3.3 Release Notes: C Compiler

This file contains release notes for for the 3.3, 3.2, 3.1, and 3.0 releases of the GNU C Compiler. Items specific to the 3.3 release are listed first, and the Release 3.2, 3.1 and 3.0 notes follow.

Notes Specific to Release 3.3

In this release, the compiler is based on the GNU C compiler version 2.5.8.

Recompilation Requirements

You need to recompile existing code under certain circumstances:

- **C++ name mangling change.** The method of ªmanglingº C++ function names has been changed. You must recompile all C++ programs when you

start using NEXTSTEP's 3.3. **libg++** is updated to use the new mangling scheme.

New Features

The following featuresÐcategorized by languageÐhave been added to the compiler for NEXTSTEP Release 3.3:

C Language Features

- **Better type checking**. The compiler's ability to do static type checking has been considerably improved. However, some programs that previously compiled may now fail to compile correctly because the compiler catches type errors it previously ignored.
- **Volatile and const declarations must match definitions.** The C compiler used to allow a declaration and its definition to mismatch with respect to **volatile** and **const** storage class specifiers. The compiler now issues an error for a declaration/definition mismatchÐfor both variables and functions. This example shows the problem for a variable:

```
// First file
extern int time_counter; // declaration
...
```

```
   // End of file

   // Second file
   volatile int time_counter;// definition: adds volatile
specifierÐ
```**Error**
```
   ...
   // End of file
```

- **Implicit cast from int to enums.** The compiler used to allow implicit casts from **int** to any **enum** type. According to both the ANSI C and C++ standards, this isn't correct behavior, so a warning is issued for such casts. Casting is a problem in this case because the integer value may not lie in the range of the **enum** type. The following example illustrates the difficulty:

```
   int counter = 12;
   enum colors{
   red, orange, yellow, green = 20, blue, indigo, violet} aColor;
   aColor = counter;   // Warning:  casts int to enum
```

  Since red == 0, orange == 1, yellow == 2, green == 20, and so on, an **enum** of type colors can't take on the value 12. No valid conversion is possible.

- **Enum arithmetic.** The compiler used to allow increment and decrement operations on **enum**s. This is forbidden in the ANSI C++ standard, so a warning is issued. The problem is that the resulting integer value may not lie in the range of the **enum** type. For instance, consider the **aColor enum** of the previous example:

```
   aColor = yellow;    // aColor = 2
```

```
aColor++; // Warning:  aColor = 3?
```

If **aColor** == 3, then it's out of range of the **enum** colors type since yellow == 2 and green == 20.

- **Assignment used as a conditional.** When the **-Wall** option is turned on, the compiler issues a warning for assignments used as conditionals in **if**, **for**, and **while** statements. For example, the code

```
if (i = generate()) { ... }
```

generates a warning, suggesting an extra set of parenthesis around the assignment
such as

```
if ((i = generate())) { ... }
```

This warning is intended to catch situations where you really meant to test for equivalence ==, not perform an assignment =. The **-Wno-parentheses** flag turns off
this warning.

- **Array and structure initializers.** An initializer's C syntax for assigning a value to a structure field is now ª*.field_name*=º. The corresponding syntax for array initializers is now ª*[index]*=º. Some examples of this usage follow:

```
NXRect point = {.origin={0,0}, .size={2,3}};
char whitespace[256] = { [' ']=1, ['\t']=1, ['\n']=1 };
```

This was changed to agree with the syntax proposed by the Numerical C Extensions Group (NCEG). Although NeXT's C++ doesn't support these kinds of initializers, the C compiler handles them properly.

- **-fkeep-inline-functions option.** Previous versions of the compiler eliminated unused static inline functions. This flag forces them to be compiled into the image.

- **-Wno-format.** This warning option used to be called **-Wnoformat** (without the dash), but has been renamed to be consistent with the rest of the compiler flags. Both forms are accepted, but the newer syntax is recommended.

- **Nested functions.** Pascal-style nested functions are now supported in C. If you use this feature, the resulting code will be incompatible with pre-3.3 systems. See the **gcc** manual for more information.

Objective C Language Features

- **Accessing instance variables in class methods.** It used to be common programming style in Objective C to assign to **self** in a class method and then access instance variables. This is bad style because **self** in the context of a class method stands for the class objectÐand shouldn't be redefined to stand for a particular *instance* of the object.

Here is an example of this *bad* style:

```
@implementation Oval : Object
{
    int x;
}
+new {
    self = [super new];  // Now self refers to a class instance
     ...
    x = 4;     // Assigns an instance variable
} ...
@end
...
x = [Oval new];     // Create an Oval object
```

To discourage this anachronistic use, the compiler issues a warning if an instance variable is referenced in a class method.

Here is a better way to instantiate an object:

```
x = [[Oval alloc] init];
```

See *Object-Oriented Programming and the Objective C Language* for more details.

- **Stricter Objective C syntax checking.** The compiler's syntax checking is now stricter, so you aren't allowed to nest **@interface** and **@implementation** blocks.

C++ Language Features

- **C++ support.** The compiler now supports ANSI C++.
- **libg++ support.** The GNU C++ library, **libg++**, provides a variety of C++ programming tools and other support to C++ programmers. **libg++** is similar to some extent to AT&T's **libC.a**. The library and full documentation is in **/NextLibrary/Documentation/GNU-libg++**. Also see the Release Note **libg++.rtf**.
- **Objective C messages to converted C++ objects (ªsmart pointersº).** You can now send an Objective C message to a C++ object that has been converted by a conversion operator. In the following example, the C++ ptrSquare object **aSquare** is implicitly converted to the Objective C type Square* using the conversion **operator Square*()**. The converted object receives the message **calculateArea**:

```
@interface Square { id a; } ...
@end

class ptrSquare {
    Square* value;
public:
    operator Square*();
};
```

```
   square (ptrSquare aSquare) {
       float z = [aSquare calculateArea]; // invokes operator
Square*()
      }
```

Due to the conversion, the compiler acts as if **aSquare** is statically typed to Square* in the message expression.

The above example uses only one conversion operator: **operator Square***. You should avoid having multiple conversion operators in the same class that produce different pointer typesÐthe compiler may choose the wrong conversion operator and not produce the desired type. If you need more than one conversion type, you must use an **operator id** conversion operatorÐthe compiler chooses this over an operator converting to any other Objective C class pointer type. If the class ptrSquare implemented other **operator X*()** conversions besides **operator Square*()**, it would also have to implement an **operator id** conversion so the compiler would know which conversion to look for.

Conversion operators allow you to implement so called ªsmart pointersº to Objective C objects. Smart pointers are objects that act like pointers and perform some other action in addition whenever an object is accessed through them. For more information on smart pointers, see Bjarne Stroustrup's *The C++ Programming Language, Second Edition* (Addison-Wesley, 1991).

- **C++ multiple virtual inheritance.** The C++ compiler now invokes virtual

functions correctlyÐexcept when a non-virtual function is redeclared as virtual in a subclass. The compiler issues a warning in this case, however.

In this example, the function **f()** in class Animal is redeclared **virtual** in the subclass Mammal:

```
   class Animal             { void f(); }
   class Mammal : public virtual Animal    { virtual void f(); }
   class Quadruped : public virtual Animal { virtual void f(); }
   class Dog : public Mammal, public Quadruped  { virtual void
f(); }
   class Terrier : public Dog         { virtual void f(); }
```

Invoking the method **f()** gives the wrong result in the following case:

```
void zoo(void) {
    Terrier* terrier = new Terrier;
    Mammal* mammal = terrier;
    Quadruped* quadruped = terrier;
    Dog* dog = terrier;

    quadruped->f();    // Wrong - invokes Dog::f()
    mammal->f();  // Right - invokes Terrier::f()
    dog->f();     // Right - invokes Terrier::f()
}
```

The compiler now warns that wrong code may be generated:

```
   warning: method `Animal::f()' redeclared as `virtual
Mammal::f()'
```

If you modify the above hierarchy by making the function **f()** in Animal virtual, the invocation works correctly. The workaround is therefore to make **f()** virtual throughout the hierarchy:

```
class Animal              { virtual void f(); }
class Mammal : public virtual Animal    { virtual void f(); }
class Quadruped : public virtual Animal { virtual void f(); }
class Dog : public Mammal, public Quadruped  { virtual void
f(); }
class Terrier : public Dog          { virtual void f(); }
```

- **Pointers to member functions.** The C++ compiler used to allow using member function pointers with objects that might not recognize the pointer or its contents. The compiler now flags these as errors. Here's an example of such errors:

```
class Mammal { public: void f(int); };
class Cat : public Mammal { public: void f(int); };

void g (Cat* aCat, Mammal* aMammal) {
  void (Mammal::*mammal_f_ptr)(int) = &Mammal::f;
  void (Cat::*cat_f_ptr)(int) = &Cat::f;

 (aCat->*mammal_f_ptr)(4);     // OK
 (aMammal->*cat_f_ptr)(5);     // Error (1)
 cat_f_ptr = &Cat::f;     // OK
 mammal_f_ptr = &Cat::f;       // Error (2)
}
```

The local variables **mammal_f_ptr** and **cat_f_ptr** are pointers to member functions, and the function g initializes them to point to the class Cat member function, **f**. It then attempts to invoke this function through these pointers. Statement (1) is an error because you can't be sure than a Mammal object, **aMammal**, ªresponds toº a Cat member pointer, **cat_f_ptr**Ðespecially since **cat_f_ptr** points to a Cat member function that Mammal would know nothing about. Even if **cat_f_ptr** were initialized to a Mammal member function, **cat_f_ptr** cannot safely be applied to a Mammal object. The assignment in (2) is an error because you cannot be sure that some member function
of a derived class (in this case Cat::**f**) is available in any of its base classes (in this
case Mammal).

- **Implicit cast from void* to C++ object pointer.** The C++ compiler used to implicitly allow casts from **void*** to any C++ object pointer type. This isn't allowed in the ANSI C++ standard, so a warning is issued when such a cast is detected. If aClass is some arbitrary class, the following cast produces a warning:

```
void *vp1;
aClass *obj1, *obj2;
vp1 = &obj1;
obj2 = vp1;    // Warning:  casts void pointer
```

- **#pragma cplusplus.** This new pragma is used to resolve the problem of having C++ system header files. All system header files are by default

included in implicit **extern "C"**. When #**pragma cplusplus** appears in a header file, the rest of that file is embedded in an implicit **extern "C++"** block. Alternatively, if either **g++** or **c++** appears in the full path name to a header file (ignoring case), it is also considered to be C++.
An error is reported if this pragma appears inside an explicit **extern "C" {...}**.

Known Problems

These problems exist in version 2.5.8 of the compiler:

Reference: 39034, 45027

Problem: A **static** pointer to an Objective C subclass can't be passed to a C++ member function.

Description: C++ has no knowledge of the Objective C class hierarchy. C++ code containing **static** references to sub-classed Objective C objects will not compile.

Workaround: Remove the **static** qualifier.

Reference: 40546

Problem: Complex number support is unreliable.

Description: The complex number handling part of the compiler is untested and has been found to be erroneous in some cases. It is not recommended that this feature be used.

Workaround: Define your own complex number class with the methods you need.

Reference: 41950

Problem: Immediate long doubles don't work sometimes.

Description: Using a form such as 23450.0L to write an immediate long double value may cause the compiler to report an internal error.

Workaround: None.

Reference: 42975

Problem: Bitfields may not work properly with **-O2** optimization.

Description: If you use **-O2** optimization, the compiler may return a wrong value in a bitfield.

Workaround: Don't use **-O2** optimization.

Reference: 44109

Problem: Programs with detectable NaN to **int** conversion won't compile with optimization on PA-RISC systems.

Description: If you attempt to compile a program with a conversion on a non-number to an **int** with **-O** optimization, the compile fails. In other compiler versions, the compiler emits a warning about the Nan to **int** conversion.

Workaround: Don't use **-O** optimization.

Notes Specific to Release 3.2

New Features

The following new features have been added to the GNU C Compiler for Release 3.2.

- **Support for RTF source code.** Rich Text Format files can now be compiled. The preprocessor strips out all RTF directives, leaving only ASCII text for the compiler itself. See the **Preprocessor.rtf** release note for more information.

- **Automatic searching for C++ headers.** When compiling a C++ file (extension

.C, **.M**, or **.cc**), the compiler adds **/NextDeveloper/Headers/g++** to its header search path. This allows **libg++** classes to be used without having to specify additional command-line options.

Notes Specific to Release 3.1

GNU C Compiler

The Release 3.1 Objective C compiler (as well as the Objective C++ compiler) is based on version 2.2.2 of the GNU Compiler (the Release 3.0 compiler was based on version 1.93). NeXT's implementation of Objective C has been integrated into the version 2.2.2 GNU sources.

New Features

The following new features have been added to the GNU C Compiler since Release 3.0.

- You can specify the target architecture you are compiling for with **-arch** *arch_type*. The full list of values for *arch_type* can be found in **arch**(3), but for now, you should only use the values **m68k** and **i386** (i386 is the processor family of the i486 processor). The option **-arch** *arch_type* specifies the target architecture, *arch_type*, of the operations to be performed. The operations affected by **-arch**

are: preprocessing, precompiling, compiling, assembling, and linking. The specification of multiple architectures results in the production of ``fat'' output files and the creation of multiple ``thin'' intermediate files from each stage. It is an error to use **-E**, **-S**, **-M**, and **-MM** with multiple architectures as the output form is textual in these cases.

- The implied meaning of file name suffixes has been extended. The following suffixes are now recognized:

  file.**c** contains C source code.
  file.**m** contains Objective C source code.
  file.**C** and *file*.**cc** contains C++ source code.
  file.**M** contains mixed Objective C and C++ source code.

- Optimizations performed by the compiler are more robust. The **-fstrength-reduce** optimization is now beneficial and is enabled by **-O**. The **-fcombine-regs** optimization is selected as part of **-O**, and has been dropped as an option.

- Several new optimizations have been added, primarily to support the i386 architecture. Most of these new optimizations are enabled by the **-O3** and **-O4** switches and are only implemented in the i386 compiler. Each optimization may be individually turned on or off by using the flag **-f***opt* or **-fno-***opt*.

Optimization	**Enabled by**	**Description**
-fcse-skip-blocks	**-O2**	Similar to **-fcse-follow-jumps**, but causes CSE

		to follow jumps which conditionally skip over blocks. When CSE encounters a simple **if** statement with no **else** clause, this option causes CSE to follow the jump around the body of the **if**. *This switch isn't specific to the i386 architecture.*
-fcompare-elim	**-O3**	Attempt to remove the compare at the end of a loop by having the loop counter count up to zero. This results in a more efficient conditional jump at the end of the loop.
-fcomplex-givs	**-O3**	Enable strength reduction of generalized induction variables that require more than one instruction for initialization.
-fcopy-prop	**-O3**	Strength reduction and loop unrolling may copy one register to another before the loop and use the copy within the loop. Here, an attempt is made to use the original in the loop and eliminate the copy.
-fexpose-invars	**-O3**	Enable rewriting of expressions in a loop with invariant subexpressions so as to combine and create more invariant subexpressions.
-fjump-back	**-O3**	Attempt to perform strength reduction in outer loops as well as inner loops.
-fspl	**-O3**	Attempt to rotate a loop that terminates with a

	floating point store so the store is at the top of the rotated loop.
-fschedule-insns -O4	As described in *NEXTSTEP Development Tools and Techniques*. *This switch isn't specific to the i386 architecture.*
-fschedule-insns2 -O4	As described in *NEXTSTEP Development Tools and Techniques*. *This switch isn't specific to the i386 architecture.*

The **-fcaller-saves** optimization generally makes code worse on the i486, so it is not enabled by default at any optimization level when you use **-arch i386**. The new optimizations are subject to change. It is likely that these will be combined into a smaller set of options. These will also be made machine-independent and will eventually be implemented for all target architectures.

There are some inconsistencies in the current mapping of optimization levels between **-arch i386** and **-arch m68k**. We expect to correct these in a future release. These are as follows:

- When you choose **-arch i386**, **-O** is taken to mean **-O2** instead of **-O1**.
- The option **-O3** includes **-fomit-frame-pointer**, which makes debugging impossible. In the future, you will need to explicitly choose **-fomit-frame-pointer**, but for now, you should use **-fno-omit-frame-pointer** when you use **-O3**.

- The options **-O3 -fno-omit-frame-pointer -fschedule-insns2** generally produce the best code. To simplify the interface, **-O4** will be eliminated as a distinct option and **-fschedule-insns2** will be added to **-O3**.

- The compiler now has additional predefined macros that can be used to determine the release version of the compiler. *Every effort should be made to minimize the use of these macros.* For each release of the compiler there will be a macro defined such as `NX_COMPILER_RELEASE_3_0` and `NX_COMPILER_RELEASE_3_1`. There will also be a macro `NX_CURRENT_COMPILER_RELEASE`. One can conditionally compile code by numerically comparing these macros. For example:

```
#if NX_CURRENT_COMPILER_RELEASE > NX_COMPILER_RELEASE_3_0

#endif
```

  Note that if one wants to use these macros in code to be compiled with any compiler before release 3.1 they must additionally check whether the macro is defined, as in:

```
#if defined(NX_CURRENT_COMPILER_RELEASE) && \
    NX_CURRENT_COMPILER_RELEASE > NX_COMPILER_RELEASE_3_0

#endif
```

- The compiler detects and generates a warning for duplicate interface declarations.

- The compiler has been changed in the way it determines whether a class implements a protocol. In order for a class to implement a protocol, P, it must (1) implement all of the methods in P and (2) either (a) implement all the methods in any protocols adopted by P or (b) have a superclass which adopts the protocols adopted by P.

Notes Specific to Release 3.0

These notes were included with the Release 3.0 version of the GNU C Compiler. Sections that are no longer relevant have been marked with an italicized comment.

For more about these and related topics, see the following chapters in *NEXTSTEP Development Tools and Techniques*:

- Chapter 1: ªPutting Together a NeXT Applicationº
- Chapter 6: ªThe GNU C Compilerº
- Chapter 7: ªThe GNU C Preprocessorº

GNU C Compiler

The Release 3.0 Objective C compiler is based on version 1.93 of the GNU Compiler

(the Release 2.0 compiler was based on version 1.36). NeXT's implementation of Objective C has been integrated into the version 2.0 GNU sources. Significant improvements have been made to certain compiler optimizations, including the addition of a loop unrolling facility.

Incompatible Changes

Incompatible Storage Class Specifier Bug Fix

A compiler bug relating to incompatible storage class specifiers has been fixed which may cause source incompatibilities. ANSI C specifies that a function which is forward-declared to be **static** should not be globally visible, even if the definition of the function does not contain the **static** keyword. The Release-2.0 compiler incorrectly gave precedence to the definition of the function over the its forward-declaration and made the symbol global. For example, the program:

```
static int foo (int x);

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

will no longer cause the symbol **foo** to be global. If the function should be global,

then change the forward-declaration to use **extern** rather than **static**.

Objective C in C Source Files Produces Error

In the Release 3.0 compiler, detecting Objective C syntax in a C language source file is an error rather than a warning. Use the **-ObjC** flag to specify that the file should be compiled as an Objective C program.

Reversal of long long Word Ordering

In the Release 2.0 compiler, the 64-bit integer type **long long** was implemented with little-endian word ordering. This was inconsistent with the 68k family's byte ordering, which is big-endian. For Release 3.0 the word ordering of the **long long** type has been reversed and is now big-endian. A new syntax for specifying long long constants has also been introduced. To specify a **long long** constant, use the suffix **LL** or **ULL**. For example:

```
long long big_integer = -1LL;
unsigned long long big_unsigned = 0xffffffffffffffffULL;
```

New Features

New Optimization Levels

The Release 3.0 compiler supports several new optimizations. Most of these new optimizations are not enabled by the **-O** switch, but only by the new **-O2** switch. Each optimization may be individually turned on or off by using the flag **-f***opt* or **-fno-***opt*.

- **-fthread-jumps** (enabled by **-O**)
- **-fcse-follow-jumps** (enabled by **-O2**)
- **-fexpensive-optimizations** (enabled by **-O2**)
- **-frerun-cse-after-loop** (enabled by **-O2**)
- **-fstrength-reduce**
- **-funroll-loops**

The **-fstrength-reduce** optimization generally makes code worse on the 68k, so it is not enabled by default at any optimization level. The **-funroll-loops** switch is new, and may not be doing a great job yet. In any case, it makes huge space vs. speed tradeoffs, so you probably don't want to use it except under special circumstances. Compiling with **-O2** may be significantly slower than compiling with **-O**, and may use much more memory. However, it consistently produces slightly better (often slightly smaller) code.

More optimizations have been added in 3.1. See new features above.

Objective C Protocols

Protocols allow you to organize related method into groups that form high-level behaviors. This gives library builders a tool to identify sets of standard protocols, independent of the class hierarchy. Protocols provide language support for the reuse of design (interface), whereas classes support the reuse of code (implementation). Well designed protocols can help users of an application framework when learning or designing new classes. Here is a simple protocol definition for archiving objects:

```
@protocol Archiving
- read:(NXTypedStream *)stream;
- write:(NXTypedStream *)stream;
@end
```

Once defined, protocols can be referenced in a class interface as follows.

```
/*
 * MyClass inherits from Object and conforms to the
 * Archiving protocol.
 */
@interface MyClass : Object <Archiving>
@end
```

Unlike copying methods to/from other class interfaces, any incompatible change made to the protocol will immediately be recognized by the compiler (the next time the class is compiled). Protocols also provide better type-checking without compromising the flexibility of untyped, dynamically bound objects.

```
MyClass *obj1 = [MyClass new];

// legal, obj2 conforms to the Archiving protocol.
id <Archiving> obj2 = obj1;

// illegal, obj1 does not conform to the TargetAction protocol.
id <TargetAction> obj3 = obj1;
```

Protocols can also be used to specify the interface to a remote object. Defining a protocol for a remote object is very similar to defining a protocol for a local object. There are, however, several method qualifiers that allow you to communicate additional information specific to providing a remote interface. For example, the **oneway** qualifier tells the compiler/runtime system not to block (that is, not to wait for the remote method to finish). In the local case, this qualifier is ignored; the method must finish before continuing execution). The complete list of qualifiers that were added for remote objects are: **in, out, inout, bycopy,** and **oneway**. Here are some example declarations:

```
- in:(in int *)a out:(out int *)b inout:(inout int *)c;

- byCopy:(bycopy)a byProxy:theDefault;

- (oneway)asyncronousMessage;
```

These new keywords may only be used in protocol declarations. They may not be used in other method declarations, or in method definitions. They may not be used

at all for functions.

Forward Declaration of Classes

In order to better support the use of protocols and static typing, Objective C now supports declarations of classes using the **@class** directive. For example, suppose you wish to create a class Foo which contains a statically typed instance variable of the class Bar. The header file **Foo.h** might look like this:

```
#import <objc/Object.h>
@class Bar;

@interface Foo : Object
{
    Bar *myBar;
}
- (Bar *) myBar;
@end
```

Note how **Foo.h** declares the class Bar using the **@class** directive, rather than importing the header file **Bar.h**. This is because the the interface of Foo does not depend on any details of the interface of Bar (such as its instance variables or methods). It only depend on the fact that there is a class named Bar, which is what the **@class** directive specifies. Of course, the implementation of Foo may well depend on the interface of Bar, so **Foo.m** may well import **Bar.h**.

The **@class** directive allows statically typed instance variables and method parameters of that class to be declared. It will not allow any operations which require the class's interface, such as sending messages to instances of the class, or referencing public instance variables. In these cases the compiler will issue an error indicating that the interface declaration for the class cannot be found.

By using the **@class** directive in class interfaces rather than importing other classes' interfaces, many problems with circular definitions can be avoided. For example, if the class Bar contains a statically typed instance variable of the class Foo, and both header files import the other, then neither will compile successfully. However, if both header files are written using **@class** as shown above, no problems will occur.

Several classes can be declared using a single **@class** directive by separating the class names with commas:

```
@class Foo, Bar;
```

New Instance Variable Access Control

The Objective C compiler now supports the access control specifiers **@private** and **@protected** in addition to **@public**. Instance variables which are declared **@public** are accessible by all; those which are declared **@protected** are accessible only by the defining class and its subclasses; and those which are declared **@private** are accessible only by the class in which they are contained. If

no access control specifier is used the default is @**protected**, for backward compatibility.

Support for New Header File Organization

As part of the reorganization of system header files, the default search path for included files has been changed. The new search path is:

/NextDeveloper/Headers
/NextDeveloper/Headers/ansi
/NextDeveloper/Headers/bsd
/LocalDeveloper/Headers
/NextDeveloper/2.0CompatImports
/usr/include
/usr/local/include

A symbolic link from **/usr/include** to **/NextDeveloper/Headers** should minimize compatibility problems.

New Preprocessor Directive

To complement the ANSI **#error** preprocessor directive, the GNU compiler now

supports a **#warning** directive. This directive prints a warning message containing the rest of the line, and continues compilation. This directive is useful for warning about use of obsolete header files, for example.

Attributes

The keyword **__attribute__** allows you to specify special attributes of variables or structure fields. The attributes currently defined are the **aligned** and **format** attributes.

The **aligned** attribute specifies the alignment of the variable or structure field. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable **x** on a 16-byte boundary. On a 68000, this could be used in conjunction with an **asm** expression to access the **move16** instruction which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned **int** pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a **aligned** member that forces the union

to be double-word aligned.

It is not possible to specify the alignment of functions; the alignment of functions is determined by the machine's requirements and cannot be changed.

The **format** attribute specifies that a function takes **printf** or **scanf** style arguments which should be type-checked against a format string. For example, the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
    __attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to **my_printf** for consistency with the **printf** style format string argument **my_format**.

The first parameter of the **format** attribute determines how the format string is interpreted, and should be either **printf** or **scanf**. The second parameter specifies the number of the format string argument (starting from 1). The third parameter specifies the number of the first argument which should be checked against the format string. For functions where the arguments are not available to be checked (such as **vprintf**), specify the third parameter as zero. In this case the compiler only checks the format string for consistency.

In the example above, the format string (**my_format**) is the second argument to **my_print** and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The **format** attribute allows you to identify your own functions which take format strings as arguments, so that GNU CC can check the calls to these functions for errors. The compiler always checks formats for the ANSI library functions **printf**, **fprintf**, **sprintf**, **scanf**, **fscanf**, **sscanf**, **vprintf**, **vfprintf** and **vsprintf** whenever such warnings are requested (using **-Wformat**), so there is no need to modify the header file **stdio.h**.

New Compiler Warnings

Version 2 of the GNU compiler has quite a few new warnings which you will no doubt encounter soon. Here is a summary of them with examples taken from **libsys** and **libNeXT**. Most of these warnings are enabled by the **-Wall** compiler switch.

· Warnings about argument mismatch now print the argument number:

```
getpass.c:43: warning: incompatible pointer type for argument 2 of
`signal'
```

· Warnings are generated for incorrect parameters to printf-style functions when the control argument is a string constant. These warnings are enabled by the **-Wformat** compiler switch, which is automatically enabled by the **-Wall** switch. These warnings can be turned off by using the **-Wnoformat** switch after the **-Wall** switch.

· Warnings are generated for empty bodies in if and else statements:

```
View.m:2000: warning: empty body in an if-statement

if ([view getVisibleRect:&visRect]);
    [view _display:&visRect :1];
```

- There are new warnings about potential operator precedence errors:

```
Application.m:2015: warning: suggest parentheses around + or -
inside shift

int curBPS = (NXNumberOfColorComponents (
    NXColorSpaceFromDepth(screens[cnt].depth)) > 1) << 8 +
    NXBPSFromDepth(screens[cnt].depth);

editsound.c:252: warning: suggest parentheses around + or - in
operand of &

s1->dataLocation = (strlen(s1->info)+3 & ~3) - 4 +
    sizeof(SNDSoundStruct);
```

- These warnings may seem fascist, but of the three occurrences of this warning in the Application Kit, for example, it turned up these two bugs:

```
ButtonCell.m:1414: warning: suggest parentheses around && within ||

if (NXDrawingStatus == NX_DRAWING &&
    (LIGHTBYGRAY && !CHANGEGRAY) || (CHANGEGRAY && !LIGHTBYGRAY))
{
```

```
    NXHighlightRect(userRect);
}

appServicesMenu.m:245: warning: suggest parentheses around && within
||

while (*s2)
{
    if (*s2 != '\t' && *s2 != ' ' || *s2 != '\n')
        *s++ = *s2;
    s2++;
}
```

- This can catch subtle bugs, but forces you to be careful with unsigned:

```
Listener.m:1306: warning: ordered comparison between signed and
unsigned

int hsize = msg->header.msg_size;

if (msg->header.msg_type != MSG_TYPE_NORMAL || hsize >
    sizeof(NXMessage))
    return NO;
```

- This is a potentially useful warning, but it has the following problem:

```
Application.m:2048: warning: cast from pointer to integer of
different size
```

```
if ((BOOL)[theWindow perform:aSelector])
    return theWindow;
```

You can work around this problem by casting to `int` and then to `BOOL`.

- I have only seen this warning for code generated by Mig:

```
lookupUser.c:135: warning: value computed is not used
```

- I haven't seen either of these, but they seem good:

```
warning: unreachable code at beginning of switch statement

warning: shift count is negative
```

- This is legal, but rather confusing:

```
symbols.c:62: warning: `sect_object_symbols' initialized and
declared `extern'

extern struct sect_object_symbols sect_object_symbols = { 0 };
```

- Forward declare static functions using ªstaticº rather than ªexternº:

```
typedstream.m:148: warning: static function declaration for
`_NXOpenEncodingStream' follows non-static
```

```
extern _CodingStream *_NXOpenEncodingStream (NXStream *physical);

static _CodingStream *_NXOpenEncodingStream (NXStream *physical) {
```

- Don't do this, it probably won't do what you want:

```
cabs.c:35: warning: structure defined inside parms

double cabs(struct {double x,y;} z)
{
    return(hypot(z.x,z.y));
}
```

New Predefined Macros

The Release 3.0 compiler predefines new macros to aid in writing architecture-independent code.

- Architecture: In addition to the existing predefines which identify specific target architectures (for example, **m68k**, **i386**), the compiler also predefines the macro `__ARCHITECTURE__` to be a string constant identifying the target architecture (ªm68kº, ªi386º). This macro is used by system header files to include the architecture-specific files without having to enumerate all supported architectures.

- Byte ordering: The compiler predefines either `__BIG_ENDIAN__` or `__LITTLE_ENDIAN__`, as appropriate for the target architecture.

Implementation Changes

Name Change of Compiler Proper

The names of the actual Objective C and C++ compilers (as opposed to the compiler driver `cc`) have been changed to conform to the GNU standards. The Objective C compiler is now named `cc1obj`, and the C++ compiler is named `cc1plus`. The compiler uses `cc1obj` when compiling C or Objective C programs, and uses `cc1plus` when compiling C++ or Objective C++ programs. The compiler never uses the old name `cc1`.

Improved Code Generation for Floating-point Conversions

The compiler no longer emits library calls for any standard C operations. Conversions from floating-point values to integral values no longer call the library functions **_fixdfsi()**, **_fixsfsi()**, **_fixunsdfsi()** or **_fixunssfsi()**. Instead, the compiler emits a sequence of inline instructions to perform the operation. Certain programs which make extensive use of floating-point conversions may notice a significant speed up.

Method Descriptors for Protocols

Type descriptors for methods defined within a protocol definition have the following additions.

method qualifier	encoding
in	'n'
inout	'N'
out	'o'
bycopy	'O'
oneway	'V'
const	'r'

Method descriptors for protocols have basic support for C structures. The format for structures and pointers to structures are encoded in the method descriptor. The following appkit method:

```
 - beginPageSetupRect:(const NXRect *)aRect
placement:(const NXPoint *)location;
```

is encoded as the following (changes underlined):

```
@16@8:12r^{_NXRect={_NXPoint=ff}{_NXSize=ff}}16r^{_NXPoint=ff}20
```

Only ªone levelº of the structure is encoded. For example, if the structure contains pointers to other structures, only the struct name is encoded, not the format. This

style of composite object is fully supported by Objective C. For example,

```
- streamType:(NXStream *)stream;
```

is encoded as the following:

```
@16@8:12^{_NXStream=I**iilii^{stream_functions}^v}16
```

Notice how the format of ªstream_functionsº is not expanded.

Improved Objective C Type Checking

Assignments between variables of type **id** and **Class** no longer generate spurious warnings, so the **Class** type can now be used more effectively.