an **if**, **while**, **do**-**while**, or **for**, as the first operand of a "**?**" operator, or as an operand of the "**&&**", "||", or "**!**" operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.
- Use of a global-scope qualifier in member references of the form **x.::A::B** and **p->::A::B**.
- The precedence of the third operand of the "**?**" operator is changed.
- If control reaches the end of the **main()** routine, and **main()** has an integral return type, it is treated as if a **return 0;** statement were executed.

- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.
- A functional-notation cast of the form **A()** can be used even if **A** is a class without a (nontrivial) constructor. The temporary created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have non-type template parameters.
- A reference to **const volatile** cannot be bound to an rvalue.
- Qualification conversions, such as conversion from **T**** to **T const * const *** are allowed.
- Digraphs are recognized.
- Operator keywords (e.g., **and**, **bitand**, etc.) are recognized.
- Static data member declarations can be used to declare member constants.
- **wchar_t** is recognized as a keyword and a distinct type.
- **bool** is recognized.
- RTTI (runtime type identification), including **dynamic_cast** and the **typeid** operator, is implemented.
- Declarations in tested conditions (in **if**, **switch**, **for**, and **while** statements) are supported.
- Array new and delete are implemented.
- New-style casts (**static_cast**, **reinterpret_cast**, and **const_cast**) are implemented.
- Definition of a nested class outside its enclosing class is allowed.
- **mutable** is accepted on non-static data member declarations.
- Namespaces are implemented, including **using** declarations and directives. Access declarations are broadened to match the corresponding **using** declarations.
- Explicit instantiation of templates is implemented.
- The `typename` keyword is recognized.
- `explicit` is accepted to declare non-converting constructors.

- The scope of a variable declared in the `for-init-statement` for a `for` loop is the scope of the loop (not the surrounding scope).
- Member templates are implemented.
- The new specialization syntax (using "`template <>`") is implemented.
- Cv-qualifiers are retained on rvalues (in particular, on function return values).
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between PODs and non-PODs with trivial constructors.
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions).
- `extern inline` functions are supported, and the default linkage for inline functions is external.
- A typedef name may be used in an explicit destructor call.
- Placement delete is implemented.
- An array allocated via a placement new can be deallocated via delete.
- Covariant return types on overriding virtual functions are supported.
- `enum` types are considered to be non-integral types.
- Partial specialization of class templates is implemented.
- Partial ordering of function templates is implemented.
- Function declarations that match a function template are regarded as independent functions, not as "guiding declarations" that are instances of the template.
- It is possible to overload operators using functions that take enum types and no class types.
- Explicit specification of function template arguments is supported.
- Unnamed template parameters are supported.
- The new lookup rules for member references of the form **x.A::B** and **p->A::B** are supported.
- The notation **:: template** (and **->template**, etc.) is supported.

In ANSI Standard C++ mode, the compiler accepts the full ANSI Standard C++ language with the exception of:

- `enum` types cannot contain values larger than can be contained in an `int`.

- `reinterpret_cast` does not allow casting a pointer to member of one class to a pointer to member of another class if the classes are unrelated.
- Two-phase name binding in templates, as described in [**temp.res**] and [**temp.dep**] of the Working Paper, is not implemented.
- In a reference of the form `f()->g()`, with a `g` a static member function, `f()` is not evaluated. This is as required by the ARM. The WP, however, requires that `f()` be evaluated.
- Class name injection is not implemented.
- Putting a `try/catch` around the initializers and body of a constructor is not implemented.
- Template template parameters are not implemented.
- Koenig lookup of function names on all calls is not implemented.
- Finding friend functions of the argument class types on name lookup on the function name in calls is not implemented.
- String literals do not have **const** type
- Universal character set escapes (e.g., **uabcd**) are not implemented.
- The **export** keyword for templates is not implemented.

We recommend Bjarne Stroustrup's *The C++ Programming Language, Third Edition* as a good reference for ANSI Standard C++.

ACCEPTED ANACHRONISMS

The following anachronisms are accepted when enabled:

- **overload**, in function declarations, is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array may be specified in an array **delete** operation. The value is ignored.
- A single **operator++()** and **operator--()** function can be used to overload both prefix and postfix operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.

- Assignment to **this** in constructors and destructors is allowed. This is allowed only if anachronisms are enabled and the **assignment to this** configuration parameter is enabled.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a non-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-const type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-const class type may be initialized from an rvalue of the class type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following declares the overloading of two functions foo:

  ```
  int foo(int);
  int foo(x) char x; {return x;}
  ```

  It will be noted that in C this code is legal but has a different meaning: a tentative declaration of **foo** is followed by its definition.
- A reference to a non-const class can be bound to a class rvalue of the same type or a derived type thereof.

  ```
  struct A {
      A(int);
      A operator=(A&);
      A operator+(const A&);
  };
  main() {
      A b(1);
      b = A(1) + A(2);  // Allowed as anachronism
  }
  ```

EXTENSIONS ACCEPTED IN NORMAL C++ MODE

The following extensions are accepted in all modes (except when strict ANSI violations are diagnosed as errors):

- A **friend** declaration for a class may omit the **class** keyword:

```
class B;
class A {
    friend B;  // Should be “friend class B”
};
```

- Constants of scalar type may be defined within classes:

```
class A {
    const int size = 10;
    int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A {
    int A::f();  // Should be int f();
};
```

- The preprocessing symbol **c_plusplus** is defined in addition to the standard **_ _cplusplus**.
- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a “default” assignment operator; that is, such a declaration blocks the implicit generation of a copy assignment operator. (This is Cfront behavior that is known to be relied upon in at least one widely used library.) For example:

```
struct A {};
struct B : public A {
     B& operator=(A&);
};
```

By default, as well as in Cfront-compatibility mode, there will be no implicit declaration of **B::operator=(const B&)**, whereas in strict ANSI mode **B::operator=(A&)** is not a copy assignment operator and **B::operator=(const B&)** is implicitly declared.

- Implicit type conversion between a point to an extern "C" function and a pointer to an extern "C++" function is permitted. Here's an example:

```
extern "C" void f();  // f's type has extern "C" linkage
void (*pf)()          // pf points to an extern "C++" function
            = &f;    // error unless implicit conversion is allowed
```

EXTENSIONS ACCEPTED IN CFRONT 2.1 COMPATIBILITY MODE

The following extensions are accepted in Cfront 2.1 compatibility mode in addition to the extensions listed in the 2.1/3.0 section following (i.e., these are things that were corrected in the 3.0 release of Cfront):

- The dependent statement of an **if**, **while**, **do-while**, or **for** is not considered to define a scope. The dependent statement may not be a declaration. Any objects constructed within the dependent statement are destroyed at exit from the dependent statement.
- Implicit conversion from integral types to enumeration types is allowed.
- A non-**const** member function may be called for a **const** object. A warning is issued.
- A **const void *** value may be implicitly converted to a **void *** value, e.g., when passed as an argument.
- When, in determining the level of argument match for overloading, a reference parameter is initialized from an argument that requires a non-class standard conversion, the conversion counts as a user-defined conversion. (This is an outright bug, which unfortunately happens to be exploited in the NIH class libraries).
- When a builtin operator is considered alongside overloaded operators in overload resolution, the match of an operand of a builtin type against the builtin type required by the builtin operator is considered a standard conversion in all cases (e.g., even when the type is exactly right without conversion).
- A reference to a non-**const** type may be initialized from a value that is a **const**-qualified version of the same type, but only if the value is the result of selecting a member from a **const** class object or a pointer to a such an object.

- A cast to an array type is allowed; it is treated like a cast to a pointer to the array element type. A warning is issued.
- When an array is selected from a class, the type qualifiers on the class object (if any) are not preserved in the selected array. (In the normal mode, any type qualifiers on the object are preserved in the element type of the resultant array.)
- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.
- A value may be supplied on the return statement in a function with a **void** return type. A warning is issued.
- A parameter of type **const void *** is allowed on **operator delete**; it is treated as equivalent to **void ***.
- A period "**.**" may be used for qualification where "**::**" should be used. Only "**::**" maybe be used as a global qualifier. Except for the global qualifier, the two kinds of qualifier operators may not be mixed in a given name (i.e., you may say **A::B::C** or **A.B.C** but not **A::B.C** or **A.B::C**). A period may not be used in a vacuous destructor reference nor in a qualifier that follows a template reference such as **A<T>::B**.
- Cfront 2.1 does not correctly look up names in friend functions that are inside class definitions. In this example, function **f** should refer to the functions and variables (e.g., **f1** and **a1**) from the class declaration. Instead, the global definitions are used.

```
int a1;
int e1;
void f1();
class A {
    int a1;
    void f1();
    friend void f()
    {
    int i1 = a1;  // cfront uses global a1
    f1();         // cfront uses global f1
    }
};
```

Only the innermost class scope is (incorrectly) skipped by Cfront as illustrated in the following example:

```
int a1;
int b1;
struct A {
  static int a1;
  class B {
    static int b1;
    friend void f()
    {
      int i1 = a1;  // cfront uses A::a1
      int j1 = b1;  // cfront uses global b1
    }
  };
};
```

- **operator=** may be declared as a nonmember function. (This is flagged as an anachronism by Cfront 2.1)
- A type qualifier is allowed (but ignored) on the declaration of a constructor or destructor. For example:

```
Class A {
    A() const;  // No error in cfront 2.1 mode
};
```

CFRONT COMPATIBILITY MODE EXTENSIONS

The following extensions are accepted in both Cfront 2.1 and Cfront 3.0 compatibility mode (i.e., these are features or problems that exist in both cfront 2.1 and 3.0):

- Type qualifiers on the **this** parameter may be dropped in contexts such as this example:

```
struct A {
    void f() const;
};
void (A::*fp)() = &A::f;
```

This is actually a safe operation. A pointer to a **const** function may be put into a pointer to non-**const**, because a call using the pointer is permitted to modify the

object and the function pointed to will actually not modify the object. The opposite assignment would not be safe.

- Conversion operators specifying conversion to **void** are allowed.
- A nonstandard friend declaration may introduce a new type. A friend declaration that omits the elaborated type specifier is allowed in default mode, but in cfront mode the declaration is also allowed to introduce a new type name.

```
struct A {
    friend B;
};
```

- The third operator of the **?** operator is a conditional expression instead of an assignment expression as it is in the modern language.
- A reference to a pointer type may be initialized from a pointer value without use of a temporary even when the reference pointer type has additional type qualifiers above those present in the pointer value. For example:

```
int *p;
const int *&r = p;  // No temporary used
```

- A reference may be initialized with a null.
- Because cfront does not check the accessibility of types, access errors for types are issued as warnings instead of errors.
- When matching arguments of an overloaded function, a const variable with value zero is not considered to be a null pointer constant. In general, in overload resolution a null pointer constant must be spelled “0” to be considered a null pointer constant (e.g., ‘\0’ is not considered a null pointer constant).
- An alternate form of declaring pointer-to-member-function variables is supported, as follows:

```
struct A {
    void f(int);
    static void f(int);
    typedef void A::T3(int);  // non-std typedef  decl
    typedef void T2(int);     // std typedef
};
typedef void A::T(int);       // non-std typedef decl
```

```
T* pmf = &A::f;          // non-std ptr-to-member decl
A::T2* pf = A::sf;       // std ptr to static mem decl
A::T3* pmf2 = &A::f;     // non-std ptr-to-member decl
```

where **T** is construed to name a routine type for a non-static member function of class **A** that takes an **int** argument and returns **void**; the use of such types is restricted to nonstandard pointer-to-member declarations. The declarations of **T** and **pmf** in combination are equivalent to a single standard point-to-member declaration:

```
void (A::*pmf)(int) = &A::f;
```

A non-standard pointer-to-member declaration that appears outside of a class declaration, such as the declaration of **T**, is normally invalid and would cause an error to be issued. However, for declarations that appear within a class declaration, such as **A::T3**, this feature changes the meaning of a valid declaration. Cfront version 2.1 accepts declarations, such as **T**, even when **A** is an incomplete type; so this case is also excepted.

- Protected member access checking is not done when the address of a protected member is taken. For example:

```
class B {protected: int i;};
class D : public B {void mf();};
void D::mf() {
    int B::* pmi1 = &B::i; // error, OK in cfront mode
    int D::* pmi2 = &D::i; // OK
}
```

Note that protected member access checking for other operations (i.e., everything except taking a pointer-to-member address) is done in the normal manner.

- The destructor of a derived class may implicitly call the private destructor of a base class. In default mode this is an error but in cfront mode it is reduced to a warning. For example:

```
class A {
    ~A();
};
class B : public A {
    ~B();
};
```

```
B::~B(){}  // Error except in cfront mode
```

- When disambiguation requires deciding whether something is a parameter declaration or an argument expression, the pattern *type-name-or-keyword***(***identifier...***)** is treated as an argument. For example:

```
class A { A(); };
double d;
A x(int(d));
A(x2);
```

By default, **int(d)** is interpreted as a parameter declaration (with redundant parentheses), and so **x** is a function; but in cfront-compatibility mode **int(d)** is an argument and **x** is a variable.

The declaration **A(x2);** is also misinterpreted by cfront. It should be interpreted as the declaration of an object named **x2**, but in cfront mode is interpreted as a function style cast of **x2** to the type **A**.

Similarly, the declaration:

```
int xyz(int());
```

declares a function named **xyz**, that takes a parameter of type "function taking no arguments and returning an **int**." In cfront mode this is interpreted as a declaration of an object that is initialized with the value **int()** (which evaluates to zero).

- A named bit-field may have a size of zero. The declaration is treated as though no name had been declared.
- Plain bit fields (i.e., bit fields declared with a type of **int**) are always unsigned.
- The name given in an elaborated type specifier is permitted to be a **typedef** name that is the synonym for a class name. For example:

```
typedef class A T;
class T *pa;  // No error in cfront mode
```

- No warning is issued on duplicate size and sign specifiers.

```
short short int i; // No warning in cfront mode
```

- Virtual function table pointer update code is not generated in destructors for base classes of classes without virtual functions, even if the base class virtual functions might be overridden in a further-derived class. For example:

```
struct A {
  virtual void f() {}
  A() {}
  ~A() {}
};
struct B : public A {
  B() {}
  ~B() {f();} // Should call A::f according to ARM
              // 12.7
};
struct C : public B {
  void f() {}
} c;
```

In cfront compatibility mode, **B::~B** calls **C::f**.

- An extra comma is allowed after the last argument in an argument list. For example:

```
f(1, 2, );
```

- A constant pointer-to-member-function may be cast to a pointer-to-function. A warning is issued.

```
struct A {int f();};
main () {
  int (*p)();
  p = (int (*)())A::f;  // OK, with warning
}
```

- Arguments of class types that allow bitwise copy construction but also have destructors are passed by value (i.e., like C structures), and the destructor is

not called on the copy. In normal mode, the class object is copied into a temporary, the address of the temporary is passed as the argument, and the destructor is called on the temporary after the call returns. Note that because the argument is passed differently (by value instead of by address), code like this compiled in cfront mode is not calling-sequence compatible with the same code compiled in normal mode. In practice, this is not much of a problem, since classes that allow bitwise copying usually do not have destructors.

- A union member may be declared to have the type of a class for which the user has defined an assignment operator (as long as the class has no constructor or destructor). A warning is issued.
- When an unnamed class appears in a **typedef** declaration, the **typedef** name may appear as the class name in an elaborated type specifier.

```
typedef struct {int i, j;} S;
struct S x;  // No error in cfront mode
```

- Two member functions may be declared with the same parameter types when one is static and the other is non-static with a function qualifier.

```
class A {
  void f(int) const;
  static void f(int);  // No error in cfront mode
};
```

- The scope of a variable declared in the for-init-statement is the scope to which the for statement belongs.

```
int f(int i) {
  for (int j = 0; j < i; ++j) { /* ... */ }
  return j;        // No error in cfront mode
};
```

- Function types differing only in that one is declared extern “C” and the other extern “C++” can be treated as identical:

```
typedef void (*PF)();
extern “C” typedef void (*PCF)();
```

```
void f(PF);
void f(PCF);
```

PF and **PCF** are considered identical and **void f(PCF)** is treated as a compatible redeclaration of **f**.

In cfront-compatibility mode an implicit type conversion will always be done between a pointer to an extern “C” function and a pointer to an extern “C++” function.

- Functions declared inline have internal linkage.
- enum types are regarded as integral types.
- An uninitialized const object of non-POD class type is allowed even if its default constructor is implicitly declared:

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

- A function parameter type is allowed to involve a pointer or reference to array of unknown bounds.

TEMPLATE INSTANTIATION

The C++ language includes the concept of templates. A template is a description of a class or function that is a model for a family of related classes or functions. For example, one can write a template for a **Stack** class, and then use a stack of integers, a stack of floats, and a stack of some user-defined type. In the source, these might be written **Stack<int>**, **Stack<float>**, and **Stack<x>**. From a single source description of the template for a stack, the compiler can create instantiations of the template for each of the types required.

The instantiation of a class template is always done as soon as it is needed in a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (hereafter referred to as template entities) are not necessarily done immediately, for several reasons:

- You would like to end up with only one copy of each instantiated entity across all the object files that make up a program. (This of course applies to entities with external linkage.)

- The language allows you to write a specialization of a template entity, i.e., a specific version to be used in place of a version generated from the template for a specific data type. (You could, for example, write a version of **Stack<int>**, or of just **Stack<int>::push**, that replaces the template-generated version; often, such a specialization provides a more efficient representation for a particular data type.) Since the compiler cannot know, when compiling a reference to a template entity, if a specialization for that entity will be provided in another compilation, it cannot do the instantiation automatically in any source file that references it.
- The language also dictates that template functions that are not referenced should not be compiled, that, in fact, such functions might contain semantic errors that would prevent them from being compiled. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

Note that certain template entities are always instantiated when used, e.g., inline functions. Also, there is no support for the export directive.

From these requirements, one can see that if the compiler is responsible for doing all the instantiations automatically, it can only do so on a program-wide basis. That is, the compiler cannot make decisions about instantiation of template entities until it has seen all the source files that make up a complete program.

The Green Hills C++ compiler provides an instantiation mechanism that does automatic instantiation at link time. For cases where the programmer wants more explicit control over instantiation, the Green Hills C++ compiler also provides instantiation modes and instantiation pragmas, which can be used to exert fine-grained control over the instantiation process.

AUTOMATIC INSTANTIATION

The goal of an automatic instantiation mode is to provide painless instantiation. The programmer should be able to compile source files to object code, then link them and run the resulting program, and never have to worry about how the necessary instantiations get done.

In practice, this is hard for a compiler to do, and different compilers use different automatic instantiation schemes with different strengths and weaknesses:

Cfront saves information about each file it compiles in a special directory called the repository. It instantiates nothing during the normal compilations. At link time, it looks for entities that are referenced but not defined, and whose mangled names indicate that they are template entities. For each such entity, it consults the repository information to find the file containing the source for the entity, and it does a compilation of the source to generate an object file containing object code for that entity. This object code for instantiated objects is then combined with the "normal" object code in the link step.

The programmer using cfront must follow a particular coding convention: all templates must be declared in **.h** files, and for each such file there must be a corresponding **.C** file containing the associated definitions. The compiler is never told about the **.C** files explicitly; one does not, for example, compile them in the normal way. The link step looks for them when and if it needs them, and does so by taking the **.h** file name and replacing its suffix.

This scheme has the disadvantage that it does a separate compilation for each instantiated function (or, at best, one compilation for all the member functions of one class). Even though the function itself is often quite small, it must be compiled along with the declarations for the types on which the instantiation is based, and those declarations can easily run into many thousands of lines. For large systems, these compilations can take a very long time. The link step tries to be smart about recompiling instantiations only when necessary, but because it keeps no fine-grained dependency information, it is often forced to "recompile the world" for a minor change in a **.h** file. In addition, cfront has no way of ensuring that preprocessing symbols are set correctly when it does these instantiation compilations, if preprocessing symbols are set other than on the command line.

Borland's C++ compiler instantiates everything referenced in a compilation, then uses a special linker to remove duplicate definitions of instantiated functions.

The programmer using Borland's compiler must make sure that every compilation sees all the source code it needs to instantiate all the template entities referenced in that compilation. That is, you cannot refer to a template entity in a source file if a definition for that entity is not included by that source file. In practice, this means that either all the definition code is put directly in the **.h** files, or that each **.h** file includes an associated **.C** (actually, **.CPP**) file.

This scheme is straightforward, and works well for small programs. For large systems, however, it tends to produce very large object files, because each object file must contain object code (and symbolic debugging information) for each template entity it references.

The Green Hills C++ approach is a little different. It requires that for each instantiation required, there is some (normal, top-level, explicitly-compiled) source file that contains both the definition of the template entity and of any types required for the particular instantiation. This requirement can be met in various ways:

- The Borland convention: each **.h** file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.
- Implicit inclusion: when the compiler sees a template declaration in a **.h** file and discovers a need to instantiate that entity, it is given permission to go off looking for an associated definition file having the same base name and a different suffix, and it implicitly includes that file at the end of the compilation. This method allows most programs written using the cfront convention to be compiled with Green Hills C++. See the section on implicit inclusion.
- The ad hoc approach: the programmer makes sure that the files that define template entities also have the definitions of all the available types, and adds code or pragmas in those files to request instantiation of the entities there.

The Green Hills C++ automatic instantiation method works as follows:

1. The first time the source files of a program are compiled, no template entities are instantiated. However, template information files are generated and contain information about entities that could have been instantiated in each compilation. These template information files have a **.ti** suffix.
2. When the object files are linked together, a program called the prelinker is run. It examines the object files, looking for references and definitions of template entities, and for the added information about entities that could be instantiated.
3. If the prelinker finds a reference to a template entity for which there is no definition anywhere in the set of object files, it looks for a file that indicates that it could instantiate that template entity. When it finds such a file, it

assigns the instantiation to it. The set of instantiations assigned to a given file is recorded in an associated instantiation request file (with a **.ii** suffix).

4. The prelinker then executes the compiler again to recompile each file for which the **.ii** file was changed. The original compilation options (saved in the **.ti** file) are used for recompilation.
5. When the compiler compiles a file, it reads the **.ii** file for that file and obeys the instantiation requests therein. It produces a new object file containing the requested template entities (and all the other things that were already in the object file). The compiler also receives a definition list file, which lists all the instantiations for which definitions already exist in the set of object files. If during compilation the compiler has the opportunity to instantiate a referenced entity that is not on that list, it goes ahead and does the instantiation. It passes back to the prelinker (in the definition list file) a list of instantiations that it has "adopted" in this way, so the prelinker can assign them to a file. This adoption process allows rapid instantiation and assignment of instantiations referenced from new instantiations, and reduces the need to recompile a given file more than once during the prelinking process.
6. The prelinker repeats steps 3-5 until there are no more instantiations to be adjusted.
7. The object files are linked together.

Once the program has been linked correctly, the **.ii** files contain a complete set of instantiation assignments. From then on, whenever source files are recompiled, the compiler will consult the **.ii** files and do the indicated instantiations as it does the normal compilations. That means that, except in cases where the set of required instantiations changes, the prelink step from then on will find that all the necessary instantiations are present in the object files and no instantiations assignment adjustments need be done. That's true even if the entire program is recompiled.

If the programmer provides a specialization of a template entity somewhere in the program, the specialization will be seen as a definition by the prelinker. Since that definition satisfies whatever references there might be to that entity, the prelinker will see no need to request an instantiation of the entity. If the programmer adds a specialization to a program that has previously been compiled, the prelinker will notice that too and remove the assignment of the instantiation from the proper **.ii** file.

The **.ii** files should not, in general, require any manual intervention. One exception: if a definition is changed in such a way that some instantiation no longer compiles (it gets errors), and at the same time a specialization is added in another file, and the first file is being recompiled before the specialization file and is getting errors, the **.ii** file for the file getting the errors must be deleted manually to allow the prelinker to regenerate it.

If the prelinker changes an instantiation assignment, it will issue a message like:

```
C++ prelinker: A<int>::f() assigned to file test.o
C++ prelinker: executing: /usr/green/gcx -c test.c
```

The automatic instantiation scheme can coexist with partial explicit control of instantiation by the programmer through the use of pragmas or command-line specification of the instantiation mode. See the following sections for more information.

Instantiations are normally generated as part of the object file of the translation unit in which the instantiations are performed. But when **One instantiation per object file** is used, each instantiation is placed in its own object file. This mode is useful when building libraries that need to include copies of the instances referenced from the library. If each instance is not placed in its own object file, it may be impossible to link the library with another library containing some of the same instances.

Automatic instantiation may optionally be turned off. If automatic instantiation is turned off, the template information file is not generated.

INSTANTIATION MODES

Normally, when a file is compiled, no template entities are instantiated (except those assigned to the file by automatic instantiation). The overall instantiation mode can, however, be changed by the following command line options:

-tnone Do not automatically create instantiations of any template entities. This is the default. It is also usually the appropriate mode when automatic instantiation is done.

-tused Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions.

-tall Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Nonmember template functions will be instantiated even if the only reference was a declaration.

-tlocal Similar to **-tused** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables.) However, one may end up with many copies of the instantiated functions, so this is not suitable for production use. **-tlocal** can not be used in conjunction with automatic template instantiation. If automatic instantiation is enabled by default, it will be disabled by the **-tlocal** option. If automatic instantiation is not enabled by default, use of **-tlocal** and **-template=auto** is an error.

In the case where the compiler is given a single file to compile and link (e.g., **gcx albatross.C**), the compiler knows that all instantiations will have to be done in the single source file. Therefore, it uses the **-tused** mode and suppresses automatic instantiation.

INSTANTIATION #PRAGMA DIRECTIVES

Instantiation pragmae can be used to control the instantiation of specific template entities or sets of template entities. There are three instantiation pragmas:

instantiate Causes a specified entity to be instantiated.

do_not_instantiate
Suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition will be supplied.

can_instantiate
Indicates that a specified entity can be instantiated in the current compilation, but need not be. This is used in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity turns out to be required.

Each of the above instantiation pragmae take an argument, which may be one of the following:

- A template class name (e.g., **A<int>**)
- A template class declaration (e.g., **class A<int>**)
- A member function name (e.g., **A<int>::f**)
- A static data member name (e.g., **A<int>::i**)
- A static data declaration (e.g., **int A<int>::i**)
- A member function declaration (e.g., **void A<int>::f(int, char)**)
- A template function declaration (**char* f(int, float)**)

A pragma directive in which the argument is a template class name is equivalent to repeating the pragma for each member function and static data member declared in the class. When instantiating an entire class a given member function or static data member may be excluded using the **do_not_instantiate** pragma. For example:

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the **instantiate** pragma and no template definition is available or a specific definition is provided, and error is issued.

```
template <class T> void f1(T);  // No body provided
template <class T> void g1(T);  // No body provided
void f1(int) {}  // Specific definition
void main()
{
    int i;
    double d;
    f1(i);
    f1(d);
    g1(i);
    g1(d);
}
#pragma instantiate void f1(int)  // error - specific
                                  // definition
#pragma instantiate void g1(int)  // error - no body
                                  // provided
```

f1(double) and **g1(double)** will not be instantiated (because no bodies were supplied) but no errors will be produced during the compilation (if no bodies are supplied at link time, a linker error will be produced).

A member function name (e.g., **A<int>::f**) can only be used as a pragma argument if it refers to a single user defined member function (i.e., not an overloaded function). Compiler-generated functions are not considered, so a name may refer to a user defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration. For example:

```
#pragma instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation pragma may not be a compiler-generated function, an inline function, or a pure virtual function.

IMPLICIT INCLUSION

When implicit inclusion is enabled, the compiler is given permission to assume that if it needs a definition to instantiate a template entity declared in a **.h** file it can implicitly include the corresponding **.C** file to get the source code for the definition. For example, if a template entity **ABC::f** is declared in file **xyz.h**, and an instantiation of **ABC::f** is required in a compilation but no definition of **ABC::f** appears in the source code processed by the compilation, the compiler will look to see if a file **xyz.C** exists, and if so it will process it as if it were included at the end of the main source file.

To find the template definition file for a given template entity, the compiler needs to know the full path name of the file in which the template was declared and whether the file was included using the system include syntax (e.g., **#include <file.h>**). This information is not available for preprocessed source containing **#line** directives. Consequently, the compiler will not attempt implicit inclusion for source code containing **#line** directives.

The following suffixes will be searched for: **.c**, **.C**, **.cpp**, **.CPP**, **.cxx**, **.CXX**, and **.cc**.

Implicit inclusion works well alongside automatic instantiation, but the two are independent. They can be enabled or disabled independently, and implicit inclusion is still useful when automatic instantiation is not done.

USING CLEARMAKE WITH GREEN HILLS C++

The Green Hills C++ compiler offers several methods of building template code. The following sections describe these methods. This information is provided by ClearCase® of Atria, Inc.

AUTOMATIC INSTANTIATION

The Green Hills C++ compiler performs automatic template instantiation to build template code by default. During a prelink step, the compiler determines the necessary template code the program requires and compiles into some of the object files to make up the program. The compiler tracks how the program uses template code. It records this information in the **.ii** files in the build directory.

If using the Automatic Instantiation method, **clearmake** sometimes executes unnecessary rebuilds of program components. When the prelinker compiles template code into an existing object file, the dependency information that **clearmake** previously recorded for that object file is no longer updated. The next time that **clearmake** is invoked, it will rebuild the object file. After this rebuild, the dependency information for the object file is correct once again. At this point, **clearmake** no longer executes unnecessary rebuilds of that object file.

The Automatic Instantiation method is the easiest to use because it requires no programmer intervention and it is suitable for most applications. Aside from the unnecessary rebuilds described above, this method does not conflict with ClearCase configuration management.

COMPILE-TIME DEMAND INSTANTIATION

The Compile-Time Demand Instantiation method instantiates templates at compile-time, rather than during a prelink step. To use this method, specify the **-tused** and **--no_auto_instantiation** option to the Green Hills C++ compiler.

This option causes the compiler to compile all the template code that the source module refers to into the object module. If multiple source modules refer to the same template class or function, copies of the compiled template code appear in multiple object modules.

The Compile-Time Demand Instantiation is easy to use, requiring the programmer only to specify extra compiler options. It is suitable for most applications, especially for building archives. Also, this method does not conflict with ClearCase configuration management. The disadvantage of the method is that the compiler uses extra time and disk space to perform redundant template instantiation, and the same instantiation may appear in multiple source files causing programs to be larger than necessary.

EXPLICIT INSTANTIATION

The Explicit Instantiation method is an alternate form of compile-time instantiation. The Green Hills C++ compiler allows you to add directives to the source code to specify which template classes to instantiate.

When it compiles a source module, the compiler instantiates all the template classes specified by the directives in the source. The compiler instantiates each template classes completely, that is, it instantiates every member function and static data member of the class.

To use the Explicit Instantiation method, follow these steps:

1. For each template class to instantiate, add one **#pragma instantiate** directive to the source code. For example, if the program requires the **Array<String>** class, then add the following directive:

```
#pragma instantiate Array<String>
```

2. In each source file that contains a **#pragma instantiate** directive, include the header files that contain definitions of the templates and classes used in the directives.
3. This step is optional: Disable automatic instantiation by specifying the **-tnone** and **--no_auto_instantiation** compiler options. Automatic instantiation does not interfere with explicit instantiation, but you may choose to disable it.

The Explicit Instantiation method requires more effort to use. However, it allows you to control the placement of instantiated template code into object modules. This control is useful in some situations, especially when building archives of instantiated template code. Using explicit instantiation does not conflict with **clearmake** build avoidance.

NAMESPACE SUPPORT

Namespaces are enabled by default except in the cfront modes. Options can be used to enable or disable the features.

Name lookup during template instantiations now does something that approximates the two-phase lookup rule of the X3J16/WG21 Working Paper. When a name is looked up as part of a template instantiation but is not found in the local context of the instantiation, it is looked up in a synthesized instantiation context. The Green Hills C++ compiler follows the new instantiation lookup rules for namespaces as closely as possible in the absence of a complete implementation of the new template name binding rules.

For example:

```
namespace N {
    int g(int);
    int x = 0;
    template <class T> struct A {
    T f(T t) {return g(t);}
    T f() {return x;}
    };
}
namespace M {
    int x =99;
    double g(double);
    N::A<int> ai;
    int i = ai.f(0);  // N::A<int>::f(int) calls
                      // N::g(int)
    int i2 = ai.f();  // N::A<int>::f() returns 0 (=
                      // N::x)
    N::A<double> ad;
    double d = ad.f(0);  // N::A<double>::f(double)
                         // calls M::g(double)
    double d2 = ad.f();  // N::A<double>::f() also
                         // returns 0 (= N::x)
}
```

The lookup of names in template instantiations does not conform to the rules in the working paper in the following respects:

- Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point at which the template was defined.
- Functions from the context in which the template was referenced are considered for all function calls in the template. Functions from the referencing context should only be visible for "dependent" functions calls.

The lookup rules for overloaded operators are implemented as specified by the Working Paper, which means that the operator functions in the global scope overload with the operator functions declared extern inside a function, instead of being hidden by them. The old operator function lookup rules are used when namespaces are turned off. This means a program can have different behavior, depending on whether it is compiled with namespace support enabled or disabled:

```
struct A {};
A operator+(A, double);
void f() {
    A a1;
    A operator+(A, int);
    a1 + 1.0; // calls operator+(A, double) with
              // namespaces enabled
}             // but otherwise calls operator+(A, int);
```

The interaction between **friend** declarations and namespaces is incompletely (or incorrectly) specified in the current Working Paper; pending clarification, the following implementation choices have been made:

- A namespace-qualified **friend** declaration must refer to a previously declared entity.
- A globally qualified name is permitted in a **friend** declaration (e.g., **friend void ::f();**) as an extension; it too must refer to an existing entity.
- An unqualified **friend** declaration may be a definition, but a namespace-qualified **friend** declaration may not.
- The lookup of an unqualified **friend** declaration begins in the innermost non-class scope and continues no further than the innermost namespace scope.

The final rule (which for **friend** declarations in non-local classes effectively requires that the scope for name lookup and the scope for name injection be the

same) prevents a namespace from being “polluted” by declarations from an enclosing namespace. For example:

```
namespace N {
    class A {
    friend void f();// always declares N::f regardless
                    // of whether ::f is visible
    };
}
```

The programmer is assured that **f** is injected into namespace **N** whether or not there is a declaration of **f** in the scope enclosing **N**.

PRECOMPILED HEADERS

It is often desirable to avoid recompiling a set of header files, especially when they introduce many lines of code and the primary source files that **#include** them are relatively small. The Green Hills C++ compiler provides a mechanism for, in effect, taking a snapshot of the state of the compilation at a particular point and writing it to a disk file before completing the compilation. Then, when recompiling the same source file or compiling another file with the same set of header files, it can recognize the snapshot point, verify that the corresponding precompiled header (PCH) file is reusable, and read it back in. Under the right circumstances, this can produce a dramatic improvement in compilation time. The trade off is that PCH files can take up a lot of disk space.

AUTOMATIC PCH PROCESSING

When **--pch** appears on the command line, automatic precompiled header processing is enabled. This means the compiler will automatically look for a qualifying precompiled header file to read in and/or will create one for use on a subsequent compilation.

The PCH file will contain a snapshot of all the code preceding the header stop point. The header stop point is typically the first token in the primary source file that does not belong to a preprocessing directive, but it can also be specified directly by **#pragma hdrstop** (see section Other Ways to Control PCH’s on page 34) if that comes first. For example:

```
#include “xxx.h”
#include “yyy.h”
int i;
```

The header stop point is **int** (the first non-preprocessor token) and the PCH file will contain a snapshot reflecting the inclusion of **xxx.h** and **yyy.h**. If the first non-preprocessor token or the **#pragma hdrstop** appears within a **#if** block, the header stop point is the outermost enclosing **#if**. To illustrate, here's a more complicated example:

```
#include “xxx.h”
#ifndef YYY_H
#define YYY_H 1
#include “yyy.h”
#endif
#if TEST
int i;
#endif
```

Here, the first token that does not belong to a preprocessing directive is again **int**, but the header stop point is the start of the **#if** block containing it. The PCH file will reflect the inclusion of **xxx.h** and conditionally the definition of **YYY_H** and inclusion of **yyy.h**; it will not contain the state produced by **#if TEST**.

A PCH file will be produced only if the header stop point and the code preceding (mainly, the header files themselves) meet certain requirements:

- The header stop point must appear at file scope; it may not be within an unclosed scope established by a header file. For example, a PCH file will not be created in this case:

  ```
  // xxx.h
  class A {

  // xxx.C
  #include “xxx.h”
  int i; };
  ```

- The header stop point may not be inside a declaration started within a header file, nor (in C++) may it be part of a declaration list of a linkage specification. For example, in the following case the header stop point is **int**, but since it is not the start of a new declaration, no PCH file will be created:

```
// yyy.h
static

// yyy.C
#include “yyy.h”
int i;
```

- Similarly, the header stop point may not be inside a **#if** block or a **#define** started within a header file.
- The processing preceding the header stop must not have produced any errors. Note that warnings and other diagnostics will not be reproduced when the PCH file is reused.
- No references to predefined the predefined macros **_ _DATE_ _** or **_ _TIME_ _** may have appeared.
- No use of the **#line** preprocessing directive may have appeared.
- **#pragma no_pch** (see section Other Ways to Control PCH’s on page 34) must not have appeared.
- The code preceding the header stop point must have introduced a sufficient number of declarations to justify the overhead associated with precompiled headers.

When a precompiled header is produced, it contains, in addition to the snapshot of the compiler state, some information that can be checked to determine under what circumstances it can be reused. This includes:

- The compiler version, including the date and time the compiler was built.
- The current directory (i.e., the directory in which the compilation is occurring).
- The command line options.
- The initial sequence of preprocessing directives from the primary source file, including **#include** directives.
- The date and time of the header files specified in **#include** directives.

This information comprises the PCH prefix. The prefix information of a given source file can be compared to the prefix information of a PCH file to determine whether the latter is applicable to the current compilation.

As an illustration, consider two source files:

```
// a.C
#include “xxx.h”
// Start of code
// b.C
#include “xxx.h”
// Start of code
```

When **a.C** is compiled with the **--pch** option, a precompiled header file named **a.pch** is created. Then, when **b.C** is compiled (or when **a.C** is recompiled), the prefix section of **a.pch** is read in for comparison with the current source file. If the command line options are identical, if **xxx.h** has not been modified, and so forth, then, instead of opening **xxx.h** and processing it line by line, the compiler reads in the rest of **a.pch** and thereby establishes the state for the rest of the compilation.

It maybe be that more than one PCH file is applicable to a given compilation. If so, the largest (i.e., the one representing the most preprocessing directives from the primary source file) is used. For instance, consider a primary source file that begins with:

```
#include “xxx.h”
#include “yyy.h”
#include “zzz.h”
```

If there is one PCH file for **xxx.h** and a second for **xxx.h** and **yyy.h** together, the latter will be selected (assuming both are applicable to the current compilation). Moreover, after the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers may be created.

When a precompiled header file is created, it takes the name of the primary source file, with the suffix replaced by **.pch**. Unless **--pch_dir** is specified (see Other Ways to Control PCH’s on page 34), it is created in the directory of the primary source file.

When a precompiled header file is created or used, a message such as the following is issued:

```
“test.C”: creating precompiled header file “test.pch”
```

You may suppress the message by using the command-line option **--no_pch_messages**.

In automatic mode (i.e., when the **--pch** option is used) the compiler will consider a precompiled header file obsolete and delete it under the following circumstances:

- if the precompiled header file is based on at least one out-of-date header file but is otherwise applicable for the current compilation
- if the precompiled header file has the same base name as the source file being compiled (e.g., **xxx.pch** and **xxx.C**) but is not applicable for the current compilation (e.g., because of different command-line options).

This handles some common cases. Other PCH file clean-up must be dealt with by the user.

Support for precompiled header processing is not available when multiple source files are specified in a single compilation: an error will be issued and the compilation aborted if the command line includes a request for precompiled header processing and specifies more than one primary source file.

MANUAL PCH PROCESSING

The command-line option **--create_pch=***filename* specifies that a precompiled header file of the specified name should be created.

The command-line option **--use_pch=***filename* specifies that the indicated precompiled header file should be used for this compilation. If it is invalid (i.e., if its prefix does not match the prefix for the current primary source file), a warning will be issued and the PCH file will not be used.

When either of these options is used in conjunction with **--pch_dir**, the indicated file name (which may be a path name) is tacked on to the directory name, unless the file name is an absolute path name.

The **--create_pch**, **--use_pch**, and **--pch** options may not be used together. If more than one of these options is specified, only the last one will apply.

Nevertheless, most of the description of automatic PCH processing applies to one or the other of these modes. Header stop points are determined the same way, PCH file applicability is determined the same way, etc.

OTHER WAYS TO CONTROL PCH'S

There are several ways you can control and/or tune how precompiled headers are created and used.

- **#pragma hdrstop** may be inserted in the primary source file at a point prior to the first token that does not belong to a preprocessing directive. It enables the user to specify where the set of header files subject to precompilation ends. For example:

```
#include “xxx.h”
#include “yyy.h”
#pragma hdrstop
#include “zzz.h”
```

  Here the precompiled header file will include processing states for **xxx.h** and **yyy.h** but not **zzz.h**. This is useful if you decide that the information added by what follows the **#pragma hdrstop** does not justify the creation of another PCH file. This applies to C++ only. See Precompiled Headers on page 29 for more information.
- **#pragma no_pch** may be used to suppress precompiled header processing for a given source file.
- The command line option **--pch_dir=***directory* is used to specify the directory in which to search for and/or create a PCH file.

PERFORMANCE ISSUES

The relative overhead incurred in writing out and reading back in a precompiled header file is quite small for reasonably large header files.

In general, it doesn't cost much to write a precompiled header file out even if it does not end up being used, and if it is used it almost always produces a significant speedup in compilation. The problem is that the precompiled header files can be quite large (from a minimum of about 250K bytes to several megabytes or more), and so you probably don't want many of them sitting around.

Thus, despite the faster recompilations, precompiled header processing is not likely to be justified for an arbitrary set of files with nonuniform initial sequences of preprocessing directives. Rather, the greatest benefit occurs when

a number of source files can share the same PCH file. The more sharing, the less disk space is used. With sharing, the disadvantage of large precompiled header files can be minimized, without giving up the advantage of a significant speedup in compilation times.

Consequently, to take full advantage of header file precompilation, users should expect to reorder the **#include** sections of their source files and/or to group **#include** directives within a commonly used header file.

Different environments and different projects will have different needs, but in general, you should be aware that making the best use of the precompiled header support will require some experimentation and probably some minor changes to source code.

Note: The **-fnone** option does not work with the standard C++ header files.

CROSS REFERENCE INFORMATION

Here is information to read the cross reference information produced by the **--xref** option (or from the GUI: **Options->C++->More Options->Listing Options->Cross Reference File**).

The format for the cross reference information is:

```
symbol-id name ref-code file-name line-number
column-number
```

All fields are separated by tabs. Here is more information to understand the cross reference information:

Field Name	Meaning
symbol-id	A unique decimal number for the symbol (differentiates different variables with the same name).
name	The symbol name.

Field Name	Meaning
ref-code	D for definition d for declaration M for modification A for address taken U for a use C for changed (used and modified in a single operation, like ++variable;) R for any other kind of reference E for an error which causes the kind of reference to be indeterminate
file-name	Source file in which the reference occurs.
line-number	Line number on which the reference occurs.
column-number	Column number at which the reference occurs.

PREPROCESSOR

Green Hills C++ uses an ANSI C compliant preprocessor.

PREDEFINED C++ SYMBOLS

The following symbols are predefined for C++, in addition to those listed in the *Green Hills C User's Guide.*

Macro Name	Value	Description
_ _ARRAY_OPERATORS	1	Defined when array new and delete are enabled.
_BOOL	1	Defined when **--bool** is specified on the driver command line
_ _cfront	1	Indicates cfront
_ _c_plusplus _ _cplusplus	1	Indicates C++ (this is for backwards compatibility with some older C++ implementations)
_ _EDG_IMPLICIT_USING_STD	1	Defined when an implicit "using namespace std" is done.
_ _EMBEDDED_CXX	1	Indicates embedded C++.
_ _EXCEPTION_HANDLING _ _EXCEPTIONS	1	Indicates C++ compiler is running in a mode that allows exception handling.
_ _ghs	1	Indicates this is a Green Hills compiler
_ _NAMESPACES	1	Indicates C++ namespaces accepted.

Table 1 **Predefined Symbols in C++**

Macro Name	Value	Description
_ _PLACEMENT_DELETE	1	Defined when placement delete is enabled.
_ _RTTI	1	Indicates Runtime Type Identification code accepted.
_ _STDC_ _	0	Indicates C++ or ANSI C
_WCHAR_T	1	Defined if **wchar_t** is a keyword.

Table 1 **Predefined Symbols in C++**

ASM STATEMENT

The **asm** statement generates in-line assembly code, and can be used anywhere a statement can appear.

The **asm** syntax is as follows:

asm (“ *assembler_instruction operands*”);

Note that there must be a space or tab between the first double quotes (**“**) and the assembler instruction.

For example:

```
asm (“ sethi  %hi(L16),%o0“);
```

This statement drops the **sethi** instruction into the assembly code generated by the compiler, corresponding exactly to where the compiler found it in the source code.

Since the code generated by Green Hills C is substantially different from the code generated by other compilers, it is usually necessary to modify most **asm** statements. Also, the code generated by the **asm** statement is of course specific to the target on which the source file was originally compiled.

It is important to note that the **asm** statement will not function if object code is directly produced.

LINKAGE

C++ accepts the **extern** *language* directive in order to achieve linkage between C++ and C. The full syntax is as follows:

extern *language* {

declarations

}

or

extern *language declaration*;

where *language* may be **C** or **C++**.

Language	Effect on Resulting Code
C	C++ does not alter the procedure name as it usually would when confronted with an overloaded function.
C++	Uses C++ naming rules. Function names are always mangled according to C++ linkage specifications. This is the default linkage.

Table 2 **Function/Procedure Naming with extern**

Note that the **extern "***language***"** directive only affects the external names of functions so that the compiler will apply the appropriate function naming rules. This directive does not modify the type or number of arguments of a function, or its return type. Normal C++ type checking rules are not altered by this directive.

For more information on using C++ with C, see Chapter 3, "Mixing Languages".

PRAGMAS

#pragma directives are used within the source program to request certain kinds of special processing. The **#pragma** directive is part of the standard C and C++ languages, but the meaning of any pragma is implementation-defined.

In addition to the **#pragma** directives specifically recognized for C++, all "**#pragma ghs**" directives are recognized. See the *Green Hills C User's Guide* for details.

The Green Hills C++ compiler recognizes several pragmae. The following are described in detail in section Template Instantiation on page 16:

```
#pragma instantiate
#pragma do_not_instantiate
#pragma can_instantiate
```

and two others are described in section Precompiled Headers on page 29:

```
#pragma hdrstop
#pragma no_pch
```

The compiler also recognizes **#pragma once**, which, when placed at the beginning of a header file, indicates that the file is written in such a way that including it several times has the same effect as including it once. Thus, if the compiler sees **#pragma once** at the start of a header file, it will skip over it if the file is included again with a **#include** statement.

A typical idiom is to place a **#ifndef** guard around the body of the file, with a #define of the guard variable after the **#ifndef**:

```
#pragma once        // optional
#ifndef FILE_H
#define FILE_H
// body of header file goes here
#endif
```

The **#pragma once** is marked as optional in this example because the compiler recognizes the **#ifndef** idiom and does the optimization even in its absence. **#pragma once** is accepted for compatibility with other compilers and to allow the programmer to use other guard-code idioms.

POST PROCESSING IN C++

NOTE: This section does not apply to VxWorks/Tornado.

Global objects in C or C++ (or non-local static objects) are those objects which are declared outside of the scope of any function and are available throughout the entire program. C++ objects which are instances of a class type have mechanisms for automatic construction/initialization and destruction/cleanup through the use of constructors and destructors. Constructors are functions which are automatically called by the compiler when an object is created. Destructors are called when an object is deleted. This implies some

implementation-specific behavior for global objects which may vary from C++ system to C++ system.

Global objects, such as those in libraries (e.g. **cin**, **cout**, and **cerr**) must be constructed and initialized for the entire program. This means that the constructor functions must be called as soon as the program begins. The compiler has no knowledge of external global objects contained in other modules or libraries except for an extern declaration. This is not enough information for the compiler to be able to properly insure that these calls are done. For targets that use the GHS linkers, the linker resolves the global constructor and destructor information. In other environments, the **cxxmunch** utility assumes this responsibility. The VxWorks environment is unique in that the module load/unload functions invoke the global constructors/destructors, or else the user executes them manually.

After an executable has been produced, the Green Hills C++ compiler driver calls the **nm** utility to find all global symbols. The output of **nm** is sent to the postlink program **cxxmunch**. **cxxmunch** searches for all global constructor and destructor calls and generates a C module which will execute these calls appropriately at program startup and exit. The driver then invokes the compiler and assembler to produce another object module. Then the linker is invoked to relink the new constructor/destructor object with the original object and libraries. This produces the fully processed C++ executable, which has all of the appropriate constructor and destructor calls.

The driver makes all of this processing completely transparent to the user. However, if you do not use the driver provided by Green Hills, then you are responsible for calling the postlink program after producing an executable, otherwise your program may not run correctly.

C++ UTILITIES

The following utility is provided with Green Hills C++.

DECODE

decode [*names*]

Print demangled string from the specified C++ mangled string. If no names are given, then standard input will be read. For example:

```
$ decode adjustfield__3ios
ios::adjustfield
```

Chapter 2

EC++/ESTL FEATURES

THIS CHAPTER CONTAINS:

- How to effectively use GHS C++
- C++ in the Wind River VxWorks/Tornado Environment
- Introduction to EC++
- Introduction to ESTL
- Getting started with EC++ and ESTL
- Standard C++
- GHS Solution

Described in this Chapter are the features of Embedded C++ (EC++) and Extended Standard Template Libraries (ESTL).

HOW TO EFFECTIVELY USE GHS C++

The needs of C++ users vary widely, depending on a number of factors. These factors involve such considerations as the target application, the target environment, foreign libraries involved, compatibility with other C++ compilers, and the trade-offs programming teams make in regard to the C++ feature set and library support they require.

To meet such a diverse set of needs, GHS supports a concept of “scalable” C++. The language level and library level is switch selectable, giving the user the choice of everything from a small and efficient “Embedded C++” compiler and library, to the power of the full ANSI draft “Standard” C++ language and library.

The C++ language and library level do not necessarily need to be matched. For example, a user may choose to program in Standard C++ but may feel that a scaled back library suits their needs. The rule of thumb is that the language level must be at least as high as the library level, otherwise the compiler may encounter features in the library header files that it isn’t authorized to deal with.

The EC++ library is the smallest, but least powerful. The Standard C++ library is the largest and most powerful. The ESTL library falls in between. For embedded applications, choose the smallest library that will meet your needs.

Allowable language/library combinations are:

Language Level	Allowable Library Level
Standard C++	Standard C++ library, ESTL library, EC++ library
ESTL	ESTL library, EC++ library
EC++	EC++ library

A comparison between language levels:

Standard C++	ESTL	EC++
Fully ANSI draft supported.	Removed from ANSI draft C++: -exception handling -multiple inheritance -virtual base classes	Removed from ANSI draft C++: -exception handling -multiple inheritance -virtual base classes -templates -namepsaces -mutable keyword -new-style casts

In ESTL and EC++ all Standard C++ keywords are retained for upward compatibility.

A comparison between library levels:

Standard C++	ESTL	EC++
All ANSI draft support.	Removed from ANSI draft: -exception handling -localization -environment	Removed from ANSI draft: -exception handling -localization -environment -STL -file operations -wchar_t -long double

Note: The Standard and ESTL libraries are built within the STD namespace. Since namespaces are not present in EC++, the EC++ library is not built in a namespace.

Considerations for picking a language level.

Standard C++

The full ANSI draft Standard C++ language, including all the latest features and changes. All of the features of Standard C++ are turned on by default, with the exception of exception handling. Exception handling must be turned on explicitly, since there is a code size and speed penalty to be paid even if EH features are not used. Many individual C++ features can be turned

off if desired (For example, the semantics of the for loop variable definitions have changed in the ANSI draft causing an incompatibility with some pieces of old code. In this case a user may choose to turn on the option that causes the compiler to use the old code for loop variable semantics). Of course one advantage to using Standard C++ is that it is the language which is described in newer C++ books, and that there is an actual standard to define proper behavior.

ESTL — A compromise language level, existing between ANSI draft C++ and Embedded C++. Aimed at large embedded applications, this language offers all of the features of Standard C++ except for those features that adversely effect code speed. These features are exception handling, multiple inheritance and virtual base classes. As with Standard C++, other features can be individually controlled through fine-tuning options. Notable features such as templates and namespaces are supported.

EC++ — The smallest and most efficient version of C++. EC++ is a standardized C++ variation designed for the needs of small embedded applications. EC++ excludes language features deemed expensive in terms of code space or speed, features that have poor runtime characteristics for speed, features that have poor runtime characteristics for some pieces of embedded code, and features that are thought to be overly complex for the needs of a small embedded applications. The result is a relatively simple and efficient, yet powerful C++ language.

Considerations for picking a library level:

Standard C++ — The full ANSI draft C++ library. As with the full Standard C++ language, this is the library that is described in C++ books. The library is built around the standard template library (STL).

ESTL — A scaled back version of the full ANSI draft library to meet the needs of large embedded projects. The library is still based around the STL, but has tossed out features which are thought to be not useful in an embedded applications.

EC++ — The smallest and most efficient C++ library. Heavily optimized for smallest code and data size.

C++ IN THE WIND RIVER VXWORKS/TORNADO ENVIRONMENT

All libraries and thread-safe exception handling are supported.

Notes:

The Standard, ESTL, and EC++ libraries require that **wchar_t** be a keyword. You may still select the type for **wchar_t**, but must not use the option that removes it from the keyword (and fundamental type) list.

COMMAND LINE DRIVER OPTIONS:

EC++ language:	--e
ESTL language:	--ee
(ANSI) Standard C++ language:	--std (ANSI violations are warnings)
(ANSI) Standard C++ language:	--STD (ANSI violations are errors)
EC++ library (no EH):	--el
EC++ library (with EH):	--ele
ESTL library(no EH):	--eel
ESTL library (with EH):	--eele
(ANSI) Standard library (no EH):	--stdl
(ANSI) Standard library (with EH):	--stdle

Note: “EH” refers to Exception Handling.

INTRODUCTION TO EC++

WHAT IS EC++?

EC++ (Embedded C++) originated in Japan through the efforts of the following committee members:

Green Hills	ADaC
Toshiba	Plum Hall
Hitachi	Dinkumware
NEC	Cygnus

The purpose of EC++ is to create a stable, simple, and efficient version of Standard C++, with the intent to produce an open standard in the future to be used worldwide. EC++ is not a new language specification that will compete with existing Standard C++. Rather, it is a pure subset for the practical user of C++.

EC++ is designed to meet the needs of the embedded industry. The committee members established the following guidelines for creating EC++. The subset fulfills the particular requirements of embedded systems designs:

- Remove complex features and specifications while retaining as many object-oriented features as possible.
- Avoid those features and specifications that do not fulfill the requirements of embedded system design. Three major requirements of embedded system designs are: 1) Avoiding excessive memory consumption, 2) Taking the care not to produce unpredictable responses, and 3) Making code ROMmable.
- Non-standard extensions to C++ should be avoided.
- The founders of the EC++ committee state: Our background is in the semiconductor business. We mainly target 32-bit RISC MCU applications as embedded systems. Although there are many applications using 4 or 8-bit MCUs, we cannot address them. We feel that the basic features of C, or even assembly language, are sufficient for these processors. On the other hand, those systems are expandable using standard buses, such as VME or PCI, are similar to those of PCs or workstations. We recognize that a full version of Standard C++ is better than Embedded C++ for those application designs.

EC++ is similar to C++ (1990) except for the following conditions:

- Many small enhancements and clarifications are retained from ANSI C++
- Unused keywords are retained from ANSI C++ for upward compatibility
- A style guide is available, providing guidelines for using EC++ wisely

New additions to the ANSI C++ language have been made. One such feature is mutable keyword. The mutable keyword allows the user to modify class members even if the object has been declared **const**. These cannot be placed in ROM. Previously, the object was placed in ROM, without the ability to manipulate it.

New-style casts force the programmer to be explicit about the type of cast being performed. The new-style casts are self documenting, and force the user to be aware when doing a dangerous cast. The EC++ committee felt the learning curve to use the new-style casts was too steep to justify their benefit.

The following items are language elements which have poor runtime characteristics and have not been included in EC++:

- Exception handling: It is difficult to estimate the time between when an exception has occurred and control has passed to a corresponding exception handler. It is also difficult to estimate memory consumption for exception handling.
- Multiple inheritance and virtual base classes: Designing a class hierarchy using multiple inheritance or recognizing the overall hierarchy of it and using it correctly is difficult. The programs are less readable, less usable, and more difficult to maintain.
- RTTI: Program size is a factor when supporting the runtime type identification facility. To support the runtime type identification (RTTI) facility, there is at least some program size overhead, because type information for polymorphic classes is needed. The compiler automatically generates the information, and it would be included in programs that do not use the RTTI facility.

The following language elements are overly complex for embedded programming and have not been included in EC++:

- Templates: Templates are complex items that increase the time of compilation and cause unexpected code explosion. For these reasons, templates have not been implemented in EC++.
- Namespaces: To avoid serious name conflicts, using a static member of a class is recommended. This will prevent the need of using namespaces in the first place.

EC++ Library Features

The EC++ library is a subset of the ANSI C++ library. The EC++ library is designed to meet the needs of the embedded industry. It is much smaller and more efficient than the full standard C++ library.

The EC++ library represents a significant addition to the typical C library supplied with an embedded compiler. For a close approximation of the Embedded C++ library, see P.J. Plauger, *The Draft Standard C++ Library*, Prentice-Hall, 1995.

- Iostreams operations are supported for **cin** and **cout**, using classes **istream**, **ostream**, **ios**, and **streambuf**.
- String operations are supported for class **string**.
- Math functions are overloaded for both **double** and **float**, in both real and complex modes.

The Embedded C++ library also benefits from a few additions:

- Input/output to strings makes sense even in an embedded environment (header or the older header).
- Allocators for string objects can also make sense, if tailorable by the programmer.
- If **fopen** and **flcose** work in the Standard C library, then the classes **ifstream**, **ofstream**, and **filebuf** (in header **fstream**) are also powerful additions.

To enhance the EC++ library, the following features were removed:

- No templates implies no Standard Template Library, no templatized string, complex, or iostreams classes.
- No exceptions implies no exception handling functions or classes.
- No runtime type identification implies no **type_info** class.

The library omits support for wide character input/output, locales, and **long double** arithmetic, because they are seldom needed in embedded applications.

INTRODUCTION TO ESTL

WHAT IS ESTL?

ESTL (Extended C++ with the Standard Template Library) was created in a joint effort by Green Hills Software and P.J. Plauger. Keep in mind that this is not an official standard.

ESTL is a scalable C++ version which is simply a compromise between the Standard C++ and EC++. Unlike EC++, it adds back features that are not costly in size or speed such as the following:

- Templates
- Namespaces
- Mutable keyword
- Most new-style casts

However, the following are still eliminated from the Standard C++:

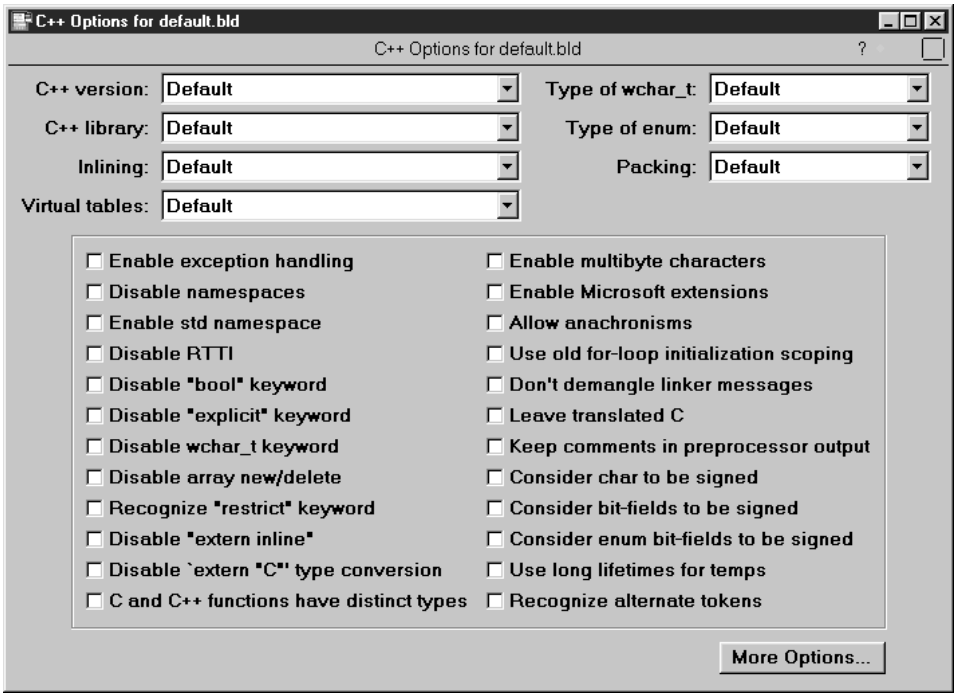
- Exception handling
- Multiple inheritance and virtual base classes

ESTL Library Features

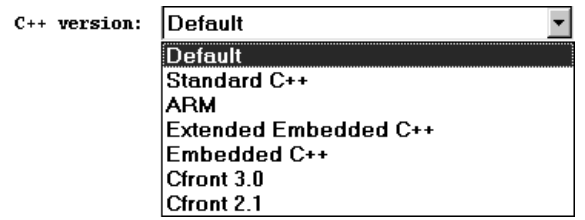
The ESTL library features include STL and namespaces. It complements EC++, allowing the users to have the most out of the enhanced language.

GETTING STARTED WITH EC++ AND ESTL

To get started with EC++ or ESTL, go to the **Options** menu and select **C++...** from the MULTI Builder window.

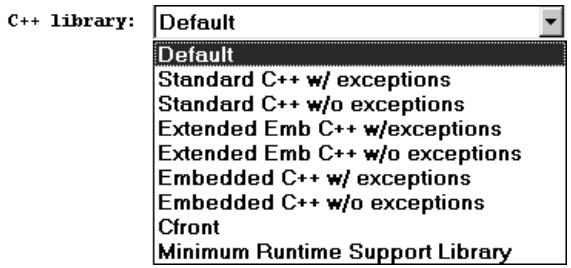


To select the desired C++ version, pull down the menu and select the following items:



Note that ESTL (Extended Standard Template Libraries) is equivalent to Extended Embedded C++ in the pull down menu options.

To select the desired C++ library, pull down the menu and select the following items:



Note that ESTL (Extended Standard Template Libraries) is equivalent to Extended Embedded C++ in the pull down menu options.

STANDARD C++

FEATURES

The key features of C++ are the following items:

- STL is built into the Standard library
- Exception handling
- Namespaces
- RTTI
- The library and headers are part of the standard

Due to the standards EC++ maintains, these features have been removed from EC++ and its corresponding library.

ADDITIONAL INFORMATION

For more information about EC++, please visit the web site:

http://www.caravan.net/ec2plus

For more information about Standard C++, we recommend:

Stroustrup, *The C++ Programming Language, Third Edition.*

Chapter

3

MIXING LANGUAGES

HOW THE DRIVER BUILDS A MIXED LANGUAGE EXECUTABLE

With Green Hills compilers, you can mix and match C, C++, FORTRAN, Pascal, and Ada routines in the same executable files, subject to certain constraints.

The Green Hills drivers are compatible. This permits a C driver to compile a FORTRAN module, and a Pascal driver to compile a C++ module. The driver uses the input filename extension to determine the correct language, rather than assuming that the name of the driver determines the source code language.

While completely interchangeable during compilation, the various drivers differ during the link phase. To link an application the driver must determine all of the languages in use, in order to know which libraries to include. The driver assumes that every application has modules written in C and assembly language, and further, that there is at least one module written in the driver's default language. If source files written in other languages are on the command line, as indicated by the file extension, then the driver recognizes that those languages exist in the application as well.

Therefore, mixing any one language with C is easy, as the driver always assumes C is in use. In this case, the driver for the language other than C should be used for linking the application, to assure the correct linkage.

To link two languages other than C into a single application, all of the source files are placed on the command line so the driver can compile and link in a single step. This assures giving the driver full information during the link phase.

The most difficult case is where each module must be compiled separately and the link phase is done strictly from object files which come from several different languages. In this case, it is best to use the driver for the language with the most complicated linkage requirements. Specifically, to link C, Fortran, and Pascal, use the Fortran driver and add the Pascal library at the end of the driver command line. To link C, C++ and either Fortran or Pascal, use the C++ driver and place the Fortran or Pascal libraries at the end of the driver command line.

THE -LANGUAGE OPTION

The **-language** option facilitates mixing languages. It is written as:

-language=*language*

where *language* is either **cxx**, **fortran**, or **pascal**. It is not necessary to specify C.

The **-language** option tells the driver that files written in *language* are being mixed with the default language. This option is specified once for each language being mixed. It is not necessary to specify the driver's default language.

EXAMPLES:

Three precompiled object files, **main.o**, **pigeon.o**, and **falcon.o**, are written in C, Pascal, and FORTRAN, respectively. The following command line tells the driver about all three languages when linking:

```
gfc -language=pascal main.o pigeon.o falcon.o
```

Here, the driver knows about FORTRAN because the FORTRAN driver is being used (**gfc**). All drivers assume C, and the **-language=pascal** option informs the driver about Pascal.

To link the same three modules with the C driver:

```
gcc -language=pascal -language=fortran main.o pigeon.o falcon.o
```

INITIALIZATION OF LIBRARIES

A multiple language application may need to perform input and output in more than one language. With a little care to avoid conflicts between languages, this is fully supported. If input and output will always be performed on different files by each language, then the initialization and deinitialization of each language's runtime routines is handled automatically by the main program in a single language application. Therefore, if the application will only perform I/O in one language other than C, then it is easy to write the main program for the application in that language. For more complex requirements, a main program may be written in C which performs the initialization and deinitialization of the library runtime routines.

A C MAIN() PROGRAM FOR C++

```
void main() {
    _main();/* must be first executable line */

    /* rest of main goes here */

    exit(0);/* must be last executable line */
}
```

A C MAIN() PROGRAM FOR FORTRAN

```
int _ _gh_argc;
char **_ _gh_argv;

extern void (*_ _gh_initrec)();
extern void (*_ _gh_uninitrec)();

int main(int argc, char **argv)
{
    _ _gh_argc = argc;
    _ _gh_argv = argv;

    if (_ _gh_initrec)
    _ _gh_initrec();

/* rest of main goes here */

    if (_ _gh_uninitrec)
    _ _gh_uninitrec();
    return(0);
}
```

A C MAIN() PROGRAM FOR PASCAL

```
void main(int argc, char **argv)
int argc;
char **argv;
{
    extern int __argc;
    extern char **__argv;
    __argc = argc;
```

```
    __argv = argv;/* the 4 lines above must be first */

    /* rest of main goes here */

    __GHSexit(0);/* must be last executable line */
}
```

A C main() Program for Ada

When using Ada, the user should use the Ada main program. The user should NOT create a main program for Ada in C language. Ada initialization must be performed by an Ada main program.

Performing I/O on a Single File in Multiple Languages

Some applications benefit from performing input and output on a single file or device from more than one language; an example is pre-opened files. In C, these are **stdin**, **stdout**, and **stderr**. In C++, they are **cin**, **cout**, and **cerr**. In FORTRAN, they are **Units 5**, **6**, and **0** respectively. In Pascal, the first two files are **input** and **output**, and the equivalent of C's **stderr** cannot be used directly.

All languages have full access to these pre-opened files, and input and output can easily be mixed between the languages on these files. However, for the best results, a complete input or output operation is done in a single language. In FORTRAN, a single **READ**, **WRITE**, or **PRINT** statement is a complete operation. In Pascal, a single **read**, **readln**, **write**, or **writeln** call is a complete operation. In C, any call to a library function which performs input or output is a complete operation. If this rule is followed, all data will be output correctly and in the intended sequence. The C library routine **fflush()** flushes the buffer of the pre-opened files in all languages, except in C++. To flush one of these files in C++, use the notation *file*<<**flush**. For example, **cout**<<**flush**.

Performing input and output on a single file which is not preopened is more difficult. It is possible to open the file once in each language and perform input and output independently in each language. In many cases this would be unacceptable, particularly when working with a device rather than a simple file.

It is possible to open a file in FORTRAN and subsequently perform input and output on that file by using the FORTRAN library routines **GETCHAN** and **GETFD**. The FORTRAN function **GETCHAN** takes a single argument which is the **Unit** number of a FORTRAN file and returns a **FILE*** which can then be

used with C library routines such as **fprintf()**, **fread()**, **fwrite()**, **fflush()**, **fseek()**, **fstat()** and **fputc()**. Operations on such a file are compatible to the same extent as the three pre-opened files.

The FORTRAN function **GETFD** takes a single argument which is the **Unit** number of a FORTRAN file and returns an integer which can then be used with lower level routines such as **read()**, **write()**, **lseek()**, and **stat()**. Because these low level routines are not compatible with **fprintf()**, **fread()**, **fwrite()**, etc., their use may conflict with the FORTRAN runtime routines.

There is currently no mechanism for performing input and output in C on a file opened in Pascal.

NATIVE UNIX LIBRARIES VERSUS GREEN HILLS LIBRARIES

This section refers only to native UNIX users.

Although the combination of multiple languages in a single application is fully supported, certain differences cannot be avoided between programs written entirely in one language and those written in multiple languages, due primarily to library selection.

The C and C++ languages use the native UNIX math and C libraries by default. The FORTRAN and Pascal languages use the Green Hills math library. ANSI C uses the Green Hills math and C libraries. This means that the combination of FORTRAN and C will cause the entire application to use the Green Hills math library, and the combination of ANSI C with C++ will cause the entire application to use the Green Hills math and C libraries. Therefore, programs written entirely in C or C++ may behave differently than otherwise identical programs written partially in C or C++ and partially in FORTRAN or ANSI C.

CALLING A C ROUTINE FROM FORTRAN

This section shows how to call C subroutines from FORTRAN.

ARGUMENT PASSING

By default, all FORTRAN arguments are passed by reference. Therefore, each parameter in the called C routine must be a pointer of the appropriate type. The following table shows how arguments passed by FORTRAN are received by C:

FORTRAN Passes	C Receives
REAL or REAL*4	float *
DOUBLE PRECISION or REAL*8	double *
INTEGER or INTEGER*4	long *
INTEGER*2	short *
INTEGER*1	signed char *
LOGICAL or LOGICAL*4	long *
LOGICAL*2	short *
LOGICAL*1	signed char *
COMPLEX or COMPLEX*8	struct complex {float realpart, imagpart} *
DOUBLE COMPLEX or COMPLEX*16	struct dcomplex {double realpart, imagpart} *
CHARACTER	signed char * and int (for length)

Table 3 **Passing Arguments from FORTRAN to C**

FORTRAN **CHARACTER** types are a special case. When a C function receives a **CHARACTER** argument by a FORTRAN routine, it receives not only a pointer to the char variable, but also its length, as an **int** (not as an **int ***). This **int** will appear at the end of the argument list. If more than one **CHARACTER** parameter is passed, then an extra **int** for every **CHARACTER** parameter will be passed at the end of the argument list, in the order that the **CHARACTER** parameters are passed. The called C routine must declare one extra variable of type **int** for every FORTRAN **CHARACTER** argument passed in order to receive the information.

For example, a FORTRAN routine calls a C function with two **CHARACTER** parameters and two **INTEGER** parameters:

```
CHARACTER A,B
INTEGER X,Y
CALL NAME(A,X,B,Y)
END
```

The C routine, then, is:

```
name_(char *a, int *x, char *b, int *y, int alen, int
blen)
{}
```

In this routine, the **char *a** points to **CHARACTER A**, **int *x** points to **INTEGER X**, **char *b** points to **CHARACTER B**, **int *y** points to **INTEGER Y**, **int alen** is the length of **CHARACTER A**, and **int blen** is the length of **CHARACTER B**. The extra arguments, **int alen** and **int blen**, appear at the end of the argument list in the order that their corresponding **CHARACTER** parameters were passed (**A** is passed before **B**, so **alen** appears before **blen**).

Although FORTRAN **CHARACTER** string constants are null terminated, **CHARACTER** variables are not. Thus, the character strings **A** and **B** in the above example do not end with an extra 0. However, if the FORTRAN code were changed to the following, the C code could remain the same:

```
CHARACTER A
INTEGER X,Y
CALL NAME(A,X,”this is a string”,Y)
END
```

The string “**this is a string**” will end in an extra zero. However, this 0 will not be counted as part of the string length being passed. So, in the above example, **blen** is 16, not 17.

RETURN TYPES

Called C functions may return values to FORTRAN routines.

SIMPLE RETURN TYPES

An **int** C function must be declared either as **INTEGER (**or **INTEGER*4**) or **LOGICAL** (or **LOGICAL*4**) in the calling FORTRAN routine.

An ANSI C function which returns a **float** must be declared as **REAL** or **REAL*4** in the calling FORTRAN routine.

An ANSI C function which returns a **double** must be declared as **DOUBLE PRECISION** or **REAL*8** in the calling FORTRAN routine.

A non-ANSI C function which returns a **float** or **double** must be declared as **DOUBLE PRECISION** or **REAL*8** in the calling FORTRAN routine.

CHARACTER

Some implementations do not allow functions which return **CHARACTER** types to be written in C. The following description applies only to those implementations, such as Green Hills, which allow this.

A **CHARACTER** type may not be returned directly with a C **return** statement. Instead, when a C function wants to return a FORTRAN CHARACTER result, then two extra arguments are passed to the C function. These arguments appear at the beginning of the argument list. The first argument in the C function must be a **char ***. The character string to be returned should be placed where this argument points. The second argument must be the maximum permitted length of the character string. For example, for:

```
CHARACTER*9 NAME
CHARACTER*9 A
A=NAME()
PRINT*,A
END
```

the C function is:

```
void name_(char *c, int b)
{
    char d[]=”pigeon”;
    int i, len;
    len=strlen(d);
    if (len > b)
                len = b;
    for (i=0; i < len; i++)
                c[i] = d[i];
    for (i=len; i < b; i++)
```

```
                c[i] = ' ';
}
```

In the FORTRAN routine, the function **name** is not called with any arguments. Since the function is declared as a **CHARACTER** return type, two arguments will be automatically passed. The C function receives these as a pointer to the return location (**char *c**) and the length (**int b**) of the character string. The C function does not use the **return** statement.

COMPLEX AND DOUBLE COMPLEX

Some implementations do not allow functions which return **COMPLEX** or **DOUBLE COMPLEX** types to be written in C. The following description applies only to those implementations, such as Green Hills, which do allow this.

COMPLEX (or **COMPLEX*8**) or **DOUBLE COMPLEX** (or **COMPLEX*16**) types may not be returned with a C **return** statement. When a function is declared to be of one of these types, then one extra argument is passed to the C function. This argument will appear at the beginning of the argument list. The C function must declare a special **struct** in which to put the return information. The first argument in the C function must be a pointer to the previously defined **struct**. Table 3 on page 3-61 lists the necessary struct declarations for these two return types. For example, a FORTRAN routine calling a C function with a COMPLEX return type:

```
COMPLEX A
COMPLEX COMP
A=COMP()
PRINT*,A
END
```

can have the C function:

```
struct complex {float realpart, imagpart;};

comp_ (c)
struct complex *c;
{
    c->realpart=1.9;
    c->imagpart=4.5;
}
```

In the FORTRAN routine, the function **comp** is not called with any arguments. Since the function is declared as having a **COMPLEX** return type, one argument will automatically be passed at the beginning of the argument list. The C function receives this argument as a pointer to a **struct** to store the return information in (**struct complex *c**). The C function does not use the **return** statement.

ALTERNATE RETURNS

A FORTRAN routine may call a C function using the alternate return conventions. The C routine would use the return statement in the same way a FORTRAN using alternate returns would, except that instead of the FORTRAN **RETURN**, the C program would use **return 0**. For example:

```
X = 9
Y = 3
CALL COMPARE(X,Y,*100,*200,*300)
PRINT*,'Illegal input'
GOTO 99
100 PRINT*,'X < Y'
GOTO 99
200 PRINT*,'X == Y'
GOTO 99
300 PRINT*,'X > Y'
GOTO 99
99 END
```

The C function could be:

```
compare_(a, b)
float *a, *b;
{
  if (*a < 0.0 || *b < 0.0)
    return 0;
  if (*a < *b)
    return 1;
  if (*a == *b)
    return 2;
  return 3;
}
```

If **compare** returns a 0 in the above example, the next line after the function call will be executed. If 1 is returned, then line 100 will be the next line executed.

SYMBOL NAMING CONVENTIONS

FORTRAN is not case-sensitive and converts all characters (outside of quotation marks) to lower case. In a FORTRAN program, the symbol names **FALCON**, **Falcon** and **falcon** are all the same item. C is case-sensitive. In a C program, the symbols **FALCON**, **Falcon** and **falcon** are three distinct identifiers. So, only C functions whose names are all lower case are called, unless FORTRAN routines are compile with a **-U** option, making FORTRAN case-sensitive.

FORTRAN also appends an underscore (_) to each function name. To call a C function from a FORTRAN routine, the name of the C routine must end in an underscore. For example, instead of naming a C routine **falcon()**, it is named **falcon_()**. This feature allows calling C routines from FORTRAN via an interface routine. The next section explains this in detail.

CALLING C ROUTINES FROM FORTRAN

Because FORTRAN passes function arguments as pointers, it is not possible to directly call pre-compiled C routines that haven't been explicitly written for FORTRAN. FORTRAN appends an underscore to the end of function names to allow an interface routine of the same name. An interface routine could be called from FORTRAN, and would then call the actual C routine with the correct arguments.

For example, for the following pre-compiled C routine:

```
int add(int i, int j)
{
    return i + j;
}
```

it is not possible to call this routine from FORTRAN because **i** and **j** are not pointers. However, with the following interface routine in C:

```
int add_(int *i, int *j)
{
    return add(*i, *j);
```

```
}
```

this routine can now be called from FORTRAN, which in turn calls the real **add** routine. For example:

```
INTEGER ADD
I = ADD(4, 5)
END
```

COMMON BLOCKS

FORTRAN modifies the names of COMMON blocks. All capital letters are converted to lowercase, but the character or characters appended to the name of the common block differ, depending on the compilation mode.

In f77 compatibility mode, a single underscore is appended to COMMON block names. Since this can cause name conflicts between subprogram names and COMMON block names, in VMS compatibility mode, a dollar sign (**$**) is appended instead.

The **-Xvmscommonname** option causes COMMON blocks to be named in the VMS style, with a dollar sign appended. This option can be selected independently of VMS compatibility mode. With this, f77 compatibility mode can be used, and **-Xvmscommonname** can be specified on the command line to name the COMMON block with a dollar sign suffix instead of an underscore.

The **-Xvmscommonname** is usually enabled in VMS compatibility mode. However, f77 style names can be specified while in VMS compatibility mode by specifying **-Z608** on the command line.

An alternate form of VMS style names for environments do not allow dollar signs in names. This is enabled with the **-Xtwounderscore** option and causes two underscores to be appended to COMMON block names instead of one dollar sign. The **-Xtwounderscore** option is ignored unless VMS style COMMON block names are being generated. (**-Xtwounderscore** in f77 mode can be used if **-Xvmscommonname** is specified.)

Mode	Switch	Effect
f77	(default)	block_
	-Xvmscommonname	block$
	-Xvmscommonname -Xtwounderscore	block_ _
VMS	(default)	block$
	-Z608	block_
	-Xtwounderscore	block_ _

Table 4 **COMMON Block Naming Conventions**

CALLING A FORTRAN ROUTINE FROM C

This section shows how to call FORTRAN subroutines from C.

ARGUMENT PASSING

All FORTRAN parameters are passed by reference, so the corresponding argument in the C call must be a pointer of the appropriate type. The table below shows the argument type that C must pass to correspond to the FORTRAN parameter.

C Passes	FORTRAN Receives
float *	REAL or REAL*4
double *	DOUBLE PRECISION or REAL*8
long *	INTEGER or INTEGER*4
short *	INTEGER*2
signed char *	INTEGER*1
long *	LOGICAL or LOGICAL*4
short *	LOGICAL*2
char *	LOGICAL*1

Table 5 **Passing Arguments from C to FORTRAN**

C Passes	FORTRAN Receives
struct complex {float realpart, imagpart;} *	COMPLEX or COMPLEX*8
struct dcomplex {double realpart, imagpart;} *	DOUBLE COMPLEX or COMPLEX*16
char * and int	CHARACTER

***Table 5* Passing Arguments from C to FORTRAN**

For example, to pass an integer variable **a** from C to FORTRAN, pass **&a**.

Passing a **char** argument to a FORTRAN function is a special case. The C routine must pass not only a pointer to the **char** variable, but also its length, as an **int** (not as an **int ***). This **int** must be passed as the last argument. If more than one **char** is being passed by the C routine, then each one will have a separate **int** associated with it. The **int**s must all appear at the end of the argument list, in the same order that their corresponding **char**s appear. For example:

```
extern int falcon_();
main()
{
    char *c1=”pigeon”;
    char *c2=”sofa sofa”;
    int extra=5;
    int len=falcon_(c1, c2, &extra, strlen(c1),
strlen(c2));
    printf(“%d\n”,len);
}
```

This C routine passes two **CHARACTER** parameters and one **INTEGER** parameter to a FORTRAN function. It accomplishes this by passing five arguments. The first two are pointers to **char**s being passed (**c1** and **c2**), the third is the **int** being passed (**extra**), and the last two are the lengths of **c1** and **c2**. The corresponding FORTRAN function is:

```
integer function falcon(a,b,x)
character*(*)a
character*(*)b
integer x
```

```
falcon=len(a)+len(b)+x
end
```

RETURN TYPES

FORTRAN functions may return values to C routines.

SIMPLE RETURN TYPES

An **INTEGER** or (**INTEGER*4**) or **LOGICAL** (or **LOGICAL*4**) FORTRAN function must be declared as **int** in the calling C routine.

A **DOUBLE PRECISION** or **REAL*8** FORTRAN function must be declared as **double** in the calling C routine. Since C usually promotes **float** return values to **double**, a **REAL** return value may not be accessible in C. This is not true for ANSI C, however.

CHARACTER

Some implementations do not allow functions which return **CHARACTER** types to be called from C. The following description applies only to those implementations which do allow this.

FORTRAN functions that have a **CHARACTER** return type are special cases. A value is not actually returned to the calling C routine; instead, the C routine must pass two extra arguments in which to store the return values. The first argument passed must be a **char *** to point to the beginning of the return string. The second argument must be an **int** that is the length of the **char ***. All other normal arguments must follow these two. For example:

```
extern void falcon();
main()
{
    char buff[20];
    char xbuff[]=”pigeon”;

    falcon_(buff, sizeof(buff), xbuff,
sizeof(xbuff)-1);
    printf(“%s\n”, buff);
}
```

Here, two extra arguments are passed, both for a character string being passed. The size of **xbuff** is passed as one short to remove the null character that C will put at the end of the string. The return string will be stored in **buff**. The FORTRAN is then:

```
character*20 function falcon(x)
character*(*) x
falcon=x // ’ sofa sofa’
end
```

This function appends a string to the input string (x), then passes back the new string as the return value.

COMPLEX, COMPLEX*8, DOUBLE COMPLEX, COMPLEX*16

Some implementations do not allow functions which return **COMPLEX**, **COMPLEX*8**, **DOUBLE COMPLEX**, or **COMPLEX*16** types to be called from C. The following description applies only to those implementations which do allow this.

FORTRAN functions that have a **COMPLEX** (or **COMPLEX*8**) or **DOUBLE COMPLEX** (or **COMPLEX*16**) return type are special cases. A value is not actually returned to the calling C routine; instead, the C routine must pass an extra argument in which to store the return value. The first argument passed must be a pointer to a predefined **struct** of the correct type. The return value will be stored in this **struct**. All other arguments must follow this one. For example:

```
struct complex {float realpart, imagpart;};
extern void falcon();
main()
{
    struct complex comp;
    int x=5;

    falcon_(&comp, &x);
    printf(“%f + %fi\n”, comp.realpart,
    comp.imagpart);
}
```

Here, the returned complex number is stored in **comp**, and **x** is an argument being passed. The FORTRAN function is:

```
complex function falcon(x)
integer x
complex y
y=(0.0 , 2.3)
falcon=y+x
end
```

ALTERNATE RETURNS

The FORTRAN alternate return statements return the corresponding integer to the calling C routine (the simple **RETURN** statement returns a 0 to C). The calling C routine makes appropriate use of these return values. Use of a **switch** statement is recommended. There should be a **case** label corresponding to each valid alternate return, and a **default** case to handle all return values outside the expected range. For example:

```
main()
{
   float x, y;
   int ret;

  x = 9;
  y = 3;
  ret = compare_(&x, &y);
  switch (ret)
  {
  default: printf("Illegal input\n");
    break;
  case 1: printf(“x < y\n”);
    break;
  case 2: printf(“x == y\n”);
   break;
  case 3: printf(“x > y\n”);
    break;
  }
}
```

The FORTRAN function would be:

```
SUBROUTINE COMPARE(A,B,*,*,*)
IF (A .LT.  0.0 .OR.  B .LT.  0.0) RETURN
IF (A .LT.  B) RETURN 1
IF (A .EQ.  B) RETURN 2
RETURN 3
END
```

SYMBOL NAMING CONVENTIONS

FORTRAN is not case-sensitive and will convert all characters to lower case. In a FORTRAN program the symbol names **FALCON**, **Falcon** and **falcon** refer to the same item. C is case-sensitive. In a C program, the symbols **FALCON**, **Falcon** and **falcon** are three distinct identifiers. Compiling FORTRAN routines with a **-U** option makes FORTRAN case-sensitive; otherwise, only C functions with lower case names can be called.

FORTRAN also appends an underscore (_) to function names. To call a C subroutine from a FORTRAN routine, the name of the C routine must end in an underscore. For example, a C routine has to be named **falcon_()** instead of **falcon()**. This feature allows calling pre-compiled C routines from FORTRAN via an interface routine, explained in detail in the next section.

COMMON BLOCKS

FORTRAN modifies the names of COMMON blocks. All capital letters are converted to lowercase, but when using a Green Hills FORTRAN compiler, the character or characters which are appended to the name of the common block differ depending upon the compilation mode.

In f77 compatibility mode, a single underscore is appended to COMMON block names. Since this can cause name conflicts between subprograms and COMMON blocks with the same name, in VMS compatibility mode, a dollar sign (**$**) is appended.

The **-Xvmscommonname** option causes COMMON blocks to be named in the VMS style, with a dollar sign appended. This option can be selected independently of VMS compatibility mode. Thus, using f77 compatibility mode and specifying **-Xvmscommonname** on the command line gives the COMMON block a dollar sign suffix instead of an underscore.

Normally, **-Xvmscommonname** is enabled in VMS compatibility mode. However, f77 style names can be specified while in VMS compatibility mode, by specifying **-Z608** on the command line.

An alternate form of VMS style names for environments does not allow dollar signs in names. This is enabled with the **-Xtwounderscore** option and causes two underscores to be appended to COMMON block names instead of one dollar sign. The **-Xtwounderscore** option is ignored unless VMS style COMMON block names are being generated. (**-Xtwounderscore** can be used in f77 mode by specifying **-Xvmscommonname**.)

Mode	Switch	Effect
f77	(default)	block_
	-Xvmscommonname	block$
	-Xvmscommonname -Xtwounderscore	block_ _
VMS	(default)	block$
	-Z608	block_
	-Xtwounderscore	block_ _

Table 6 **COMMON Block Naming Conventions**

CALLING A C ROUTINE FROM ADA

This section shows how to call C subroutines from Ada.

PRAGMA IMPORT

Pragma import specifies that a subprogram is written in some other language, and the definition of that subprogram resides in a separate object module. Pragma import is allowed at the place of a declarative item in a package specification. The subprogram specification for which pragma import is given must appear in the same compilation unit, with the optional *link-name* limited to 62 characters.

For example, to create a link to call C routine *name* in Ada, a package specification has to first be created, containing the Ada declaration of the C routine. The package specification **C_LINK** is:

```
     PACKAGE C_LINK IS
        PROCEDURE Name ;
     PRIVATE
        pragma import(C, Name, ”name”) ;
     END C_LINK:
The corresponding C routine is:
     void name()
     {
        printf(“This routine is called from Ada”);
     }
```

ARGUMENT PASSING

Each parameter in the called C routine must be the appropriate type. The following table shows how the arguments passed by Ada are received by C:

Ada Passes	C Receives
INTEGER	int
INTEGER	long
SHORT_INTEGER	short
CHARACTER	char
BYTE_INTEGER	char
FLOAT	float
LONG_FLOAT	double

Table 7 **Passing Arguments from Ada to C**

The previous example can pass an integer and float to the C routine; it is modified to:

```
     PACKAGE C_LINK IS
        PROCEDURE Name(A_Integer: INTEGER; A_Float: FLOAT) ;
     PRIVATE
        pragma short(C, Name, “name”) ;
     END C_LINK:
The corresponding C routine is:
   void name(int a_integer; float a_float)
   {
     printf(“This routine is called from Ada”);
     printf(“This is an integer passed from Ada %d\n”, a_integer);
     printf(“This is a float passed from Ada %f\n”, a_float);
```

```
}
```

Function calls operate in the same manner as procedures. The Function types must be compatible in C and Ada.

Array and String Types

For Static Ada Array Types, individual components must be structurally compatible to the corresponding C variable. Dynamic Arrays, however, can be passed from Ada to C using the address of the first element:

```
Dynamic_Array(Dynamic_Array'First)'Address
```

In Ada, information is kept in the record regarding bounds of the array.

C strings are terminated by an ASCII null character, ASCII 16#00#. Passing a string to C is much like passing a Dynamic Array, with the exception of appending an ASCII null character to the end of the string.

For example, for an Ada string declared:

```
My_String: STRING(1 . . 8);
```

To pass this to a C string:

```
My_String(My_String'First)'Address
```

Pointers and Address Types

The address convention is identical for Green Hills Ada and C compilers.

Calling an Ada Routine from C

This section shows how to call Ada subroutines from C.

Pragma Export

Pragma Export applies a language-targeted naming convention to a section of a Green Hills Ada program. This allows external access to Green Hills Ada routines and data. Its form is:

pragma Export (*convention-identifier, local-name, external-name*)

where *symbol-form* is the C language. See the Green Hills *Ada 95 Language User's Guide and Reference Manual* for more information on this feature.

For example, to create a link to call C routine *name* in Ada, a package specification must first be created, containing the Ada declaration of the C routine. The package specification, labeled **Export_AdaCode**, is:

```
PACKAGE Export_AdaCode IS
   PRAGMA Names(C);
   PROCEDURE Name;
   PRAGMA Export(Ada);
END Export_AdaCode;

WITH Text_IO;
PACKAGE BODY Export_AdaCode IS
   PROCEDURE Name IS
   BEGIN
      Text_IO.Put_Line("This routine is called
from C");
   END Name;
END Export_AdaCode;
```

The corresponding C routine is:

```
extern name();
void callAdaRoutine()
{
   name();
}
```

When calling Ada Subprograms from a non-Ada task, Pragma **SUPPRESS ALL_CHECK** is recommended if the Ada subprogram exists outside the Ada Runtime; that is, the main program is not in Ada, or the Ada routine is an Interrupt Service Routine, or ISR. Also, results of raising an exception are undefined.

In addition, variables defined in C can be imported by using this method. From the previous example, the variable `global_variable` can be imported from C by declaring:

```
extern name();
```

```
int global_variable;     /* Global Variable used in Ada */
void callAdRoutine()
{
    global_variable= 1;
    name();
}
```

Then, in the Ada package specification:

```
PACKAGE Export_AdaCode IS
    Global_Variable : INTEGER;
    PROCEDURE Name;
    PRAGMA Import(Ada);
END Export_AdaCode;
```

This provides access to **global_variable** for any package body in **Export_AdaCode** package specification.

ARGUMENT PASSING

Each parameter in the called Ada routine must be the appropriate type. The following table shows how the arguments passed by C are received by Ada:

C Passes	Ada Receives
int	INTEGER
long	INTEGER
short	SHORT_INTEGER
char	CHARACTER
char	BYTE_INTEGER
float	FLOAT
double	LONG_FLOAT

Table 8 **Passing Arguments from C to Ada**

Continuing with the first example, the Ada routine can receive an integer and float, modified to:

```
PACKAGE Export_AdaCode IS
   PRAGMA Export(C);
```

```
        PROCEDURE Name(A_Integer: INTEGER; A_Float:
FLOAT) ;
        PRAGMA Export(Ada);
     END Export_AdaCode;
     WITH Text_IO;
     PACKAGE BODY Export_AdaCODE IS
       PACKAGE Flt_IO IS NEW Text_IO.Float_IO(FLOAT);
        PACKAGE Int_IO IS NEW
Text_IO.Integer_IO(INTEGER);
        PROCEDURE Name (A_Integer:INTEGER: ALFloat:
FLOAT) IS
        BEGIN
           Text_IO.Put_Line(“This routine is called
from C”);
          Text_IO.Put(“This is an integer passed from
           C”);
           Int_IO.Put(A_Integer);
           Text_IO.New_Line;
           Text_IO.Put(“This is a float passed from
           C”);
           Flt_IO.Put(A_Float);
           Text_IO.New_Line;
        END Name;
     END Export_AdaCode;
```

The corresponding C routine is:

```
extern name(int a_integer, float a_float);
void callAdaRoutine()
{
   int x;
   float y;
   x = 1;
   y = 10.0
   name(x,y);
}
```

Function calls operate in the same manner as procedures. The function types must be compatible in both Ada and C.

ARRAY AND STRING TYPES

For Static Ada Array Types, individual components must be structurally compatible to the corresponding C variable. Dynamic Arrays, however, can be passed from Ada to C using the address of the first element:

```
Dynamic_Array(Dynamic_Array'First)'Address
```

In Ada, information is kept in the record regarding bounds of the array.

C strings are terminated by an ASCII null character, ASCII 16#00#. Passing a string to C is much like passing a Dynamic Array, with the exception of appending an ASCII null character to the end of the string.

For example, for an Ada string declared:

```
My_String: STRING(1 . . 8);
```

To pass this to a C string:

```
My_String(My_String'First)'Address
```

POINTERS AND ADDRESS TYPES

Address convention is the same for Green Hills Ada and C compilers.

INTERFACING PASCAL AND C

This section shows how to interface Pascal and C:

NAMING CONVENTIONS

By default, the names of Pascal external variables, procedures, and functions are accessible from C functions linked with the Pascal program. External Pascal names are accessed by using the same name in C. Green Hills Pascal is case-sensitive by default; however, using the **-s** or **-Xnocasesensitivity** compile time options make the Pascal compiler case-insensitive, causing it to convert all uppercase character to lowercase. C is always case-sensitive.

When compiling with the **-s** or **-Xappunderscore** option (Strict ISO mode), the names of Pascal external procedures and functions are changed by appending an additional underscore (_). If this option is used, then to call the Pascal function

Falcon from C means calling the function **falcon_**. This is the only function of the **-Xappunderscore** option, while **-s** has many effects.

This option causes all of the C library functions provided with Pascal to become inaccessible.

REDEFINING WRITE OR READ

If a Pascal program redefines the built-in procedure **WRITE** or **READ**, it must be compiled with the **-s** option. The Green Hills C Run-time Library and the UNIX C library use the names **write** and **read** (to which **WRITE** and **READ** are translated by Pascal) for the basic I/O primitives. If the program redefines these names, then very strange results (often infinite loops) occur. The **-s** compile-time option translates these names to **write_** and **read_** instead, so no redefinition will occur. However, under these options communication between Pascal and C or the C Library becomes much more complicated.

C ROUTINES AND HEADER FILES IN C++

The C++ language allows much use of existing C code. Therefore, it is fairly straightforward to call functions written in ANSI C from C++. The syntax of the two languages is very similar and the use of header files has been continued in C++.

By default, the names of functions are encoded or mangled in C++, whereas in C, the names of functions are unchanged. C++ provides the **extern** specifier to identify non-C++ functions so their names will not be mangled. Therefore, this specifier, allows including ANSI declarations for any C functions and then linking with the compiled C object code.

To specify a C declaration:

- Specify or declare functions individually. For example:

```
extern “C” {
        int fclose(FILE *);
        FILE *fopen(const char *, const char *);
}
```

  specifies that two functions with external C linkage are to be declared.

It is easy to include ANSI C in a C++ source. C++ requires prototyped declarations, as does ANSI C. It is not advisable to include non-prototyped declarations since they mean something different in C++. If they are used, any error messages may or may not point to the non-prototype declaration.

- Declare entire header files with **extern**. For example:

```
extern “C” {
        #include <stdio.h>
        #include <string.h>
}
```

has the same effect on all the function declarations that appear in **stdio.h** and **string.h** as the previous example has on the two specific functions (**fclose** and **fopen**).

- If code contains both C and C++, then the **extern** statements can be placed within **#ifdef _ _cplusplus** statements. This practice is common within header files. For example:

```
#ifdef _ _cplusplus
     extern "C" {
#endif
     void assert(int );
             void _assert(const char *,const int
             ,const char *);
#ifdef _ _cplusplus
     }
#endif
```

USING C++ IN C PROGRAMS

Many features in the C++ language simplify complicated tasks, for most languages. It makes little sense to attempt to call C++ from C in most cases, since doing so would force the C programmer to do reproduce work performed by the C++ compiler. The various implementations of C++ use different mechanisms for implementing the following details, making porting of C programs which call C++ more difficult.

Inclusion of C++ modules in C programs is not a trivial. C has no support for any of the C++ extensions to the language. The C programmer must manually perform some of the tasks automatically done by a C++ compiler. Some

knowledge of the internal mechanics and details of a C++ implementation is necessary, as follows:

- Encoding of C++ names can be a problem. The C++ compiler encodes or mangles function and class member names. Any C++ function or class members called from a C program must be referenced by the encoded or mangled names.
- The way member functions are handled by a C++ compiler must be known. All member functions (except static member functions) have the special object member pointer **this** inserted automatically as the first argument in the parameter list. A C programmer must add the argument **this** manually when calling any member functions from C.
- Special processing is needed to handle constructors and destructors for static objects. On most systems the **main** module has special function calls inserted to insure that all static constructor/destructor calls are made properly. If **main** is not in a C++ module, then the C programmer must manually include calls to **_main** in the C main module. The **_main** code is contained in the C++ library and therefore must be linked into the final executable.
- Finally, after the executable is produced, the postlink program must be run, as is the case with any C++ executable. If no static objects are used in the program, this step is not necessary. There are four global objects in the **iostream** library: **cout**, **cin**, **cerr**, and **clog**. (Only **cout** and **cin** are in EC++ and ESTL.) If any of these objects are used in a program, the postlink program must be run on the executable.
- Virtual functions are also handled automatically by a C++ compiler, but involve additional coding to access or use them from a C environment.

FUNCTION PROTOTYPING IN C VERSUS C++

In ANSI C and C++, header files provide prototypes for library functions which enforce a standard interface between the calling program and the called function.

Function prototyping requires that the function declaration include the function return type and the number and type of the arguments. When a prototype is available for a function, the compiler is able to perform argument checking and coercion on calls to that function. If a prototype is not available for a function when it is called, ANSI C will behave like K&R C. The return type of the

function is assumed to be **int**, and actual arguments will be promoted to **int**s, **long**s, **double**s, or pointers as appropriate. In C++, however, it is an error to call a function which has not been declared with a prototype.

Another important difference between ANSI C and C++ is that a non-prototype declaration of a function, such as:

```
char *function_name();
```

has no effect on the number and type of the arguments in ANSI C, but in C++ it is understood as:

```
char *function_name(void)
```

which means that the function has no arguments at all. If the function declaration occurs within the scope of an **extern "C"** declaration, the function has non-C++ linkage and therefore cannot be overloaded. This means that if a traditional K&R style declaration of a function appears in a header file and the **#include** directive which accesses that header file is enclosed in **extern "C" { }**, then it will be impossible to redeclare that function with arguments.

Chapter

4

Writing Portable Code

The C++ language has been implemented by many vendors on a large collection of machines and systems. One important reason for using C++ is to simplify the task of building and maintaining software on multiple platforms. But not all features of the C++ language cater to this goal. C++ intentionally provides features which behave differently on different systems. For C++ programs to be truly portable, the programmer must be careful to avoid these non-portable features of the language.

Certain differences between C++ compilers are vendor specific differences. If all of the C++ compilers you ever use come from a single vendor you can avoid these differences. Many more of the differences however are particular to the processor and the operating system in use. When porting between different platforms it may be impossible to avoid these differences, except by careful coding.

COMPATIBILITY BETWEEN GREEN HILLS COMPILERS

All Green Hills C++ compilers follow the same interpretation of the C++ language, as described in this manual. There are a few features and options which are not available in all Green Hills C++ compilers. These features and options are described in the Development Guide for each compiler. To ensure a C++ program is portable between all Green Hills C++ compilers, we recommend using only those features and options described in this manual and the texts mentioned in the preface. Command line options may also be needed to adjust for differences between the default behavior of each Green Hills C++ compiler.

For information on compatibility with Green Hills compilers for other languages, see Chapter 3, “Mixing Languages”.

WORD SIZE DIFFERENCES

Green Hills compilers are available on machines with 32-bit and 64-bit word size. Other C++ compilers have been written for machines with other word sizes. Porting C++ programs between machines of different word sizes requires particular care because most primary data types can be effected by word size.

RANGE OF REPRESENTABLE VALUES

The size of each basic numeric type controls the range of values which may be represented by that type. The header files **limits.h** and **float.h** provide defined symbols which represent the minimum and maximum values for all numeric

data types in C++. A portable program should use these symbols and never depend on the use of values outside the allowed range.

If arithmetic operations cause overflow, underflow, or loss of precision, the program may not detect the error or may behave differently on different systems.

Relative Sizes of Data Types

C++ places very weak requirements on the relative size of the basic types, but it is not unusual for C++ programs to assume otherwise. For example, C++ only requires that short be no larger than int and that int be no larger than long. It would be legal for short, int, and long to all be the same size or for them all to be different. With all 32-bit Green Hills C++ compilers short is 16 bits and int and long are 32 bits. To assume int is twice as large as short, but the same size as long is non-portable. With 64-bit Green Hills C++ compilers, short is 16 bits, and int and long are either 32 or 64 bits. You can only be assured that long is not smaller than int.

Another common non-portable assumption is that pointers are the same size as int or long. Neither is guaranteed. With all 32-bit Green Hills C++ compilers, pointers are 32 bits. But with 64-bit Green Hills C++ compilers, pointers may be either 32 or 64 bits, independent of the size of int or long.

In C++, all integer constants have type **int** unless marked with a type suffix. In certain cases the use of a plain integer constant instead of a long integer constant can be non-portable.

Byte Order Problems

Since the success of the IBM/360, byte machines have been more popular than word machines. The advantage of byte machines is their efficient processing of character data. The general acceptance of byte machines has led to easier program portability between machines.

There is, however, one major portability problem between byte machines. The first successful byte machine, the IBM/360, placed the most significant byte of a multiple byte integer value at the lowest address. Many byte machines such as MC68000, RS/6000, SH-7000, and SPARC have followed the IBM convention. The second successful byte machine, the PDP-11, placed the least significant

byte of a multiple byte integer value at the lowest address. Intellectual descendants of the PDP-11, such as the VAX, and i386/i486/Pentium and some RISC processors, such as Clipper and V800, have followed the DEC convention. These two groups seem to be so well entrenched that no agreement on byte ordering is possible. A further complication arises because some processors, such as the i960, M88000, PowerPC, R4000, and Weitek-XL support both byte orders, although a given system is normally built to use only one byte order.

Between machines with different byte ordering, programs which overlay characters and integers in memory or which use character pointers to integer variables and vice versa are often not portable.

Programs that declare a single variable with different integer types in different modules may fail when ported to a machine with a different byte order.

ALIGNMENT REQUIREMENTS

Some systems will not load or store a 2 byte object unless that object is on an even address. Other systems have a similar requirement for 4 or 8 byte objects. Others may allow certain accesses, but require more time to perform them. Therefore, alignment of data is both a matter of correctness and time efficiency. Although increased alignment may improve performance, it also consumes space, due to padding inserted to achieve alignment.

The alignment requirements on each system are chosen both to satisfy the restrictions of the hardware and to achieve a reasonable balance between performance and space. The alignment rules for each system differ and often are not configurable. Therefore programs that make assumptions about the relative position of data objects in memory or elements within classes or arrays are not portable, even among the Green Hills C++ compilers.

The C++ language imposes these restrictions on size and alignment:

- The alignment of a class or array is equal to the maximum alignment requirement of any of its members.
- The size of a class or array is always a multiple of the maximum alignment requirement of any of its members.
- The offset of any member of a class or array is always a multiple of its alignment requirement.

- All dynamic memory allocation routines provided with the compiler will return a pointer aligned to the maximum alignment for any object on that machine.

All Green Hills C++ compilers also satisfy these principles:

- The stack is maintained on an alignment suitable for any object.
- Parameters and local variables are allocated on the stack according to their alignment requirement.
- Local variables are arranged on the stack to avoid unnecessary padding due to alignment.

If a program does not use integer arithmetic for pointer computations and ensures that all general purpose memory allocation routines return maximally aligned pointers, then all references to dynamically allocated memory will be properly aligned.

CLASSES AND BIT FIELDS

The preceding issues of size, byte order, and alignment all effect the allocation of data in memory. In particular, compound data structures such as classes, bit fields and arrays are very much effected by them.

UNIONS

A union in C++ allows the same memory location to be accessed as more than one type. This is inherently non-portable. Suppose a union consists of an integer and an array of four characters. Whether the first element of the array is the most significant part of the integer or the least depends on byte order. It is not even certain that the integer and the array of character have the same size.

These problems increase when integer, floating point and pointer fields are combined and are even more severe when structures or bit fields are members of unions.

CLASSES

Green Hills C++ always allocates fields in a class in the order in which they are declared.

The exact offset of each field from the base of the class depends on the size and alignment of the field itself and of those which precede it. The offset of the first field is always 0, but padding is inserted as necessary to satisfy the alignment requirement of each subsequent field, and may also be added at the end of the class to make its overall size a multiple of its alignment.

Any program which assumes the offset of a field within a class or which assumes that certain fields in two different classes always have the same offset are non-portable.

BIT FIELDS

The allocation of bit fields in a structure is very dependent on alignment rules. In addition, the exact layout of bits within a bit field varies between systems and cannot be assumed by a portable program.

CHARACTER SET DEPENDENCIES

Not all computer systems use the same characters. All computer systems recognize letters, digits, and the standard punctuation characters. But there is considerable variation among the less commonly used characters. Therefore, programs which use the less common characters may not be portable.

Your Green Hills compiler uses the ASCII character set and the ASCII collating sequence. Some language implementations use a different collating sequence, such as EBCDIC.

Programs which manipulate character data, especially string sorting algorithms, may be dependent on a particular character collating sequence. The collating sequence is the order in which characters are defined by the implementation. If one character appears before a second character in the collating sequence, then the first character will be “less than”' the second character when they are compared. In the ASCII collating sequence, the lowercase letters “a” to “z” appear as the contiguous integer values 97 to 122 (decimal). In other collating sequences, such as EBCDIC, the lowercase letters are not contiguous.

To make character and string sorting programs portable, care must be taken to avoid dependencies on the character collating sequence. If a program is designed to operate with a collating sequence other than ASCII, it may be

necessary to modify string and character comparison code to operate with ASCII.

FLOATING POINT RANGE AND ACCURACY

One of the most variable aspects of different machines is floating point arithmetic, where the range, precision, accuracy and base can vary widely. This can lead to many portability problems which can only be addressed numerically. Your Green Hills compiler uses IEEE floating point representation.

OPERATING SYSTEM DEPENDENCIES

Programs which access operating system resources, such as files, by their system names are often not portable. The file and I/O device naming conventions vary greatly among computer systems. In order to write portable programs it is necessary to minimize the use of explicit file names in the program. It is best if these names can be input to the program when the program is run.

If a program contains explicit file names it may be necessary to change them to names acceptable to the target system. Refer to your target operating system documentation for a description of legal file names for your environment.

ASSEMBLY LANGUAGE INTERFACES

Programs which use embedded assembly code or interface to external assembly will require all of the assembly code to be redone when the program is transported to a new machine.

EVALUATION ORDER

None of the language specifications fully specify the order in which the various components of an expression or statement must be evaluated, and they disallow computations whose results depend on which permitted evaluation order is used. Many illegal programs have gone undetected because they have only been compiled with one compiler. Since your Green Hills compiler's evaluation order may not be identical to the evaluation order of other compilers, some of these illegal programs which operate as expected with another compiler may not operate the same way when compiled with your Green Hills compiler.

Some language implementations may evaluate the arguments to a function from right to left, others from left to right.

Expressions with side effects, such as subroutine, procedure, or function calls, may be executed in a different order by your Green Hills compiler and other compilers. When a variable is modified as a side effect of an expression and its value is also used at another point in the expression, it is not defined whether the value used at either point in the expression is the value before or after modification. Different values for the same variable could potentially be used at different places in the expression depending on the order the compiler chose for evaluation.

The operators ++, --, +=, etc., may be executed in a different order by your Green Hills compiler and other compilers.

Your Green Hills compiler may allocate some pointer variables not declared **register** to registers. This may allow the compiler to generate more efficient sequences for post increment operators than other compilers. These sequences may involve incrementing at a different position in the statement than with other compilers. In particular, statements of the form:

***p++** = *expression involving p*

often evaluate differently under PCC than they do with a Green Hills compiler.

A particular case of evaluation order dependency is the use of the **?:** operator in an expression which is an argument to a function call. Your Green Hills compiler evaluates all question mark operators before any other arguments, and keeps the result in temporary storage. PCC evaluates the **?:** operator at its position in the argument list. The call:

```
foo(b?i:i+i, i++)
```

will usually evaluate differently under PCC than under your Green Hills compiler.

MACHINE-SPECIFIC ARITHMETIC

Certain arithmetic operators in C++ are intended to generate the most efficient corresponding operation on the target machine. If all input values are within the

expected range, the results are portable, but out of range values may give different results on different systems.

SHIFT

The shift operators in C++ have this characteristic. If the right-hand operand is negative or exceeds the number of bits in the left-hand operand the behavior is undefined. In Green Hills C++ compilers, the operands will be given to the hardware as if the operands were legal and the result depends entirely on the hardware. Some systems accept a negative shift and reverse the direction of the shift, but many do not. Shifting by more than the number of bits is the same as shifting by 1 less than the number of bits on some systems, but on others it has very different results.

If the left-hand operand of a right shift is signed, C++ does not require the compiler to propagate the sign bit. That means a correct C++ compiler is allowed to yield a positive number when right shifting a negative number by one.

DIVISION

The division operator may round up or down when applied to signed integers if one or both of them is negative. Division by 0 produces different results on different machines.

The remainder operator always satisfies the rule

```
( a / b ) * b + a % b == a
```

as long as **b** is not 0. Therefore if **a** or **b** is negative, the sign of the remainder may or may not match the sign of the dividend, depending on the machine.

ILLEGAL ASSUMPTIONS ABOUT COMPILER OPTIMIZATIONS

Some programs illegally depend on the exact code that some particular compiler generates. Such programs are particularly difficult to port to an advanced optimizing compiler, such as your Green Hills compiler, because the optimizer makes major changes in the code in order to make the program smaller and/or faster. Described below are some of the most common illegal assumptions made about code generation. Please familiarize yourself with the optimizations described in Chapter 5, “Optimization”, before reading further.

IMPLIED REGISTER USAGE

Some programs rely on the exact register allocation scheme used by the compiler. Such programs are completely illegal, and will never transport without modification.

For example, C++ programs that rely on register variables being allocated sequentially to pass hidden parameters will not work. Hidden returns (i.e. using **return** and expecting to return the value of the last evaluated expression) will not work either.

MEMORY ALLOCATION ASSUMPTIONS

Memory is allocated by your Green Hills compiler in a different way than by the industry's standard compilers and other companies' compilers. This can cause problems in porting programs which illegally depend on the memory allocation peculiarities of other compilers:

- Some programs depend on the compiler allocating variables in memory in the order that they are declared. Your Green Hills compiler will not necessarily allocate variables in the order of declaration.
- Some programs depend on knowing that the compiler will allocate all variables even if they are not used. Your Green Hills compiler may not allocate unused variables.
- Some programs depend on knowing that certain variables will be allocated in memory. Your Green Hills compiler will allocate certain variables to registers that the standard compilers would always allocate to memory.

Programs compiled with your Green Hills compiler must not make assumptions regarding the order or allocation of variables in memory (except where the language standard specifies it).

MEMORY OPTIMIZATION RESTRICTIONS

READ THIS SECTION CAREFULLY IF YOU ARE PORTING SYSTEM CODE OR APPLICATIONS THAT USE SHARED MEMORY OR SIGNALS.

Using the command line option **-OM** will enable the compiler to assume that memory locations do not change asynchronously with respect to the running program. In particular, when the compiler reads or writes some memory

location, it will assume that the same value is still there several instructions later. To avoid the (potentially high) speed penalties involved in re-reading memory, the compiler will attempt to find a copy of the value which is itself still in a register, and use that instead.

This can easily cause problems for many parts of operating systems, device drivers, memory mapped I/O locations, shared memory environments, multiple process environments, interrupt driven routines, and when UNIX style signals are enabled. In C++, general optimizations may be used as described in the next section.

MEMORY OPTIMIZATION IN C++

An example of potential problems with memory optimizations is that many UNIX device drivers need to use memory locations which are really I/O registers that can change at any time. A typical example of a loop waiting for a device register to change is:

```
while (!(*TSRADDR & (1 << TXSBIT)));
```

If memory optimizations are enabled while compiling this loop, the compiler may generate code that reads the value pointed to by **TSRADDR** only once. With **-OLM**, it is almost certain that this will be the case. When this happens, the loop will execute either once or forever, depending on the value of the bit when it is first tested, and the loop will be rendered either ineffective or fatal.

Depending on the situation, the compiler may be able to detect loops like the above, and generate code that operates correctly even with **-OM** set. However, if the loop body were to test more than one bit at the same address, the compiler will contort the loop in an attempt to read memory as few times as possible.

The compiler assumes that you will use the **volatile** type qualifier when it is available. This means that **-O** always implies **-OM** in C++. If, for some reason, you are unable to use **volatile**, and this is a real problem, you can add the option **-Onomemory** to your command line to force memory optimization off. Note that **-Onomemory** also implies **-O**.

PROBLEMS WITH SOURCE LEVEL DEBUGGERS

This section describes various problems relating to source level debuggers.

VARIABLE ALLOCATION

Once a variable is allocated to a register it will always reside in that register. However, since other variables may share the register, it may not always contain the current value of the variable. This may cause a source level debugger to give incorrect results. If you ask for the value of a variable at a point outside the range of its use, the compiler may have temporarily allocated that register for some other purpose. Always check results just after they are assigned, or when the current value is going to be used later. Near the end of a function most of the local variables will no longer be in use, so it is more likely that the register has been reallocated.

ADVANCED OPTIMIZATIONS

In general, Green Hills recommends that all optimizations be turned off if source level debugging is to be performed. The following are examples of specific problems that can be caused when optimizations are used in conjunction with source level debuggers.

- The common subexpression elimination optimization causes the compiler to try to precalculate expressions which are used more than once and save the result in a register. During debugging, the programmer will not find the expression itself, since it was evaluated and saved at an earlier time.
- Various loop and branch optimizations rearrange entire statements or blocks of statements causing difficulties with source level debugging since there will no longer be a direct correlation between source lines and executable instructions.

PROBLEMS WITH COMPILER MEMORY SIZE

Your Green Hills compiler is an advanced optimizing compiler. It is much better than the current generation of “optimizing” microprocessor compilers. In accordance with its greater capability, it requires more memory. The compiler requires 1 megabyte of memory just for the program. It is designed to work best when 2 megabytes or more of memory are available. It will run in less memory but with some degradation of performance or capability.

The compiler’s primary use of memory is for the program, static data structures, global declarations, parse trees, and generated machine code. Global declarations consist of the global constant, type, variable, and function

declarations. Memory usage increases when large numbers of declarations are included in a compilation. Even unused global declarations must be stored throughout the compilation. If memory size problems exist, try to reduce the size of the include files by including just the declarations that are needed.

Memory is also needed for basic blocks. Every possible branch creates a new block. Machine generated programs with very large **switch** statements or a very large number of small **if** statements may increase memory usage.

Your Green Hills compiler is a one pass compiler. That is, it reads the source program only once. Each function is converted into a parse tree as it is read. When the end of the function is reached, the optimizer is called with the parse tree as input. The optimizer modifies the parse tree and then passes it on to the code generator. The code generator produces an internal representation of the machine code to be output for the function. Another optimization phase is then called to modify this machine code. Finally the optimized machine code for the function is output. After the machine code is output, the memory being used for the parse tree and machine code is released for use in compiling the next function.

The maximum memory usage for parse trees and machine code is determined by the size of the largest function in the program. If memory size problems exist, turn off the optimizer and reduce the size of the largest function. A simple function of less than 100 lines should not cause memory size problems. However, procedures which are more than 1000 lines, or contain very complex statements, can require several megabytes of memory to compile.

Chapter

5

OPTIMIZATION

Along with providing standard optimizations available with other compilers, the Green Hills compiler supports an advanced set of optimizations. Among these optimizations are specialized suboptions which allow you to target specific types and areas of code for improved performance.

This Chapter describes the Green Hills compiler optimizations under three categories:

- Optimizations performed by default
- General optimizations enabled with the **-O** option
- Specialized optimizations enabled with the suboptions **-OALMI**

DEFAULT OPTIMIZATIONS

This section describes the optimizations that the compiler performs by default, when no options are set:

- Constant Folding
- Register Allocation by Coloring
- Register Coalescing
- Loop Rotation

CONSTANT FOLDING

Constant folding optimization is performed when the compiler can determine at compile-time that an expression is a constant. The compiler substitutes the constant for any reference to the constant expression.

In these examples, the constant expression INT_MAX/2 with its value, 16383.

Examples:

Initial C source code:

```
#define INT_MAX 32767
short subr(){
    int x;
    x=INT_MAX/2;
    return(x); }
```

Optimized C source code:

```
short subr(){
    int x;
    x = 16383;
    return(x); }
```

REGISTER ALLOCATION BY COLORING

Register allocation by coloring is used to permanently maintain a selected set of local scalar variables in registers based on their frequency of reference and their lifetimes. During program compilation, the optimizer uses data flow analysis to determine the lifetime of each variable. The register allocator also uses this information to assign different variables within a function to the same register if the lifetimes of the variables do not overlap. This increases the opportunity for allocating variables to registers.

With the local variables preallocated to registers, the compiler can optimize the code significantly, since additional memory load and store instructions are not required to reference the variables.

In these examples, the variables **a** and **b** are both assigned to the same register since their lifetimes do not overlap (note that the code could be optimized still further, but is left as is to simplify the examples).

EXAMPLES:

Initial C source code:

```
int subr(x)
int x;
{
    int a,b;
    a=x;
    b=x*2;
    return b;
}
```

Optimized C source code:

```
int subr(x)
int x;
{
    int a;
    a=x;
    a=x*2;
    return a;
}
```

For small functions, the compiler maintains all local variables in registers. Scalars generally are considered for register allocation unless their values are accessed with the address operator (**&**). This optimization is disabled with the **-nooverload** option.

REGISTER COALESCING

With register coalescing optimization, the optimizer uses the destination register as a work register when evaluating the associated expression and organizes the instruction sequence so the result ends up in the destination register. This optimization eliminates the additional register-to-register copies required when using a temporary register.

EXAMPLES:

Initial C source code:

```
int fun(a,b,c)
int a,b,c;
{
    int ret = a+b+c;
    return ret;
}
```

Optimized C source code:

```
int fun(a,b,c)
int a,b,c;
{
    return a+b+c;
}
```

LOOP ROTATION

Loop rotation optimization refers to locating the termination test and a conditional branch at the bottom of the loop. Therefore, the loop only processes one branch instruction on each iteration. Most compilers place the termination test and an unconditional branch at the top of the loop and an additional unconditional branch at the bottom.

EXAMPLES:

Initial C source code:

```
int subr(i)
int i;
{
    while (i < 10)
    i *= i;
    return(i);
}
```

Optimized C source code:

```
int subr(i)
int i;
{
    goto L7;
    do {
    i *= i;
    L7:
    } while (i < 10);
    return(i);
}
```

In addition, if the compiler can determine that the loop is executed at least one time, the loop is entered at the top. If not, the compiler generates an unconditional branch at the top of the loop to the termination test.

GENERAL OPTIMIZATIONS ENABLED WITH THE -O OPTION

General optimizations are enabled with the **-O** option. When **-O** is selected, all of the following optimizations are performed:

- Pipeline Instruction Scheduling
- Static Address Elimination
- Peephole Optimization
- Common Subexpression Elimination
- Tail Recursion
- Dead Code Elimination
- Constant Propagation

Certain **-O** optimizations can be controlled with **-Ono** options, each of which disables a specific **-O** optimization but enables all others. For example, the **-Onocse** option enables all **-O** optimizations except for common subexpression elimination. These options are described in the appropriate optimization sections.

STATIC ADDRESS ELIMINATION

With static address elimination optimization, the optimizer assigns frequently used static variables to registers within the scope of the function. This optimization eliminates the loads and stores required with memory allocation. It is enabled with the **-OM** option.

In these examples, the address of the static variable **x** is maintained in register.

EXAMPLES:

Initial C source code:

```
int subr(q)
int q;
{
    static int x=0;
    x++;
    q+=x;
    return(q);
}
```

Optimized C source code:

```
int subr(q)
int q;
{
    static int x=0;
    register int x_ = x;
    x_++;
    q+=x_;
    x=x_;
    return(q);
}
```

Note that this optimization is performed not only for locally defined static variables, but also for global variables, as shown in the following example:

Initial C source code:

```
int x = 0;

int subr(q)
int q;
{
    x++;
    q+=x;
    return q;
}
```

Optimized C source code:

```
int x=0;

int subr(q)
int q;
{
    register int x_ = x;
    x_++;
    q+=x_;
    x=x_;
    return(q);
}
```

PEEPHOLE OPTIMIZATION

Peephole optimization identifies common code patterns and replaces this code with more efficient code patterns. This includes optimizations such as removal of unreachable code, flow of control and algebraic simplifications. The compiler only performs this optimization when local code analysis insures that the results will be correct without further analysis of the surrounding code. This optimization is disabled with the **-Onopeep** option.

EXAMPLES:

Initial C source code:

```
int subr(x,y,z)
int x,y,z;
{
    y = x;
    z = y;
    return z;
}
```

Optimized C source code:

```
int subr(x,y,z)
int x,y,z;
{
    return x;
}
```

COMMON SUBEXPRESSION ELIMINATION

Common subexpression elimination is performed when a previously calculated expression is part of a later expression and none of the variable values in the subexpression have changed. The optimizer retains the value of the subexpression in a register for reuse. This optimization is disabled with the **-Onocse** option.

EXAMPLES:

Initial C source code:

```
int subr(x,y)
int x,y;
{
    int a, b;
    x += a+b;
    y += a+b;
    if (y < 0)
    return(y);
    return(x);
}
```

Optimized C source code:

```
int subr(x,y)
{
    int a, b, _v6;
    x+=(_v6=a+b);
    y+=_v6;
    if (y<0)
    return y;
    return x;
}
```

TAIL RECURSION

A procedure is considered tail recursive if the last statement executed is a procedure call to itself followed by a return statement. This is sometimes simply called a recursive procedure. Tail recursion optimization replaces the procedure call with a branch instruction and eliminates the return statement.

EXAMPLES:

Initial C source code:

```
int sum(n)
int n;
{
    if (n <= 1
```

```
    return(1);
    else
    return(n+ sum(n-1));
}
```

Optimized C source code:

```
int sum(n)
int n;
{
    int _v3=0;
L1:
    if (n <= 1)
        return _v3+1;
    _v3 += n;
    _n--;
    goto L1;
}
```

DEAD CODE ELIMINATION

With dead code elimination, the optimizer does not generate assembly code for statements computing values that are never used and therefore have no effect on the program results.

In this example, the optimizer eliminates all code for processing the variable **a** since it knows at compile-time that the variable **a** is zero and therefore any code referencing it is not used.

EXAMPLES:

Initial C source code:

```
#define F0 0
#define F2 2
int subr(x)
int x;
{
    int a,b,c;
    a=F0*x;
    b=F2*x;
```

```
    return ((a)? a : b);
}
```

Optimized C source code:

```
int subr(x)
int x;
{
    int b;
    b=2*x;
    return(b);
}
```

CONSTANT PROPAGATION

Constant propagation is the replacement of one or more variables with constants over the course of a variable's lifetime if the variable's value is known and does not change during that lifetime. The following simple examples show code optimized with constant propagation:

EXAMPLES:

C source code:

```
main()
{
    int i,a,b;
    a = 3;

    for (i=0;i<1000;i++)
    b += a;    /* a is constant over the lifetime of the loop */
    printf("%d\n", b);
}
```

Optimized C source code:

```
main()
{
    int i,b;
    for (i=0; i<1000; i++)
    b+=3;
    printf("%d\n", b);
}
```

SPECIALIZED OPTIMIZATIONS SET WITH THE SUBOPTIONS -OLAMIS

The specialized optimizations are enabled using the **-OL**, **-OA**, **-OM**, **-OI**, or **-OS** options. These optimizations enable the general optimization along with the indicated suboptions. The optimizations provided by each option are as follows:

-OL	Loop Optimization: Strength Reduction Loop Invariant Analysis Loop Unrolling
-OA	Algorithmic Optimization
-OM	Memory Optimization
-OI	Inlining Optimization
-OS	Size Optimization

You can combine these suboptions (**L**, **A**, **M**, **I** and **S**) in any order by appending them to the **-O** option. For example, the **-OLAMIS** option turns on all optimizations.

LOOP OPTIMIZATION WITH -OL

Loop optimization is selected with the **-OL** option. This option informs the compiler that most computation is performed within the innermost loops. Therefore, the compiler focuses most of the available machine resources on optimizing that portion of code.

The following loop optimizations are performed:

- Strength Reduction
- Loop Invariant Analysis
- Loop Unrolling.

You can also list specific functions for this optimization using the following syntax:

-OL=*func1*,*func2*,...,*funcn*

The **-Onounroll** and **-Ounroll8** options can be used with **-OL** to affect loop unrolling. See the section on Loop Unrolling, below.

STRENGTH REDUCTION

Strength reduction optimization is applied to arrays subscripted with the loop index. Most compilers access the array element by multiplying the size of the element by the loop index. The Green Hills compilers store the address of the array in a register and add the size of the array element to the register on each iteration of the loop.

EXAMPLES:

Initial C source code:

```
subr()
{
    int i;
    int q[4];
    for (i=0;i<4;i++)
    q[i]=i;
}
```

Optimized C source code:

```
subr()
{
    int i;
    int q[4];
    int *_ptr;
    for (i=0, _ptr=q; i<4; i++)
    *_ptr++ = i;
}
```

Strength reduction also applies to multiplying a loop invariant with the loop index. The optimizer replaces a multiply instruction or a call to the **mul()** library function with add and shift instructions.

LOOP INVARIANT ANALYSIS

Loop invariant analysis is used to enhance loop performance. Each loop is examined for expressions or address calculations that do not change within the loop. These computations are located outside the loop and their values are stored in registers.

This optimization is particularly valuable for reducing the code generated to access an element of an array when the array index does not change within the loop.

EXAMPLES:

Initial C source code:

```
subr()
{
    int i,j;
    int q[4],p[4];
    for (i=3;i>=0;i--)
    q[i]=i;
    for (j=0;j<4;j++)
    p[j]=q[i];
}
```

Optimized C source code:

```
subr()
{
    int i,j;
    int q[4],p[4];
    int *_ptr;
    for (i=3; i>=0; i--)
    q[i] = i;
    for (j=0, _ptr = &q[i]; j<4; j++)
    p[j] = *_ptr;
}
```

LOOP UNROLLING

With loop unrolling optimization, the compiler duplicates the code in the innermost loop up to a maximum of four times by default. This optimization produces more straightline code, which removes much of the loop overhead in testing for stop condition and branching. This allows better use of the register allocator and more opportunity for instruction pipelining. It is most effective when the innermost loop is relatively short causing minimal increase in code size.

There are two options that can be used along with **-OL** to affect loop unrolling. **-Ounroll8** allows loops to be unrolled up to 8 times instead of the default maximum of 4 times. **-Onounroll** disables loop unrolling but enables the other **-OL** options.

The following simple examples use a constant loop size of 100 and a maximum loop index of four to show the effect of this optimization.

EXAMPLES:

Initial C source code:

```
subr(a)
int a[];
{
    int i;
    for (i=0;i<100;i++)
    a[i]=i;
}
```

Optimized C source code:

```
subr(a)
int a[];
{
    int i;
    for (i=0;i<100;i+=4) {
    a[i]=i;
    a[i+1]=i+1;
    a[i+2]=i+2;
    a[i+3]=i+3;
    }
}
```

Calling the size of the loop **n**, suppose that **n** is large (auxiliary loop execution time is negligible); then, the original loop takes **n***(4 cycles per iteration) == 4**n** cycles to complete. The unrolled loop takes **n**/4*(10 cycles per iteration) == 2.5**n** cycles to complete. With **n** large, the unrolling has the effect of making the loop execute in only 63% of the time required by the original loop.

ALGORITHMIC OPTIMIZATION WITH -OA

These optimizations assume the program implements a portable algorithm which is not affected by the limitations of finite hardware. For example, these optimizations may apply algebraic properties such as associativity without respect to the possibility of overflow, underflow, round-off, loss of precision, or division by zero.

Furthermore, these optimizations assume that the algorithm never makes use of the characteristics of two's complement integer arithmetic or IEEE floating point arithmetic beyond that implementation independent rules of ANSI C. For example, ANSI C states that the size of an **int** is implemenation defined and in most environments supported by Green Hills compilers, an **int** is a 32-bit two's complement number. For example, any program that depends on an **int** having exactly 32 bits, rather than 35 bits, or which depends on two's complement arithmetic rather than signed magnitude or some other representation should NOT be compiled with -OA.

For example,

```
unsigned char c = -1;
if (c == 255)
        foo(); /* with -OA this might not be called */

signed char s = -127;
if (c - 5 > 0) /* note that c-5 yields 4 because of overflow */
        bar(); /* with -OA this might not be called */
```

In ANSI C, the include file "**limits.h**" provides implementation defined bounds of all integral types. Any code which depends on the result of an arithmetic operation which exceeds these bounds should not be compiled with **-OA**.

Some programs achieve portability by intentionally forcing overflow in order to determine the limitations of the hardware. The results of these tests are then used to avoid overflow in the rest of the program. These overflow tests should NOT be compiled with **-OA**.

ALGEBRAIC ALGORITHMIC OPTIMIZATION

With some systems there is an additional type of algorithmic optimization that can be enabled with the **-X915** option (note that **-OA** must also be specified for this to work). With this optimization, whenever the compiler finds a multiply

across an add, such as **X*(Y+Z)**, where **X** is a constant, it will distribute the multiply across the add, so our previous example would become: **X*Y+X*Z**. Even though this actually increases the number of calculations performed (from two to three) it can actually increase the speed of the calculation due to better register usage on some systems.

MEMORY OPTIMIZATION WITH -OM

Memory optimization is enabled with the **-OM** option. This allows the compiler to optimize repeated memory reads by placing the value in a register. Subsequent read operations then refer to the register rather than the actual memory location. With this optimization the compiler assumes that memory locations only change with explicit store instructions and therefore are not affected by any external sources.

It is therefore not recommended for applications in which memory could by externally affected: device drivers, operating systems, and shared memory. This also applies in a non-virtual memory environment when interrupts are enabled.

The **-OM** option is automatically set with the **-O** option in full ANSI or 90% ANSI mode (the **-ANSI** or **-ansi** options), since the **volatile** keyword is defined to explicitly identify objects that may change without the compiler's knowledge or control. If you wish to want to use **-O** without using **-OM** in one of these modes, you may use the **-Onomemory** option. This option turns on **-O**, but turns off memory optimization.

SPACE OPTIMIZATION WITH -OS

Space optimization is enabled with the **-OS** option. This tells the compiler to perform all default and general optimizations that would increase efficiency but not greatly increase code size. For instance, if you compiled your code with the optimization option **-OSL**, the compiler would omit the loop unrolling phase.

INLINING WITH -OI

The term "inlining" refers to the process of substituting the contents of a function or subroutine in place of the call to that function or subroutine. The resulting code is faster, since the overhead of a jump-to-subroutine call has been eliminated. Typically, a small function or subroutine that is frequently executed,

but is called from only a few locations within the program, is the best candidate for inlining. This way, the maximum benefit can be obtained by increasing efficiency in high usage areas, while not significantly increasing program size. This feature is currently supported with C++ with the following limitations:1) the **-OI**=*function-name* style of inlining can't be used with C++, and 2) the **-OI** option can't be used. C++ does all of the inlining that is built into the language. In addition, there is a **--max_inlining** option which will be more aggressive in inlining. See your release notes for more information.

The following program illustrates the basic principles of inlining. The main program in this case contains a simple loop which calls the function **sub()**. The call itself occurs only once in the program code, but the function is executed for each iteration of the loop. The call is easily replaced by the routine code for **sub** itself, eliminating both the need for parameter passing and the overhead of a jump-to-subroutine. The reduced overhead per execution becomes a major savings in program speed.

EXAMPLES:

Initial C source code:

```
_ _inline sub(x) {
    printf("x=%d\n",x);
    return;
}
main() {
    int i;
    for (i=1;i<10;i++)
    sub(i);
}
```

Optimized C source code:

```
 sub(x) {
    printf("x=%d\n",x);
    return;
 }
 main() {
    int i;
    for (i=1;i<10;i++)
    printf("x=%d\n",i);
```

```
}
```

Note that the code for **sub** has not been eliminated, although the main program no longer contains a call to **sub**. The compiler generates code for each function, whether or not it is inlined, so that it will be available to be called from other modules and so that its address can be taken. While the size of the actual generated code was not changed significantly, the execution speed of the main program was improved by eliminating the jump-to-subroutine overhead.

USING THE INLINER

The Green Hills implementation of inlining is language independent within the Green Hills family of compilers. Routines of one language may be freely inlined into programs of another language. Also, inlining is performed across modules: if a function **foo()** to be inlined is defined in one module but used in several, the compiler will be able to inline **foo()** in all the modules in which it is used.

For the sake of brevity, the word “function” in the following sections on inlining is used to apply to FORTRAN subroutines as well.

SELECTING FUNCTIONS TO BE INLINED

There are three methods for selecting the functions to be inlined:

Manual Inlining

The **_ _inline** keyword may be inserted in the source code immediately before the declaration of each function to be inlined. This is referred to as manual inlining. Manual inlining is always active even if no other optimizations or inlining methods have been enabled.

Automatic Inlining

With automatic inlining, the compiler determines which functions will be inlined. Automatic inlining is selected with the command line option **-OI**.

Command Line Inlining

Command line inlining allows the user to specify the names of certain functions to be inlined on the command line. This resembles manual inlining in that the user determines whether or

not each function will be inlined. Command line inlining is selected with the command line option **-OI=***name1*,*name2*.

SINGLE-PASS AND TWO-PASS INLINING

Whenever a function is used in only one file, and is defined in that file before it is used, and is manually marked for inlining with **_ _inline**, the function will be inlined during the normal course of compilation. This is referred to as single-pass inlining.

In order to inline a function which is not declared before it is used, or which is called from a file other than the file in which it is declared, two-pass inlining is required.

The command line options **-OI** and **-OI=** always enable two-pass inlining in addition to determining the criteria for selecting the functions to be inlined. Therefore, it may be necessary to specify the **-OI** or **-OI=** option to enable two-pass inlining, even if every function is manually marked for inlining.

USING THE COMMAND LINE OPTIONS

-OI The **-OI** option indicates that automatic inlining should be performed and that manual inlining should be performed in two passes. The compiler will automatically select functions to be inlined. In addition, each function which is manually marked with **_ _inline** will be inlined, including those which are used before they are declared in a file and those which are used in files in which they are not declared. For example,

```
% gcc -OI main.c prog1.c prog2.c
```

will cause the compiler to be invoked twice for each of the three source modules. First, each of the source files will be processed to produce an inline file with a **.inf** extension. Then each source file will be compiled again to produce an object file. On the second pass, both the original source file and the three **.inf** inline files will be used as input.

-OI=*names* The **-OI=***name* option also indicates that command line inlining should be performed and that manual inlining should be performed in two passes. A list of names of functions to be inlined

may be specified after the **-OI=** option, separated by commas. In addition, each function which is manually marked for inlining with **_ _inline** will be inlined, including those which are used before they are declared in a file and those which are used in files in which they are not declared. If automatic inlining is also to be performed the **-OI** option must be used as well.
The command line

```
% gcc -OI=sub,func main.c prog1.c prog2.c
```

will cause the functions **sub()** and **func()** to be inlined wherever they are encountered, along with each function which is manually marked for inlining with **_ _inline**.

-OI= The **-OI=** option without any arguments indicates that only manual inlining should be performed in two passes.

TWO-PASS INLINING IMPLEMENTATION

When two-pass inlining is enabled, the compiler driver invokes the compiler inliner once for each source module, creating an inline file for each module. All of the functions in a single source file which are candidates for inlining are stored in the corresponding inline file. The name of the inline file is formed by taking the source filename and replacing the suffix with a **.inf** suffix.

Next, the compiler is invoked a second time for each source file. The original source file along with all previously created **.inf** inline files will be used as input. An object file will be generated for each source file, exactly as it would if no inlining had been performed. The difference will be that certain calls to functions will have been replaced with inline copies of those routines. (Functions which are inlined will also have code generated for them, ensuring full compatibility with conventional programming techniques.) Finally, all of the object files will be linked normally.

INLINING OPTIMIZATION ENHANCEMENTS

Inlining is traditionally considered an optimization which increases program size for the sake of improving program speed. Program size is increased because a single function is generated in each place where it is called. Program speed is improved because the branch-to-subroutine call is eliminated. In fact,

there are many ways in which inlining serves to reduce program size as well as improve program speed. When a call is replaced by inlined code, the compiler can usually avoid saving and restoring several registers before and after the call. Parameters which normally must be passed on the stack to a called routine can be accessed directly by the inlined routine in their original location.

Furthermore, because Green Hills compilers perform inlining before most global optimizations, the process of inlining can significantly enhance the opportunities for additional optimizations resulting in very efficient code.

For example, if one or more parameter values are constant, large portions of the inlined routine may be reduced or eliminated at compile-time and loops which normally execute a variable number of times may become constant.

Register allocation may improve because the overhead associated with a call is eliminated. On most architectures, when a call to a routine exists within a routine, the number of registers available for local variables and temporaries is reduced. If all routine calls can be eliminated by inlining, the number of registers available for variables and temporaries will be increased.

Pessimistic assumptions made by the compiler when compiling the caller may not be necessary if no call is made. Normally the compiler must assume that global variables may be changed when a call is performed. This prevents the compiler from optimizing the values of expressions which contain global variables across a call to a function. When the function is inlined, the call is eliminated and the global variables may be optimized freely.

INLINING LIMITATIONS

The inlining optimization is subject to the following limitations:

- Source line number information related to inlined routines is deleted. When executing a program under control of a source debugger, no source code will be available for the inlined routine. Single stepping by source line will cause the entire inlined call to be executed as a single statement. However, you can debug the inlined call by stepping through the sequence of inlined machine instructions at the point of the source-level call.
- Functions containing **asm** statements cannot be inlined.

- Routines written in assembly language cannot be inlined because they are simply assembled to produce an object file. They cannot be processed by the compiler inliner.

SELECTING OPTIMIZATIONS

This section provides a demonstration on using the Unix system profiling utility to take full advantage of the specialized optimizations available with the Green Hills Compiler to improve the performance of your application.

The information that is generated by the profiler is commonly used to identify time-critical or inefficient code. This data is also very useful to select the appropriate optimizations for your particular application and specifically to identify functions for inlining and loop optimizations.

The system profiler produces a profile of your application which contains statistics relative to each function. Using the **-p** compiler option results in an executable containing calls to the system routine “monitor”. When your executable is run, these calls keep track of each function's performance. This raw data is written to a file called **mon.out**. The profile utility, **prof**, interprets the data in **mon.out** and generates a formatted report. The following list shows the categories of information in the report and what each category means.

%time	percentage of total run-time spent within a function
cumsecs	cumulative seconds spent for processing a function
#call	number of times a function is called
ms/call	time in milliseconds per function call
name	function name

When your code is linked, the compiler driver uses special profiled libraries to generate your executable.

Appendix

A

IMPLEMENTATION NOTES

IDENTIFIERS

Green Hills C++ reserves identifiers that contain a sequence of two underscores for its own use. In addition, identifiers reserved in the ANSI C standard are also reserved by Green Hills C++.

LINKAGE SPECIFICATIONS

Green Hills C++ supports linkage to C and C++.

The effect of a **"C"** linkage specification (**extern "C"**) on a function that is not a member function is that the function name is not encoded with the type information, as is otherwise done for C++ functions. Member functions are not affected by linkage specifications.

The C linkage specification (**extern "C"**), when applied to a non-function declaration, has no effect.

CLASS MEMBERS

The *Reference Manual* states that the order of allocation of non-static data members across access-specifiers is implementation dependent. Green Hills C++ allocates non-static data members in declaration order.

Appendix

B

ERROR MESSAGES

The following is a list of error messages. List shows the error number and the error tag on the same line, with the actual message for that error on the following line.

0001 LAST_LINE_INCOMPLETE

last line of file ends without a newline

0002 LAST_LINE_BACKSLASH

last line of file ends with a backslash

0003 INCLUDE_RECURSION

#include file "*xxxx*" includes itself

0004 OUT_OF_MEMORY

out of memory

0005 SOURCE_FILE_COULD_NOT_BE_OPENED

could not open source file "*xxxx*"

0006 COMMENT_UNCLOSED_AT_EOF

comment unclosed at end of file

0007 BAD_TOKEN

unrecognized token

0008 UNCLOSED_STRING

missing closing quote

0009 NESTED_COMMENT

nested comment is not allowed

0010 BAD_USE_OF_SHARP

"#" not expected here

0011 BAD_PP_DIRECTIVE_KEYWORD

unrecognized preprocessing directive

0012 END_OF_FLUSH

parsing restarts here after previous syntax error

0013 EXP_FILE_NAME

expected a file name

0014 EXTRA_TEXT_IN_PP_DIRECTIVE

extra text after expected end of preprocessing directive

0015 SOURCE_FILE_HAS_BAD_FORMAT

"*xxxx*" is not a file containing source text

0016 ILLEGAL_SOURCE_FILE_NAME

"*xxxx*" is not a valid source file name

0017 EXP_RBRACKET

expected a "]"

0018 EXP_RPAREN

expected a ")"

0019 EXTRA_CHARS_ON_NUMBER

extra text after expected end of number

0020 UNDEFINED_IDENTIFIER

identifier "*xxxx*" is undefined

0021 USELESS_TYPE_QUALIFIERS

type qualifiers are meaningless in this declaration

0022 BAD_HEX_DIGIT

invalid hexadecimal number

0023 INTEGER_TOO_LARGE

integer constant is too large

0024 BAD_OCTAL_DIGIT

invalid octal digit

0025 ZERO_LENGTH_STRING

quoted string should contain at least one character

0026 TOO_MANY_CHARACTERS

too many characters in character constant

0027 BAD_CHARACTER_VALUE

character value is out of range

0028 EXPR_NOT_CONSTANT

expression must have a constant value

0029 EXP_PRIMARY_EXPR

expected an expression

0030 BAD_FLOAT_VALUE

floating constant is out of range

0031 EXPR_NOT_INTEGRAL

expression must have integral type

0032 EXPR_NOT_ARITHMETIC

expression must have arithmetic type

0033 EXP_LINE_NUMBER

expected a line number

0034 BAD_LINE_NUMBER

invalid line number

0035 ERROR_DIRECTIVE

#error directive: *xxxx*

0036 MISSING_PP_IF

the #if for this directive is missing

0037 MISSING_ENDIF

the #endif for this directive is missing

0038 PP_ELSE_ALREADY_APPEARED

directive is not allowed -- an #else has already appeared

0039 DIVIDE_BY_ZERO

division by zero

0040 EXP_IDENTIFIER

expected an identifier

0041 EXPR_NOT_SCALAR

expression must have arithmetic or pointer type

0042 INCOMPATIBLE_OPERANDS

operand types are incompatible ("*type*" and "*type*")

0044 EXPR_NOT_POINTER

expression must have pointer type

0045 CANNOT_UNDEF_PREDEF_MACRO

#undef may not be used on this predefined name

0046 CANNOT_REDEF_PREDEF_MACRO

this predefined name may not be redefined

0047 BAD_MACRO_REDEF

macro redefined differently

0048 MIXED_FUNCTION_OBJECT_POINTERS

cast between pointer-to-object and pointer-to-function

0049 DUPLICATE_MACRO_PARAM_NAME

duplicate macro parameter name

0050 PASTE_CANNOT_BE_FIRST

"##" may not be first in a macro definition

0051 PASTE_CANNOT_BE_LAST

"##" may not be last in a macro definition

0052 EXP_MACRO_PARAM

expected a macro parameter name

0053 EXP_COLON

expected a ":"

0054 TOO_FEW_MACRO_ARGS

too few arguments in macro invocation

0055 TOO_MANY_MACRO_ARGS

too many arguments in macro invocation

0056 SIZEOF_FUNCTION

operand of sizeof may not be a function

0057 BAD_CONSTANT_OPERATOR

this operator is not allowed in a constant expression

0058 BAD_PP_OPERATOR

this operator is not allowed in a preprocessing expression

0059 BAD_CONSTANT_FUNCTION_CALL

function call is not allowed in a constant expression

0060 BAD_INTEGRAL_OPERATOR

this operator is not allowed in an integral constant expression

0061 INTEGER_OVERFLOW

integer operation result is out of range

0062 NEGATIVE_SHIFT_COUNT

shift count is negative

0063 SHIFT_COUNT_TOO_LARGE

shift count is too large

0064 USELESS_DECL

declaration does not declare anything

0065 EXP_SEMICOLON

expected a ";"

0066 ENUM_VALUE_OUT_OF_INT_RANGE

enumeration value is out of "int" range

0067 EXP_RBRACE

expected a "}"

0068 INTEGER_SIGN_CHANGE

integer conversion resulted in a change of sign

0069 INTEGER_TRUNCATED

integer conversion resulted in truncation

0070 INCOMPLETE_TYPE_NOT_ALLOWED

incomplete type is not allowed

0071 SIZEOF_BIT_FIELD

operand of sizeof may not be a bit field

0075 BAD_INDIRECTION_OPERAND

operand of "*" must be a pointer

0076 EMPTY_MACRO_ARGUMENT

argument to macro is empty

0077 MISSING_DECL_SPECIFIERS

this declaration has no storage class or type specifier

0078 INITIALIZER_IN_PARAM

a parameter declaration may not have an initializer

0079 EXP_TYPE_SPECIFIER

expected a type specifier

0080 STORAGE_CLASS_NOT_ALLOWED

a storage class may not be specified here

0081 MULT_STORAGE_CLASSES

more than one storage class may not be specified

0082 STORAGE_CLASS_NOT_FIRST

storage class is not first

0083 DUPL_TYPE_QUALIFIER

type qualifier specified more than once

0084 BAD_COMBINATION_OF_TYPE_SPECIFIERS

invalid combination of type specifiers

0085 BAD_PARAM_STORAGE_CLASS

invalid storage class for a parameter

0086 BAD_FUNCTION_STORAGE_CLASS

invalid storage class for a function

0087 TYPE_SPECIFIER_NOT_ALLOWED

a type specifier may not be used here

0088 ARRAY_OF_FUNCTION

array of functions is not allowed

0089 ARRAY_OF_VOID

array of void is not allowed

0090 FUNCTION_RETURNING_FUNCTION

function returning function is not allowed

0091 FUNCTION_RETURNING_ARRAY

function returning array is not allowed

0092 PARAM_ID_LIST_NEEDS_FUNCTION_DEF

identifier-list parameters may only be used in a function definition

0093 FUNCTION_TYPE_MUST_COME_FROM_DECLARATOR

function type may not come from a typedef

0094 ARRAY_SIZE_MUST_BE_POSITIVE

the size of an array must be greater than zero

0095 ARRAY_SIZE_TOO_LARGE

array is too large

0096 EMPTY_TRANSLATION_UNIT

a translation unit must contain at least one declaration

0097 BAD_FUNCTION_RETURN_TYPE

a function may not return a value of this type

0098 BAD_ARRAY_ELEMENT_TYPE

an array may not have elements of this type

0099 DECL_SHOULD_BE_OF_PARAM

a declaration here must declare a parameter

0100 DUPL_PARAM_NAME

duplicate parameter name

0101 D_ALREADY_DECLARED

"*xxxx*" has already been declared in the current scope

0102 NONSTD_FORWARD_DEF_ENUM

forward-defined enum type is nonstandard

0103 CLASS_TOO_LARGE

class is too large

0104 STRUCT_TOO_LARGE

struct or union is too large

0105 BAD_BIT_FIELD_SIZE

invalid size for bit field

0106 BAD_BIT_FIELD_TYPE

invalid type for a bit field

0107 ZERO_LENGTH_BIT_FIELD_MUST_BE_UNNAMED

zero-length bit field must be unnamed

0108 SIGNED_ONE_BIT_FIELD

signed bit field of length 1

0109 EXPR_NOT_PTR_TO_FUNCTION

expression must have (pointer-to-) function type

0110 EXP_DEFINITION_OF_TAG

expected either a definition or a tag name

0111 CODE_IS_UNREACHABLE

statement is unreachable

0112 EXP_WHILE

expected “while”

0113 NONSTD_DEFAULT_ARG

this use of a default argument is nonstandard

0114 NEVER_DEFINED

entity-kind "*entity*" was referenced but not defined

0115 CONTINUE_MUST_BE_IN_LOOP

a continue statement may only be used within a loop

0116 BREAK_MUST_BE_IN_LOOP_OR_SWITCH

a break statement may only be used within a loop or switch

0117 NO_VALUE_RETURNED_IN_NON_VOID_FUNCTION

non-void *entity-kind* "*entity*" should return a value

0118 VALUE_RETURNED_IN_VOID_FUNCTION

a void function may not return a value

0119 CAST_TO_BAD_TYPE

cast to type "*type*" is not allowed

0120 BAD_RETURN_VALUE_TYPE

return value type does not match the function type

0121 CASE_LABEL_MUST_BE_IN_SWITCH

a case label may only be used within a switch

0122 DEFAULT_LABEL_MUST_BE_IN_SWITCH

a default label may only be used within a switch

0123 CASE_LABEL_APPEARS_MORE_THAN_ONCE

case label value has already appeared in this switch

0124 DEFAULT_LABEL_APPEARS_MORE_THAN_ONCE

default label has already appeared in this switch

0125 EXP_LPAREN

expected a "("

0126 EXPR_NOT_AN_LVALUE

expression must be an lvalue

0127 EXP_STATEMENT

expected a statement

0128 LOOP_NOT_REACHABLE

loop is not reachable from preceding code

0129 BLOCK_SCOPE_FUNCTION_MUST_BE_EXTERN

a block-scope function may only have extern storage class

0130 EXP_LBRACE

expected a "{"

0131 EXPR_NOT_PTR_TO_CLASS

expression must have pointer-to-class type

0132 EXPR_NOT_PTR_TO_STRUCT_OR_UNION

expression must have pointer-to-struct-or-union type

0133 EXP_MEMBER_NAME

expected a member name

0134 EXP_FIELD_NAME

expected a field name

0135 NOT_A_MEMBER

entity-kind "*entity*" has no member "*xxxx*"

0136 NOT_A_FIELD

entity-kind "*entity*" has no field "*xxxx*"

0137 EXPR_NOT_A_MODIFIABLE_LVALUE

expression must be a modifiable lvalue

0138 ADDRESS_OF_REGISTER_VARIABLE

taking the address of a register variable is not allowed

0139 ADDRESS_OF_BIT_FIELD

taking the address of a bit field is not allowed

0140 TOO_MANY_ARGUMENTS

too many arguments in function call

0141 ALL_PROTO_PARAMS_MUST_BE_NAMED

unnamed prototyped parameters not allowed when body is present

0142 EXPR_NOT_POINTER_TO_OBJECT

expression must have pointer-to-object type

0143 PROGRAM_TOO_LARGE

program too large or complicated to compile

0144 BAD_INITIALIZER_TYPE

a value of type "*type*" cannot be used to initialize an entity of type "*type*"

0145 CANNOT_INITIALIZE

entity-kind "*entity*" may not be initialized

0146 TOO_MANY_INITIALIZER_VALUES

too many initializer values

0147 NOT_COMPATIBLE_WITH_PREVIOUS_DECL

declaration is incompatible with *entity-kind* "*entity*" (declared at line *xxxx*)

0148 ALREADY_INITIALIZED

entity-kind "*entity*" has already been initialized

0149 BAD_FILE_SCOPE_STORAGE_CLASS

a global-scope declaration may not have this storage class

0150 TYPE_CANNOT_BE_PARAM_NAME

a type name may not be redeclared as a parameter

0151 TYPEDEF_CANNOT_BE_PARAM_NAME

a typedef name may not be redeclared as a parameter

0152 NON_ZERO_INT_CONV_TO_POINTER

conversion of nonzero integer to pointer

0153 EXPR_NOT_CLASS

expression must have class type

0154 EXPR_NOT_STRUCT_OR_UNION

expression must have struct or union type

0155 OLD_FASHIONED_ASSIGNMENT_OPERATOR

old-fashioned assignment operator

0156 OLD_FASHIONED_INITIALIZER

old-fashioned initializer

0157 EXPR_NOT_INTEGRAL_CONSTANT

expression must be an integral constant expression

0158 EXPR_NOT_AN_LVALUE_OR_FUNCTION_DESIGNATOR

expression must be an lvalue or a function designator

0159 DECL_INCOMPATIBLE_WITH_PREVIOUS_USE

declaration is incompatible with previous "*entity*" (declared at line *xxxx*)

0160 EXTERNAL_NAME_CLASH

name conflicts with previously used external name "*xxxx*"

0161 UNRECOGNIZED_PRAGMA

unrecognized #pragma

0163 CANNOT_OPEN_TEMP_FILE

could not open temporary file "*xxxx*"

0164 TEMP_FILE_DIR_NAME_TOO_LONG

name of directory for temporary files is too long ("*xxxx*")

0165 TOO_FEW_ARGUMENTS

too few arguments in function call

0166 BAD_FLOAT_CONSTANT

invalid floating constant

0167 INCOMPATIBLE_PARAM

argument of type "*type*" is incompatible with parameter of type "*type*"

0168 FUNCTION_TYPE_NOT_ALLOWED

a function type is not allowed here

0169 EXP_DECLARATION

expected a declaration

0170 POINTER_OUTSIDE_BASE_OBJECT

pointer points outside of underlying object

0171 BAD_CAST

invalid type conversion

0172 LINKAGE_CONFLICT

external/internal linkage conflict with previous declaration

0173 FLOAT_TO_INTEGER_CONVERSION

floating-point value does not fit in required integral type

0174 EXPR_HAS_NO_EFFECT

expression has no effect

0175 SUBSCRIPT_OUT_OF_RANGE

subscript out of range

0177 DECLARED_BUT_NOT_REFERENCED

entity-kind "*entity*" was declared but never referenced

0178 PCC_ADDRESS_OF_ARRAY

"&" applied to an array has no effect

0179 MOD_BY_ZERO

right operand of "%%" is zero

0180 OLD_STYLE_INCOMPATIBLE_PARAM

argument is incompatible with formal parameter

0181 PRINTF_ARG_MISMATCH

argument is incompatible with corresponding format string conversion

0182 EMPTY_INCLUDE_SEARCH_PATH

could not open source file "*xxxx*" (no directories in search list)

0183 CAST_NOT_INTEGRAL

type of cast must be integral

0184 CAST_NOT_SCALAR

type of cast must be arithmetic or pointer

0185 INITIALIZATION_NOT_REACHABLE

dynamic initialization in unreachable code

0186 UNSIGNED_COMPARE_WITH_ZERO

pointless comparison of unsigned integer with zero

0187 ASSIGN_WHERE_COMPARE_MEANT

possible use of "=" where "==" was intended

0188 MIXED_ENUM_TYPE

enumerated type mixed with another type

0189 FILE_WRITE_ERROR

error while writing *xxxx* file

0190 BAD_IL_FILE

invalid intermediate language file

0191 CAST_TO_QUALIFIED_TYPE

type qualifier is meaningless on cast type

0192 UNRECOGNIZED_CHAR_ESCAPE

unrecognized character escape sequence

0193 UNDEFINED_PREPROC_ID

zero used for undefined preprocessing identifier

0194 EXP_ASM_STRING

expected an asm string

0195 ASM_FUNC_MUST_BE_PROTOTYPED

an asm function must be prototyped

0196 BAD_ASM_FUNC_ELLIPSIS

an asm function may not have an ellipsis

0219 FILE_DELETE_ERROR

error while deleting file "*xxxx*"

0220 INTEGER_TO_FLOAT_CONVERSION

integral value does not fit in required floating-point type

0221 FLOAT_TO_FLOAT_CONVERSION

floating-point value does not fit in required floating-point type

0222 BAD_FLOAT_OPERATION_RESULT

floating-point operation result is out of range

0223 IMPLICIT_FUNC_DECL

function declared implicitly

0224 TOO_FEW_PRINTF_ARGS

the format string requires additional arguments

0225 TOO_MANY_PRINTF_ARGS

the format string ends before this argument

0226 BAD_PRINTF_FORMAT_STRING

invalid format string conversion

0227 MACRO_RECURSION

macro recursion

0228 NONSTD_EXTRA_COMMA

trailing comma is nonstandard

0229 ENUM_BIT_FIELD_TOO_SMALL

bit field cannot contain all values of the enumerated type

0230 NONSTD_BIT_FIELD_TYPE

nonstandard type for a bit field

0231 DECL_IN_PROTOTYPE_SCOPE

declaration is not visible outside of function

0232 DECL_OF_VOID_IGNORED

old-fashioned typedef of “void” ignored

0233 OLD_FASHIONED_FIELD_SELECTION

left operand is not a struct or union containing this field

0234 OLD_FASHIONED_PTR_FIELD_SELECTION

pointer does not point to struct or union containing this field

0235 VAR_RETAINED_INCOMP_TYPE

variable “*xxxx*” was declared with a never-completed type

0236 BOOLEAN_CONTROLLING_EXPR_IS_CONSTANT

controlling expression is constant

0237 SWITCH_SELECTOR_EXPR_IS_CONSTANT

selector expression is constant

0238 BAD_PARAM_SPECIFIER

invalid specifier on a parameter

0239 BAD_SPECIFIER_OUTSIDE_CLASS_DECL

invalid specifier outside a class declaration

0240 DUPL_DECL_SPECIFIER

duplicate specifier in declaration

0241 BASE_CLASS_NOT_ALLOWED_FOR_UNION

a union is not allowed to have a base class

0242 ACCESS_ALREADY_SPECIFIED

multiple access control specifiers are not allowed

0243 MISSING_CLASS_DEFINITION

class or struct definition is missing

0244 NAME_NOT_MEMBER_OF_CLASS_OR_BASE_CLASSES

qualified name is not a member of class "*type*" or its base classes

0245 MEMBER_REF_REQUIRES_OBJECT

a nonstatic member reference must be relative to a specific object

0246 NONSTATIC_MEMBER_DEF_NOT_ALLOWED

a nonstatic data member may not be defined outside its class

0247 ALREADY_DEFINED

entity-kind "*entity*" has already been defined

0248 POINTER_TO_REFERENCE

pointer to reference is not allowed

0249 REFERENCE_TO_REFERENCE

reference to reference is not allowed

0250 REFERENCE_TO_VOID

reference to void is not allowed

0251 ARRAY_OF_REFERENCE

array of reference is not allowed

0252 MISSING_INITIALIZER_ON_REFERENCE

reference *entity-kind* "*entity*" requires an initializer

0253 EXP_COMMA

expected a ","

0254 TYPE_IDENTIFIER_NOT_ALLOWED

type name is not allowed

0255 TYPE_DEFINITION_NOT_ALLOWED

type definition is not allowed

0256 BAD_TYPE_NAME_REDECLARATION

invalid redeclaration of type name "*entity*" (declared at line *xxxx*)

0257 MISSING_INITIALIZER_ON_CONST

const *entity-kind* "*entity*" requires an initializer

0258 THIS_USED_INCORRECTLY

"this" may only be used inside a nonstatic member function

0259 CONSTANT_VALUE_NOT_KNOWN

constant value is not known

0260 MISSING_TYPE_SPECIFIER

explicit type is missing (“int” assumed)

0261 MISSING_ACCESS_SPECIFIER

access control not specified (“*xxxx*” by default)

0262 NOT_A_CLASS_OR_STRUCT_NAME

not a class or struct name

0263 DUPL_BASE_CLASS_NAME

duplicate base class name

0264 BAD_BASE_CLASS

invalid base class

0265 NO_ACCESS_TO_NAME

entity-kind “*entity*” is inaccessible

0266 AMBIGUOUS_NAME

“*entity*” is ambiguous

0267 OLD_STYLE_PARAMETER_LIST

old-style parameter list (anachronism)

0268 DECLARATION_AFTER_STATEMENTS

declaration may not appear after executable statement in block

0269 INACCESSIBLE_BASE_CLASS

base class “*type*” is inaccessible

0274 IMPROPERLY_TERMINATED_MACRO_CALL

improperly terminated macro invocation

0276 ID_MUST_BE_CLASS_OR_NAMESPACE_NAME

name followed by "::" must be a class or namespace name

0277 BAD_FRIEND_DECL

invalid friend declaration

0278 VALUE_RETURNED_IN_CONSTRUCTOR

a constructor or destructor may not return a value

0279 BAD_DESTRUCTOR_DECL

invalid destructor declaration

0280 CLASS_AND_MEMBER_NAME_CONFLICT

invalid declaration of a member with the same name as its class

0281 GLOBAL_QUALIFIER_NOT_ALLOWED

global-scope qualifier (leading "::") is not allowed

0282 NAME_NOT_FOUND_IN_FILE_SCOPE

the global scope has no "*xxxx*"

0283 QUALIFIED_NAME_NOT_ALLOWED

qualified name is not allowed

0284 NULL_REFERENCE

NULL reference is not allowed

0285 BRACE_INITIALIZATION_NOT_ALLOWED

initialization with "{...}" is not allowed for object of type "*type*"

0286 AMBIGUOUS_BASE_CLASS

base class "*type*" is ambiguous

0287 AMBIGUOUS_DERIVED_CLASS

derived class "*type*" contains more than one instance of class "*type*"

0288 DERIVED_CLASS_FROM_VIRTUAL_BASE

derived class "*type*" has class "*type*" as a virtual base class

0289 NO_MATCHING_CONSTRUCTOR

no instance of constructor "*entity*" matches the argument list

0290 AMBIGUOUS_COPY_CONSTRUCTOR

copy constructor for class "*type*" is ambiguous

0291 NO_DEFAULT_CONSTRUCTOR

no default constructor exists for class "*type*"

0292 NOT_A_FIELD_OR_BASE_CLASS

"*xxxx*" is not a nonstatic data member or base class of class "*type*"

0293 INDIRECT_NONVIRTUAL_BASE_CLASS_NOT_ALLOWED

indirect nonvirtual base class is not allowed

0294 BAD_UNION_FIELD

invalid union member -- class "*type*" has a disallowed member function

0295 OVERLOADED_FUNCTION_TYPES_TOO_SIMILAR

cannot overload functions -- parameter types are too similar

0296 BAD_RVALUE_ARRAY

invalid use of non-lvalue array

0297 EXP_OPERATOR

expected an operator

0298 INHERITED_MEMBER_NOT_ALLOWED

inherited member is not allowed

0299 INDETERMINATE_OVERLOADED_FUNCTION

cannot determine which instance of *entity-kind* "*entity*" is intended

0300 BOUND_FUNCTION_MUST_BE_CALLED

a pointer to a bound function may only be used to call the function

0301 DUPLICATE_TYPEDEF

typedef name has already been declared (with same type)

0302 FUNCTION_REDEFINITION

entity-kind "*entity*" has already been defined

0303 OVERLOADED_FUNCTION_INCOMPATIBLE_TYPE

type does not match any instance of *entity-kind* "*entity*"

0304 NO_MATCHING_FUNCTION

no instance of *entity-kind* "*entity*" matches the argument list

0305 TYPE_DEF_NOT_ALLOWED_IN_FUNC_TYPE_DECL

type definition is not allowed in function return type declaration

0306 DEFAULT_ARG_NOT_AT_END

default argument not at end of parameter list

0307 DEFAULT_ARG_ALREADY_DEFINED

redefinition of default argument

0308 AMBIGUOUS_OVERLOADED_FUNCTION

more than one instance of *entity-kind* "*entity*" matches the argument list:

0309 AMBIGUOUS_CONSTRUCTOR

more than one instance of constructor "*entity*" matches the argument list:

0310 BAD_DEFAULT_ARG_TYPE

default argument of type "*type*" is incompatible with parameter of type "*type*"

0311 RETURN_TYPE_CANNOT_DISTINGUISH_FUNCTIONS

cannot overload functions distinguished by return type alone

0312 NO_USER_DEFINED_CONVERSION

no suitable user-defined conversion from "*type*" to "*type*" exists

0313 FUNCTION_QUALIFIER_NOT_ALLOWED

type qualifier is not allowed on this function

0314 VIRTUAL_STATIC_NOT_ALLOWED

only nonstatic member functions may be virtual

0315 UNQUAL_FUNCTION_WITH_QUAL_OBJECT

the object has type qualifiers that are not compatible with the member function

0316 TOO_MANY_VIRTUAL_FUNCTIONS

program too large to compile (too many virtual functions)

0317 BAD_RETURN_TYPE_ON_VIRTUAL_FUNCTION_OVERRIDE

type differs from base class virtual function by return type alone

0318 AMBIGUOUS_VIRTUAL_FUNCTION_OVERRIDE

override of virtual *entity-kind* "*entity*" is ambiguous

0319 PURE_SPECIFIER_ON_NONVIRTUAL_FUNCTION

pure specifier ("= 0") allowed only on virtual functions

0320 BAD_PURE_SPECIFIER

badly-formed pure specifier (only "= 0" is allowed)

0321 BAD_DATA_MEMBER_INITIALIZATION

data member initializer is not allowed

0322 ABSTRACT_CLASS_OBJECT_NOT_ALLOWED

object of abstract class type is not allowed

0323 FUNCTION_RETURNING_ABSTRACT_CLASS

function returning abstract class is not allowed

0324 DUPLICATE_FRIEND_DECL

duplicate friend declaration

0325 INLINE_AND_NONFUNCTION

inline specifier allowed on function declarations only

0326 INLINE_NOT_ALLOWED

"inline" is not allowed

0327 BAD_STORAGE_CLASS_WITH_INLINE

invalid storage class for an inline function

0328 BAD_MEMBER_STORAGE_CLASS

invalid storage class for a class member

0329 LOCAL_CLASS_FUNCTION_DEF_MISSING

local class member *entity-kind* "*entity*" requires a definition

0330 INACCESSIBLE_SPECIAL_FUNCTION

entity-kind "*entity*" is inaccessible

0332 MISSING_CONST_COPY_CONSTRUCTOR

class "*type*" has no copy constructor to copy a const object

0333 DEFINITION_OF_IMPLICITLY_DECLARED_FUNCTION

defining an implicitly declared member function is not allowed

0334 NO_SUITABLE_COPY_CONSTRUCTOR

class "*type*" has no suitable copy constructor

0335 LINKAGE_SPECIFIER_NOT_ALLOWED

linkage specification is not allowed

0336 BAD_LINKAGE_SPECIFIER

unknown external linkage specification

0337 INCOMPATIBLE_LINKAGE_SPECIFIER

linkage specification is incompatible with previous "*entity*" (declared at line *xxxx*)

0338 OVERLOADED_FUNCTION_LINKAGE

more than one instance of *entity-kind* "*entity*" has "C" linkage

0339 AMBIGUOUS_DEFAULT_CONSTRUCTOR

class "*type*" has more than one default constructor

0340 TEMP_USED_FOR_REF_INIT

value copied to temporary, reference to temporary used

0341 NONMEMBER_OPERATOR_NOT_ALLOWED

"operator*xxxx*" must be a member function

0342 STATIC_MEMBER_OPERATOR_NOT_ALLOWED

operator may not be a static member function

0343 TOO_MANY_ARGS_FOR_CONVERSION

no arguments allowed on user-defined conversion

0344 TOO_MANY_ARGS_FOR_OPERATOR

too many arguments for operator function

0345 TOO_FEW_ARGS_FOR_OPERATOR

too few arguments for operator function

0346 NO_ARGS_WITH_CLASS_TYPE

nonmember operator requires an argument with class type

0347 DEFAULT_ARG_EXPR_NOT_ALLOWED

default argument is not allowed

0348 AMBIGUOUS_USER_DEFINED_CONVERSION

more than one user-defined conversion from "*type*" to "*type*" applies:

0349 NO_MATCHING_OPERATOR_FUNCTION

no operator "*xxxx*" matches these operands

0350 AMBIGUOUS_OPERATOR_FUNCTION

more than one operator "*xxxx*" matches these operands:

0351 BAD_ARG_TYPE_FOR_OPERATOR_NEW

first parameter of allocation function must be of type "size_t"

0352 BAD_RETURN_TYPE_FOR_OP_NEW

allocation function requires “void *” return type

0353 BAD_RETURN_TYPE_FOR_OP_DELETE

deallocation function requires “void” return type

0354 BAD_FIRST_ARG_TYPE_FOR_OPERATOR_DELETE

first parameter of deallocation function must be of type “void *”

0355 BAD_SECOND_ARG_TYPE_FOR_OPERATOR_DELETE

second parameter of deallocation function must be of type “size_t”

0356 TYPE_MUST_BE_OBJECT_TYPE

type must be an object type

0357 BASE_CLASS_ALREADY_INITIALIZED

base class “*type*” has already been initialized

0358 BASE_CLASS_INIT_ANACHRONISM

base class name required -- “*type*” assumed (anachronism)

0359 MEMBER_ALREADY_INITIALIZED

entity-kind “*entity*” has already been initialized

0360 MISSING_BASE_CLASS_OR_MEMBER_NAME

name of member or base class is missing

0361 ASSIGNMENT_TO_THIS

assignment to “this” (anachronism)

0362 OVERLOAD_ANACHRONISM

“overload” keyword used (anachronism)

0363 ANON_UNION_MEMBER_ACCESS

invalid anonymous union -- nonpublic member is not allowed

0364 ANON_UNION_MEMBER_FUNCTION

invalid anonymous union -- member function is not allowed

0365 ANON_UNION_STORAGE_CLASS

anonymous union at global or namespace scope must be declared static

0366 MISSING_INITIALIZER_ON_FIELDS

entity-kind "*entity*" provides no initializer for:

0367 CANNOT_INITIALIZE_FIELDS

implicitly generated constructor for class "*type*" cannot initialize:

0368 NO_CTOR_BUT_CONST_OR_REF_MEMBER

entity-kind "*entity*" defines no constructor to initialize the following:

0369 VAR_WITH_UNINITIALIZED_MEMBER

entity-kind "*entity*" has an uninitialized const or reference member

0370 VAR_WITH_UNINITIALIZED_FIELD

entity-kind "*entity*" has an uninitialized const field

0371 MISSING_CONST_ASSIGNMENT_OPERATOR

class "*type*" has no assignment operator to copy a const object

0372 NO_SUITABLE_ASSIGNMENT_OPERATOR

class "*type*" has no suitable assignment operator

0373 AMBIGUOUS_ASSIGNMENT_OPERATOR

ambiguous default assignment operator for class "*type*"

0374 CONST_VOLATILE_NOT_ALLOWED

const or volatile qualifier is not allowed

0375 MISSING_TYPEDEF_NAME

declaration requires a typedef name

0377 VIRTUAL_NOT_ALLOWED

“virtual” is not allowed

0378 STATIC_NOT_ALLOWED

“static” is not allowed

0379 BOUND_FUNCTION_CAST_ANACHRONISM

cast of bound function to normal function pointer (anachronism)

0380 EXPR_NOT_PTR_TO_MEMBER

expression must have pointer-to-member type

0381 EXTRA_SEMICOLON

extra “;” ignored

0382 NONSTD_CONST_MEMBER

declaring a member constant is nonstandard

0384 NO_MATCHING_NEW_FUNCTION

no instance of overloaded “*entity*” matches the argument list

0385 DELETE_ALREADY_DECLARED

operator delete() may not be overloaded

0386 NO_MATCH_FOR_ADDR_OF_OVERLOADED_FUNCTION

no instance of *entity-kind* “*entity*” matches the required type

0387 DELETE_COUNT_ANACHRONISM

delete array size expression ignored (anachronism)

0388 BAD_RETURN_TYPE_FOR_OP_ARROW

"*type*" is an invalid return type for "*entity*"

0389 CAST_TO_ABSTRACT_CLASS

a cast to an abstract class is not allowed

0390 BAD_USE_OF_MAIN

function "main" may not be called or have its address taken

0391 INITIALIZER_NOT_ALLOWED_ON_ARRAY_NEW

a new-initializer may not be specified for an array

0392 MEMBER_FUNCTION_REDECL_OUTSIDE_CLASS

member function "*entity*" may not be redeclared outside its class

0393 PTR_TO_INCOMPLETE_CLASS_TYPE_NOT_ALLOWED

pointer to incomplete class type is not allowed

0394 REF_TO_NESTED_FUNCTION_VAR

reference to local variable of enclosing function is not allowed

0395 SINGLE_ARG_POSTFIX_INCR_DECR_ANACHRONISM

single-argument function used for postfix "*xxxx*" (anachronism)

0397 BAD_DEFAULT_ASSIGNMENT

implicitly generated assignment operator cannot copy:

0398 NONSTD_ARRAY_CAST

cast to array type is nonstandard (treated as cast to "*type*")

0399 CLASS_WITH_OP_NEW_BUT_NO_OP_DELETE

entity-kind "*entity*" has an operator new*xxxx*() but no operator delete*xxxx*()

0400 CLASS_WITH_OP_DELETE_BUT_NO_OP_NEW

entity-kind "*entity*" has an operator delete*xxxx*() but no operator new*xxxx*()

0401 BASE_CLASS_WITH_NONVIRTUAL_DTOR

destructor for base class "*type*" is not virtual

0402 NO_ACCESS_TO_CONSTRUCTORS

entity-kind "*entity*" has no accessible constructors

0403 MEMBER_FUNCTION_REDECLARATION

entity-kind "*entity*" has already been declared

0404 INLINE_MAIN

function "main" may not be declared inline

0405 CLASS_AND_MEMBER_FUNCTION_NAME_CONFLICT

member function with the same name as its class must be a constructor

0406 NESTED_CLASS_ANACHRONISM

using nested *entity-kind* "*entity*" (anachronism)

0407 TOO_MANY_PARAMS_FOR_DESTRUCTOR

a destructor may not have parameters

0408 BAD_CONSTRUCTOR_PARAM

copy constructor for class "*type*" may not have a parameter of type "*type*"

0409 INCOMPLETE_FUNCTION_RETURN_TYPE

function return type is incomplete

0410 PROTECTED_ACCESS_PROBLEM

protected *entity-kind* "*entity*" is not accessible through a "*type*" pointer or object

0411 PARAM_NOT_ALLOWED

a parameter is not allowed

0412 ASM_DECL_NOT_ALLOWED

an "asm" declaration is not allowed here

0413 NO_CONVERSION_FUNCTION

no suitable conversion function from "*type*" to "*type*" exists

0414 DELETE_OF_INCOMPLETE_CLASS

delete of pointer to incomplete class

0415 NO_CONSTRUCTOR_FOR_CONVERSION

no suitable constructor exists to convert from "*type*" to "*type*"

0416 AMBIGUOUS_CONSTRUCTOR_FOR_CONVERSION

more than one constructor applies to convert from "*type*" to "*type*":

0417 AMBIGUOUS_CONVERSION_FUNCTION

more than one conversion function from "*type*" to "*type*" applies:

0418 AMBIGUOUS_CONVERSION_TO_BUILTIN

more than one conversion function from "*type*" to a built-in type applies:

0424 ADDR_OF_CONSTRUCTOR_OR_DESTRUCTOR

a constructor or destructor may not have its address taken

0425 DOLLAR_USED_IN_IDENTIFIER

dollar sign ("$") used in identifier

0426 NONCONST_REF_INIT_ANACHRONISM

temporary used for initial value of reference to non-const (anachronism)

0427 QUALIFIER_IN_MEMBER_DECLARATION

qualified name is not allowed in member declaration

0428 MIXED_ENUM_TYPE_ANACHRONISM

enumerated type mixed with another type (anachronism)

0429 NEW_ARRAY_SIZE_MUST_BE_NONNEGATIVE

the size of an array in “new” must be non-negative

0430 RETURN_REF_INIT_REQUIRES_TEMP

returning reference to local temporary

0431 CFRONT_NONCONST_REF_INIT

const qualifier dropped in initializing reference to non-const

0432 ENUM_NOT_ALLOWED

“enum” declaration is not allowed

0433 QUALIFIER_DROPPED_IN_REF_INIT

initial value of reference has excess type qualifiers

0434 BAD_NONCONST_REF_INIT

initial value of reference to non-const has incorrect type

0435 DELETE_OF_FUNCTION_POINTER

a pointer to function may not be deleted

0436 BAD_CONVERSION_FUNCTION_DECL

conversion function must be a nonstatic member function

0437 BAD_TEMPLATE_DECLARATION_SCOPE

template declaration is not allowed here

0438 EXP_LT

expected a "<"

0439 EXP_GT

expected a ">"

0440 MISSING_TEMPLATE_PARAM

template parameter declaration is missing

0441 MISSING_TEMPLATE_ARG_LIST

argument list for *entity-kind* "*entity*" is missing

0442 TOO_FEW_TEMPLATE_ARGS

too few arguments for *entity-kind* "*entity*"

0443 TOO_MANY_TEMPLATE_ARGS

too many arguments for *entity-kind* "*entity*"

0444 NOT_A_TYPE_ARG

template parameter for a function template must be a type

0445 NOT_USED_IN_TEMPLATE_FUNCTION_PARAMS

entity-kind "*entity*" is not used in declaring the argument types of *entity-kind* "*entity*"

0446 CFRONT_MULTIPLE_NESTED_TYPES

two nested types have the same name: "*entity*" and "*entity*" (declared at line *xxxx*)2 (cfront compatibility)

0447 CFRONT_GLOBAL_DEFINED_AFTER_NESTED_TYPE

global "*entity*" was declared after nested "*entity*" (declared at line *xxxx*)2 (cfront compatibility)

0449 AMBIGUOUS_PTR_TO_OVERLOADED_FUNCTION

more than one instance of *entity-kind* "*entity*" matches the required type

0450 NONSTD_LONG_LONG

the type "long long" is nonstandard

0451 NONSTD_FRIEND_DECL

omission of "*xxxx*" is nonstandard

0452 RETURN_TYPE_ON_CONVERSION_FUNCTION

return type may not be specified on a conversion function

0456 RUNAWAY_RECURSIVE_INSTANTIATION

excessive recursion at instantiation of *entity-kind* "*entity*"

0457 BAD_TEMPLATE_DECLARATION

"*xxxx*" is not a function or static data member

0458 BAD_NONTYPE_TEMPLATE_ARG

argument of type "*type*" is incompatible with template parameter of type "*type*"

0459 INIT_NEEDING_TEMP_NOT_ALLOWED

initialization requiring a temporary or conversion is not allowed

0460 DECL_HIDES_FUNCTION_PARAMETER

declaration of "*xxxx*" hides function parameter

0461 NONCONST_REF_INIT_FROM_RVALUE

initial value of reference to non-const must be an lvalue

0463 TEMPLATE_NOT_ALLOWED

“template” is not allowed

0464 NOT_A_CLASS_TEMPLATE

“*type*” is not a class template

0466 FUNCTION_TEMPLATE_NAMED_MAIN

“main” is not a valid name for a function template

0467 UNION_NONUNION_MISMATCH

invalid reference to *entity-kind* “*entity*” (union/nonunion mismatch)

0468 LOCAL_TYPE_IN_TEMPLATE_ARG

a template argument may not reference a local type

0469 TAG_KIND_INCOMPATIBLE_WITH_DECLARATION

tag kind of *xxxx* is incompatible with declaration of *entity-kind* “*entity*” (declared at line *xxxx*)

0470 NAME_NOT_TAG_IN_FILE_SCOPE

the global scope has no tag named “*xxxx*”

0471 NOT_A_TAG_MEMBER

entity-kind “*entity*” has no tag member named “*xxxx*”

0472 PTR_TO_MEMBER_TYPEDEF

member function typedef (allowed for cfront compatibility)

0473 BAD_USE_OF_MEMBER_FUNCTION_TYPEDEF

entity-kind "*entity*" may be used only in pointer-to-member declaration

0475 NONEXTERNAL_ENTITY_IN_TEMPLATE_ARG

a template argument may not reference a non-external entity

0476 ID_MUST_BE_CLASS_OR_TYPE_NAME

name followed by "::~" must be a class name or a type name

0477 DESTRUCTOR_NAME_MISMATCH

destructor name does not match name of class "*type*"

0478 DESTRUCTOR_TYPE_MISMATCH

type used as destructor name does not match type "*type*"

0479 CALLED_FUNCTION_REDECLARED_INLINE

entity-kind "*entity*" redeclared "inline" after being called

0480 VACUOUS_DESTRUCTOR_NAME_MISMATCH

destructor name does not match left operand of "->" or "."

0481 BAD_STORAGE_CLASS_ON_TEMPLATE_DECL

invalid storage class for a template declaration

0482 NO_ACCESS_TO_TYPE_CFRONT_MODE

entity-kind "*entity*" is an inaccessible type (allowed for cfront compatibility)

0483 RETURN_TYPE_NOT_ALLOWED

a return type is not allowed

0484 INVALID_INSTANTIATION_PRAGMA_ARGUMENT

invalid instantiation pragma argument

0485 NOT_INSTANTIATABLE_ENTITY

entity-kind "*entity*" is not an entity that can be instantiated

0486 COMPILER_GENERATED_FUNCTION_CANNOT_BE_INSTANTIATED

compiler generated function *entity-kind* "*entity*" cannot be instantiated

0487 INLINE_FUNCTION_CANNOT_BE_INSTANTIATED

inline function *entity-kind* "*entity*" cannot be instantiated

0488 PURE_VIRTUAL_FUNCTION_CANNOT_BE_INSTANTIATED

pure virtual function *entity-kind* "*entity*" cannot be instantiated

0489 INSTANTIATION_REQUESTED_NO_DEFINITION_SUPPLIED

entity-kind "*entity*" cannot be instantiated -- no template definition was supplied

0490 INSTANTIATION_REQUESTED_AND_SPECIFIC_DEFINITION

entity-kind "*entity*" cannot be instantiated -- a specific definition has been supplied

0491 NO_CONSTRUCTOR

class "*type*" has no constructor

0492 TEMPLATE_PARAM_ONLY_USED_IN_DEFAULT_ARGS

entity-kind "*entity*" must be used in a parameter without a default value in *entity-kind* "*entity*"

0493 NO_MATCH_FOR_TYPE_OF_OVERLOADED_FUNCTION

no instance of *entity-kind* "*entity*" matches the specified type

0494 NONSTD_VOID_PARAM_LIST

declaring a void parameter list with a typedef is nonstandard

0495 CFRONT_NAME_LOOKUP_BUG

global *entity-kind* "*entity*" used instead of *entity-kind* "*entity*" (cfront compatibility)

0496 REDECLARATION_OF_TEMPLATE_PARAM_NAME

template parameter "*xxxx*" may not be redeclared in this scope

0497 DECL_HIDES_TEMPLATE_PARAMETER

declaration of "*xxxx*" hides template parameter

0498 MUST_BE_PROTOTYPE_INSTANTIATION

template argument list must match the parameter list

0499 CONVERSION_TO_TYPE_NOT_ALLOWED

conversion function to convert from "*type*" to "*type*" is not allowed

0500 BAD_EXTRA_ARG_FOR_POSTFIX_OPERATOR

extra argument of postfix "operator*xxxx*" must be of type "int"

0501 FUNCTION_TYPE_REQUIRED

an operator name must be declared as a function

0502 OPERATOR_NAME_NOT_ALLOWED

operator name is not allowed

0503 SPECIFIC_DEF_MUST_BE_GLOBAL

class template specialization "*entity*" may not be declared in the current scope

0504 NONSTD_MEMBER_FUNCTION_ADDRESS

nonstandard form for taking the address of a member function

0505 TOO_FEW_TEMPLATE_PARAMS

too few template parameters -- does not match previous declaration

0506 TOO_MANY_TEMPLATE_PARAMS

too many template parameters -- does not match previous declaration

0507 TEMPLATE_OPERATOR_DELETE

function template for operator delete() is not allowed

0508 CLASS_TEMPLATE_SAME_NAME_AS_TEMPL_PARAM

class template and template parameter may not have the same name

0509 BAD_CONSTRUCTOR_NAME

"*entity*" cannot be used to designate constructor for *entity-kind* "*entity*"

0510 UNNAMED_TYPE_IN_TEMPLATE_ARG

a template argument may not reference an unnamed type

0511 ENUM_TYPE_NOT_ALLOWED

enumerated type is not allowed

0512 QUALIFIED_REFERENCE_TYPE

type qualifier on a reference type is not allowed

0513 INCOMPATIBLE_ASSIGNMENT_OPERANDS

a value of type "*type*" cannot be assigned to an entity of type "*type*"

0514 UNSIGNED_COMPARE_WITH_NEGATIVE

pointless comparison of unsigned integer with a negative constant

0515 CONVERTING_TO_INCOMPLETE_CLASS

cannot convert to incomplete class "*type*"

0516 MISSING_INITIALIZER_ON_UNNAMED_CONST

const object requires an initializer

0517 UNNAMED_OBJECT_WITH_UNINITIALIZED_FIELD

object has an uninitialized const or reference member

0518 NONSTD_PP_DIRECTIVE

nonstandard preprocessing directive

0519 UNEXPECTED_TEMPLATE_ARG_LIST

entity-kind "*entity*" may not have a template argument list

0520 MISSING_INITIALIZER_LIST

initialization with "{...}" expected for aggregate object

0521 INCOMPATIBLE_PTR_TO_MEMBER_SELECTION_OPERANDS

pointer-to-member selection class types are incompatible ("*type*" and "*type*")

0522 SELF_FRIENDSHIP

pointless friend declaration

0523 PERIOD_USED_AS_QUALIFIER

"." used in place of "::" to form a qualified name (cfront anachronism)

0524 CONST_FUNCTION_ANACHRONISM

non-const function called for const object (cfront anachronism)

0525 DEPENDENT_STMT_IS_DECLARATION

a dependent statement may not be a declaration

0526 VOID_PARAM_NOT_ALLOWED

a parameter may not have void type

0529 BAD_TEMPL_ARG_EXPR_OPERATOR

this operator is not allowed in a template argument expression

0530 MISSING_HANDLER

try block requires at least one handler

0531 MISSING_EXCEPTION_DECLARATION

handler requires an exception declaration

0532 MASKED_BY_DEFAULT_HANDLER

handler is masked by default handler

0533 MASKED_BY_HANDLER

handler is masked by previous handler for type "*type*"

0534 LOCAL_TYPE_USED_IN_EXCEPTION

use of a local type to specify an exception

0535 REDUNDANT_EXCEPTION_SPECIFICATION_TYPE

redundant type in exception specification

0536 INCOMPATIBLE_EXCEPTION_SPECIFICATION

exception specification is incompatible with that of previous *entity-kind* "*entity*" (declared at line *xxxx*):

0537 PREVIOUS_EXCEPTION_SPECIFICATION_WAS_EMPTY

previously specified: no exceptions will be thrown

0538 OMITTED_IN_PREVIOUS_EXCEPTION_SPECIFICATION

previously omitted: "*type*"

0539 INCLUDED_IN_PREVIOUS_EXCEPTION_SPECIFICATION

previously specified but omitted here: "*type*"

0540 NO_EXCEPTION_SUPPORT

support for exception handling is disabled

0541 OMITTED_EXCEPTION_SPECIFICATION

omission of exception specification is incompatible with previous *entity-kind* "*entity*" (declared at line *xxxx*)

0542 CANNOT_CREATE_INSTANTIATION_INFORMATION_FILE

could not create instantiation information file "*xxxx*"

0543 NON_ARITH_OPERATION_IN_TEMPL_ARG

non-arithmetic operation not allowed in nontype template argument

0544 LOCAL_TYPE_IN_NONLOCAL_VAR

use of a local type to declare a nonlocal variable

0545 LOCAL_TYPE_IN_FUNCTION

use of a local type to declare a function

0546 BRANCH_PAST_INITIALIZATION

transfer of control bypasses initialization of:

0548 BRANCH_INTO_HANDLER

transfer of control into an exception handler

0549 USED_BEFORE_SET

entity-kind "*entity*" is used before its value is set

0550 SET_BUT_NOT_USED

entity-kind "*entity*" was set but never used

0551 BAD_SCOPE_FOR_DEFINITION

entity-kind "*entity*" cannot be defined in the current scope

0552 EXCEPTION_SPECIFICATION_NOT_ALLOWED

exception specification is not allowed

0553 TEMPLATE_AND_INSTANCE_LINKAGE_CONFLICT

external/internal linkage conflict for *entity-kind* "*entity*" (declared at line *xxxx*)

0554 CONVERSION_FUNCTION_NOT_USABLE

entity-kind "*entity*" will not be called for implicit or explicit conversions

0555 TAG_KIND_INCOMPATIBLE_WITH_TEMPLATE_PARAMETER

tag kind of *xxxx* is incompatible with template parameter of type "*type*"

0556 TEMPLATE_OPERATOR_NEW

function template for operator new(size_t) is not allowed

0558 BAD_MEMBER_TYPE_IN_PTR_TO_MEMBER

pointer to member of type "*type*" is not allowed

0559 ELLIPSIS_ON_OPERATOR_FUNCTION

ellipsis is not allowed in operator function parameter list

0560 UNIMPLEMENTED_KEYWORD

"*entity*" is reserved for future use as a keyword

0561 CL_INVALID_MACRO_DEFINITION

invalid macro definition:

0562 CL_INVALID_MACRO_UNDEFINITION

invalid macro undefinition:

0563 CL_INVALID_PREPROCESSOR_OUTPUT_FILE

invalid preprocessor output file

0564 CL_CANNOT_OPEN_PREPROCESSOR_OUTPUT_FILE

cannot open preprocessor output file

0565 CL_IL_FILE_MUST_BE_SPECIFIED

IL file name must be specified if input is

0566 CL_INVALID_IL_OUTPUT_FILE

invalid IL output file

0567 CL_CANNOT_OPEN_IL_OUTPUT_FILE

cannot open IL output file

0568 CL_INVALID_C_OUTPUT_FILE

invalid C output file

0569 CL_CANNOT_OPEN_C_OUTPUT_FILE

cannot open C output file

0570 CL_ERROR_IN_DEBUG_OPTION_ARGUMENT

error in debug option argument

0571 CL_INVALID_OPTION

invalid option:

0572 CL_BACK_END_REQUIRES_IL_FILE

back end requires name of IL file

0573 CL_COULD_NOT_OPEN_IL_FILE

could not open IL file

0574 CL_INVALID_NUMBER

invalid number:

0575 CL_INCORRECT_HOST_ID

incorrect host CPU id

0576 CL_INVALID_INSTANTIATION_MODE

invalid instantiation mode:

0578 CL_INVALID_ERROR_LIMIT

invalid error limit:

0579 CL_INVALID_RAW_LISTING_OUTPUT_FILE

invalid raw-listing output file

0580 CL_CANNOT_OPEN_RAW_LISTING_OUTPUT_FILE

cannot open raw-listing output file

0581 CL_INVALID_XREF_OUTPUT_FILE

invalid cross-reference output file

0582 CL_CANNOT_OPEN_XREF_OUTPUT_FILE

cannot open cross-reference output file

0583 CL_INVALID_ERROR_OUTPUT_FILE

invalid error output file

0584 CL_CANNOT_OPEN_ERROR_OUTPUT_FILE

cannot open error output file

0585 CL_VTBL_OPTION_ONLY_IN_CPLUSPLUS

virtual function tables can only be suppressed when compiling C++

0586 CL_ANACHRONISM_OPTION_ONLY_IN_CPLUSPLUS

anachronism option can be used only when compiling C++

0587 CL_INSTANTIATION_OPTION_ONLY_IN_CPLUSPLUS

instantiation mode option can be used only when compiling C++

0588 CL_AUTO_INSTANTIATION_OPTION_ONLY_IN_CPLUSPLUS

automatic instantiation mode can be used only when compiling C++

0589 CL_IMPLICIT_INCLUSION_OPTION_ONLY_IN_CPLUSPLUS

implicit template inclusion mode can be used only when compiling C++

0590 CL_EXCEPTIONS_OPTION_ONLY_IN_CPLUSPLUS

exception handling option can be used only when compiling C++

0591 CL_STRICT_ANSI_INCOMPATIBLE_WITH_PCC

strict ANSI mode is incompatible with K&R mode

0592 CL_STRICT_ANSI_INCOMPATIBLE_WITH_CFRONT

strict ANSI mode is incompatible with cfront mode

0593 CL_MISSING_SOURCE_FILE_NAME

missing source file name

0594 CL_OUTPUT_FILE_INCOMPATIBLE_WITH_MULTIPLE_INPUTS

output files may not be specified when compiling several input files

0595 CL_TOO_MANY_ARGUMENTS

too many arguments on command line

0596 CL_NO_OUTPUT_FILE_NEEDED

an output file was specified, but none is needed

0597 CL_IL_DISPLAY_REQUIRES_IL_FILE_NAME

IL display requires name of IL file

0598 VOID_TEMPLATE_PARAMETER

a template parameter may not have void type

0599 TOO_MANY_UNUSED_INSTANTIATIONS

excessive recursive instantiation of *entity-kind* "*entity*" due to instantiate-all mode

0600 CL_STRICT_ANSI_INCOMPATIBLE_WITH_ANACHRONISMS

strict ANSI mode is incompatible with allowing anachronisms

0601 VOID_THROW

a throw expression may not have void type

0602 CL_TIM_LOCAL_CONFLICTS_WITH_AUTO_INSTANTIATION

local instantiation mode is incompatible with automatic instantiation

0603 ABSTRACT_CLASS_PARAM_TYPE

parameter of abstract class type is not allowed

0604 ARRAY_OF_ABSTRACT_CLASS

array of abstract class is not allowed

0605 FLOAT_TEMPLATE_PARAMETER

floating-point template parameter is nonstandard

0606 PRAGMA_MUST_PRECEDE_DECLARATION

this pragma must immediately precede a declaration

0607 PRAGMA_MUST_PRECEDE_STATEMENT

this pragma must immediately precede a statement

0608 PRAGMA_MUST_PRECEDE_DECL_OR_STMT

this pragma must immediately precede a declaration or statement

0609 PRAGMA_MAY_NOT_BE_USED_HERE

this kind of pragma may not be used here

0610 NONOVERRIDING_FUNCTION_DECL

entity-kind "*entity*" does not match "*entity*" -- virtual function override intended?

0611 PARTIAL_OVERRIDE

overloaded virtual function "*entity*" is only partially overridden in *entity-kind* "*entity*"

0612 SPECIALIZATION_OF_CALLED_INLINE_TEMPLATE_FUNCTION

specific definition of inline template function must precede its first use

0613 CL_INVALID_ERROR_TAG

invalid error tag:

0614 CL_INVALID_ERROR_NUMBER

invalid error number:

0615 PARAM_TYPE_PTR_TO_ARRAY_OF_UNKNOWN_BOUND

parameter type involves pointer to array of unknown bound

0616 PARAM_TYPE_REF_ARRAY_OF_UNKNOWN_BOUND

parameter type involves reference to array of unknown bound

0617 PTR_TO_MEMBER_CAST_TO_PTR_TO_FUNCTION

pointer-to-member-function cast to pointer to function

0618 NO_NAMED_FIELDS

struct or union must declare at least one named field

0619 NONSTD_UNNAMED_FIELD

nonstandard unnamed field

0620 NONSTD_UNNAMED_MEMBER

nonstandard unnamed member

0621 FUNCTION_TYPE_IN_TEMPLATE_ARG

a function type cannot be used as a template argument

0622 CL_INVALID_PCH_OUTPUT_FILE

invalid precompiled header output file

0623 CL_CANNOT_OPEN_PCH_OUTPUT_FILE

cannot open precompiled header output file

0624 NOT_A_TYPE_NAME

"*xxxx*" is not a type name

0625 CL_CANNOT_OPEN_PCH_INPUT_FILE

cannot open precompiled header input file

0626 INVALID_PCH_FILE

precompiled header file "*xxxx*" is either invalid or not generated by this version of the compiler

0627 PCH_CURR_DIRECTORY_CHANGED

precompiled header file "*xxxx*" was not generated in this directory

0628 PCH_HEADER_FILES_HAVE_CHANGED

header files used to generate precompiled header file "*xxxx*" have changed

0629 PCH_CMD_LINE_OPTION_MISMATCH

the command line options do not match those used when precompiled header file "*xxxx*" was created

0630 PCH_FILE_PREFIX_MISMATCH

the initial sequence of preprocessing directives is not compatible with those of precompiled header file "*xxxx*"

0631 UNABLE_TO_GET_MAPPED_MEMORY

unable to obtain mapped memory

0632 USING_PCH

"*xxxx*": using precompiled header file "*xxxx*"

0633 CREATING_PCH

"*xxxx*": creating precompiled header file "*xxxx*"

0634 MEMORY_MISMATCH

memory usage conflict with precompiled header file "*xxxx*"

0635 CL_INVALID_PCH_SIZE

invalid PCH memory size

0636 CL_PCH_MUST_BE_FIRST

PCH options must appear first in the command line

0637 OUT_OF_MEMORY_DURING_PCH_ALLOCATION

insufficient memory for PCH memory allocation

0638 CL_PCH_INCOMPATIBLE_WITH_MULTIPLE_INPUTS

precompiled header files may not be used when compiling several input files

0639 NOT_ENOUGH_PREALLOCATED_MEMORY

insufficient preallocated memory for generation of precompiled header file (*xxxx* bytes required)

0640 PROGRAM_ENTITY_TOO_LARGE_FOR_PCH

very large entity in program prevents generation of precompiled header file

0641 CANNOT_CHDIR

"*xxxx*" is not a valid directory

0642 CANNOT_BUILD_TEMP_FILE_NAME

cannot build temporary file name

0643 RESTRICT_NOT_ALLOWED

"restrict" is not allowed

0644 RESTRICT_POINTER_TO_FUNCTION

a pointer or reference to function type may not be qualified by "restrict"

0645 BAD_DECLSPEC_MODIFIER

"*xxxx*" is an invalid __declspec attribute

0646 CALLING_CONVENTION_NOT_ALLOWED

a calling convention modifier may not be specified here

0647 CONFLICTING_CALLING_CONVENTIONS

conflicting calling convention modifiers

0648 CL_STRICT_ANSI_INCOMPATIBLE_WITH_MICROSOFT

strict ANSI mode is incompatible with Microsoft mode

0649 CL_CFRONT_INCOMPATIBLE_WITH_MICROSOFT

cfront mode is incompatible with Microsoft mode

0650 CALLING_CONVENTION_IGNORED

calling convention specified here is ignored

0651 CALLING_CONVENTION_MAY_NOT_PRECEDE_NESTED_DECLARATOR

a calling convention may not be followed by a nested declarator

0652 CALLING_CONVENTION_IGNORED_FOR_TYPE

calling convention is ignored for this type

0653 CALLING_CONVENTION_NOT_ALLOWED_FOR_TYPE

calling conventions may only be applied to function types

0654 DECL_MODIFIERS_INCOMPATIBLE_WITH_PREVIOUS_DECL

declaration modifiers are incompatible with previous declaration

0655 DECL_MODIFIERS_INVALID_FOR_THIS_DECL

the modifier "*xxxx*" is not allowed on this declaration

0656 BRANCH_INTO_TRY_BLOCK

transfer of control into a try block

0657 INCOMPATIBLE_INLINE_SPECIFIER_ON_SPECIFIC_DECL

inline specification is incompatible with previous "*entity*" (declared at line *xxxx*)

0658 TEMPLATE_MISSING_CLOSING_BRACE

closing brace of *entity-kind* "*entity*" not found

0659 CL_WCHAR_T_OPTION_ONLY_IN_CPLUSPLUS

wchar_t keyword option can be used only when compiling C++

0660 BAD_PACK_ALIGNMENT

invalid packing alignment value

0661 EXP_INT_CONSTANT

expected an integer constant

0662 CALL_OF_PURE_VIRTUAL

call of pure virtual function

0663 BAD_IDENT_STRING

invalid source file identifier string

0665 ASM_NOT_ALLOWED

"asm" is not allowed

0666 BAD_ASM_FUNCTION_DEF

"asm" must be used with a function definition

0667 NONSTD_ASM_FUNCTION

"asm" function is nonstandard

0668 NONSTD_ELLIPSIS_ONLY_PARAM

ellipsis with no explicit parameters is nonstandard

0669 NONSTD_ADDRESS_OF_ELLIPSIS

"&..." is nonstandard

0670 BAD_ADDRESS_OF_ELLIPSIS

invalid use of "&..."

0671 CL_ALTERNATIVE_TOKEN_OPTION_ONLY_IN_CPLUSPLUS

alternative token option can be used only when compiling C++

0672 CONST_VOLATILE_REF_INIT_ANACHRONISM

temporary used for initial value of reference to const volatile (anachronism)

0673 BAD_CONST_VOLATILE_REF_INIT

initial value of reference to const volatile has incorrect type

0674 CONST_VOLATILE_REF_INIT_FROM_RVALUE

initial value of reference to const volatile must be an lvalue

0675 CL_SVR4_C_OPTION_ONLY_IN_ANSI_C

SVR4 C compatibility option can be used only when compiling ANSI C

0676 USING_OUT_OF_SCOPE_DECLARATION

using out-of-scope declaration of *entity-kind* "*entity*" (declared at line *xxxx*)

0677 CL_STRICT_ANSI_INCOMPATIBLE_WITH_SVR4

strict ANSI mode is incompatible with SVR4 C mode

0678 CANNOT_INLINE_CALL

call of *entity-kind* "*entity*" cannot be inlined

0679 CANNOT_INLINE

entity-kind "*entity*" cannot be inlined

0680 CL_INVALID_PCH_DIRECTORY

invalid PCH directory:

0681 EXP_EXCEPT_OR_FINALLY

expected __except or __finally

0682 LEAVE_MUST_BE_IN_TRY

a __leave statement may only be used within a __try

0688 NOT_FOUND_ON_PACK_ALIGNMENT_STACK

"*xxxx*" not found on pack alignment stack

0689 EMPTY_PACK_ALIGNMENT_STACK

empty pack alignment stack

0690 CL_RTTI_OPTION_ONLY_IN_CPLUSPLUS

RTTI option can be used only when compiling C++

0691 INACCESSIBLE_ELIDED_CCTOR

entity-kind "*entity*", required for copy that was eliminated, is inaccessible

0692 UNCALLABLE_ELIDED_CCTOR

entity-kind "*entity*", required for copy that was eliminated, is not callable because reference parameter cannot be bound to rvalue

0693 TYPEID_NEEDS_TYPEINFO

<typeinfo> should be included before typeid is used

0694 CANNOT_CAST_AWAY_CONST

xxxx cannot cast away const or other type qualifiers

0695 BAD_DYNAMIC_CAST_TYPE

the type in a dynamic_cast must be a pointer or reference to a complete class type, or void *

0696 BAD_PTR_DYNAMIC_CAST_OPERAND

the operand of a pointer dynamic_cast must be a pointer to a complete class type

0697 BAD_REF_DYNAMIC_CAST_OPERAND

the operand of a reference dynamic_cast must be an lvalue of a complete class type

0698 DYNAMIC_CAST_OPERAND_MUST_BE_POLYMORPHIC

the operand of a runtime dynamic_cast must have a polymorphic class type

0699 CL_BOOL_OPTION_ONLY_IN_CPLUSPLUS

bool option can be used only when compiling C++

0700 BAD_STORAGE_CLASS_ON_CONDITION_DECL

invalid storage class for condition declaration

0701 ARRAY_TYPE_NOT_ALLOWED

an array type is not allowed here

0702 EXP_ASSIGN

expected an “=”

0703 EXP_DECLARATOR_IN_CONDITION_DECL

expected a declarator in condition declaration

0704 REDECLARATION_OF_CONDITION_DECL_NAME

"*xxxx*", declared in condition, may not be redeclared in this scope

0705 DEFAULT_TEMPLATE_ARG_NOT_ALLOWED

default template arguments are not allowed for function templates

0706 EXP_COMMA_OR_GT

expected a "," or ">"

0707 MISSING_TEMPLATE_PARAM_LIST

expected a template parameter list

0708 INCR_OF_BOOL_DEPRECATED

incrementing a bool value is deprecated

0709 BOOL_TYPE_NOT_ALLOWED

bool type is not allowed

0710 BASE_CLASS_OFFSET_TOO_LARGE

offset of base class "*entity*" within class "*entity*" is too large

0711 EXPR_NOT_BOOL

expression must have bool type (or be convertible to bool)

0712 CL_ARRAY_NEW_AND_DELETE_OPTION_ONLY_IN_CPLUSPLUS

array new and delete option can be used only when compiling C++

0713 BASED_REQUIRES_VARIABLE_NAME

entity-kind "*entity*" is not a variable name

0714 BASED_NOT_ALLOWED_HERE

__based modifier is not allowed here

0715 BASED_NOT_FOLLOWED_BY_STAR

__based does not precede a pointer operator, __based ignored

0716 BASED_VAR_MUST_BE_PTR

variable in __based modifier must have pointer type

0717 BAD_CONST_CAST_TYPE

the type in a const_cast must be a pointer, reference, or pointer to member to an object type

0718 BAD_CONST_CAST

a const_cast can only adjust type qualifiers; it cannot change the underlying type

0719 MUTABLE_NOT_ALLOWED

mutable is not allowed

0720 CANNOT_CHANGE_ACCESS

redeclaration of *entity-kind* "*entity*" is not allowed to alter its access

0721 NONSTD_PRINTF_FORMAT_STRING

nonstandard format string conversion

0722 PROBABLE_INADVERTENT_LBRACKET_DIGRAPH

use of alternative token "<:" appears to be unintended

0723 PROBABLE_INADVERTENT_SHARP_DIGRAPH

use of alternative token "%%:" appears to be unintended

0724 NAMESPACE_DEF_NOT_ALLOWED

namespace definition is not allowed

0725 MISSING_NAMESPACE_NAME

namespace name is required

0726 NAMESPACE_ALIAS_DEF_NOT_ALLOWED

namespace alias definition is not allowed

0727 NAMESPACE_QUALIFIED_NAME_REQUIRED

namespace-qualified name is required

0728 NAMESPACE_NAME_NOT_ALLOWED

a namespace name is not allowed

0729 BAD_COMBINATION_OF_DLL_ATTRIBUTES

invalid combination of DLL attributes

0730 SYM_NOT_A_CLASS_TEMPLATE

entity-kind "*entity*" is not a class template

0731 ARRAY_OF_INCOMPLETE_TYPE

array with incomplete element type is nonstandard

0732 ALLOCATION_OPERATOR_IN_NAMESPACE

allocation operator may not be declared in a namespace

0733 DEALLOCATION_OPERATOR_IN_NAMESPACE

deallocation operator may not be declared in a namespace

0734 CONFLICTS_WITH_USING_DECL

entity-kind "*entity*" conflicts with using-declaration of *entity-kind* "*entity*"

0735 USING_DECL_CONFLICTS_WITH_PREV_DECL

using-declaration of *entity-kind* "*entity*" conflicts with *entity-kind* "*entity*" (declared at line *xxxx*)

0736 CL_NAMESPACES_OPTION_ONLY_IN_CPLUSPLUS

namespaces option can be used only when compiling C++

0737 USELESS_USING_DECLARATION

using-declaration ignored -- it refers to the current namespace

0738 CLASS_QUALIFIED_NAME_REQUIRED

a class-qualified name is required

0739 ARGUMENT_LIST_TYPES_ADD_ON

argument types are: (*xxxx*)

0740 OPERAND_TYPES_ADD_ON

operand types are: *xxxx*

0741 USING_DECLARATION_IGNORED

using-declaration of *entity-kind* "*entity*" ignored

0742 NOT_AN_ACTUAL_MEMBER

entity-kind "*entity*" has no actual member "*xxxx*"

0743 NONSTD_GLOBAL_QUALIFIER_ON_FRIEND_DECL

global-scope qualifier (leading "::") on friend declaration is nonstandard

0744 MEM_ATTRIB_INCOMPATIBLE

incompatible memory attributes specified

0745 MEM_ATTRIB_IGNORED

memory attribute ignored

0746 MEM_ATTRIB_MAY_NOT_PRECEDE_NESTED_DECLARATOR

memory attribute may not be followed by a nested declarator

0747 DUPL_MEM_ATTRIB

memory attribute specified more than once

0748 DUPL_CALLING_CONVENTION

calling convention specified more than once

0749 TYPE_QUALIFIER_NOT_ALLOWED

a type qualifier is not allowed

0750 TEMPLATE_INSTANCE_ALREADY_USED

entity-kind "*entity*" (declared at line *xxxx*) was used before its template was declared

0751 STATIC_NONSTATIC_WITH_SAME_PARAM_TYPES

static and nonstatic member functions with same parameter types cannot be overloaded

0752 NO_PRIOR_DECLARATION

no prior declaration of *entity-kind* "*entity*"

0753 TEMPLATE_ID_NOT_ALLOWED

a template-id is not allowed

0754 CLASS_QUALIFIED_NAME_NOT_ALLOWED

a class-qualified name is not allowed

0755 BAD_SCOPE_FOR_REDECLARATION

entity-kind "*entity*" may not be redeclared in the current scope

0756 QUALIFIER_IN_NAMESPACE_MEMBER_DECL

qualified name is not allowed in namespace member declaration

Index

D

E

F

G

H

I

J

K

L

R

S