3.3, 3.2hp, PDO 2.0 Release Notes: C Compiler

Notes Specific to Release 3.3/3.2hp

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

This is also the compiler for NeXTSTEP/hppa 3.2. Please contack Kresten_Thorup@NeXT.COM if you have eny problems with it, and the problems are not mentioned in this note.

Recompilation

The following changes in this version of the GNU C compiler requires you to recompile some existing code.

· **C++ name mangling.** The method of ``mangling'' C++ function names has been changed.   So you must recompile all C++ programs completely when you start using NeXTSTEP 3.3.   libg++ is updated to use the new mangling.

- **Struct arguments on hppa.** A bug fix in passing of structure arguments for the hppa architecture makes code compiled with the GNU C compiler incompatible with code compiled with earlier versions (if it passes struct arguments of 33 to 64 bits, interspersed with other types of arguments). This is a very rare problem, since the calling conventions are really changed only when certain conditions are met, so most code will work withou recompilation.

Features in test bed

The following are features which have just been introduced, and are thus not enabled as default:

- **Dylib codegen; -k.** (m68k and i386) Generate code for the new dynamic shared library scheme. Eventually, this flag will be named `-NEXTSTEP-deployment- target 3.3'.

New Features

The following new features have been added to the GNU C Compiler for Release 3.3. Using some of them may disable you to run the resulting application on pre-3.3 NeXTSTEP. The details of this should be described elsewhere...

- **IEEE Compliant Floating Point.** Parts of the backend for the m68k and i386 compilers have been rewritten in order to make sure that the generated code is fully IEEE compliant. Some programmers may have been using -ffloat-store to obtain this, but that option is no longer needed. *To enable this feature, use the -ffppc flag to the compiler in stead of -ffloat-store. -fppc will eventaully be enabled as default, so better test it now before that happens.*

This feature is incompatible with pre-3.3 NeXTSTEP threads, and should, once it becomes the default behavior, be turned off using -fno-fppc (and replaced by -ffloat-store) when building binaries using threads to be run on any pre-3.3 systems.

- **C++ System Header Files.** A new pragma \``#pragma cplusplus`' 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 embeded in an implicit \``extern "C++"`' block. Alternatively, if either of `g++` or `c++` appears in the full path name to a headerfile (ignoring case), it is also considered C++.

  An error is reported if this pragma appears inside an explicit \``extern "C" {...}`'.

- **Long double on i386.** The GNU C compiler now supports `long double' meaningfully on the i386 (96-bit floating point).

- **Nested functions.** Pascal-style nested functions are now supported in C. This have been in the GNU C compiler for a while, but is now also supported for the NeXTSTEP Operating System. Using this will disable you to run the resulting binary on pre-3.3

systems.

- **Array and structure initializers.** The C syntax for specifying which structure field comes next in an initializer is now \``.`*field_name*`=`'. The corresponding syntax for array initializers is now \``[`*index*`]=`'. For example,

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

  This was changed to accord with the syntax proposed by the Numerical C Extensions Group (NCEG). C++ does not support this kind of initializers.

- **Accessing instance variables in class methods.** It used to be common programming style in Objective-C to assign self, and then access instance variables as in the context of an instance method. To discurage this anachroistic use, a warning will be issued.

- **C++ \`smart pointers' and Objective-C.** The Objective-C++ compiler has been changed to recognize type conversion operators for the receiver in a message expression. This allows you to implement so called \`smart pointers' to Objective-C objects. However, this conversion is a bit subtle for various implementation reasons. You should avoid having multiple type conversion operators (from the same class) to different pointer types. This may confuse the machinery to choose the wrong type. If however, you need more than, you must make an \``operator id`' which will then be chosen over any Objective-C class pointer types. For example:

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

class ptrFoo {
   Foo* value;
  public:
   operator Foo*();
};

foo (ptrFoo xx) {
   int i = [xx doSomething]; // calls operator Foo*
 }
```

Here, the he compiler will recognize `xx` as statically typed to `` `Foo*' `` in the message expression. However, if `` `class ptrFoo' `` had implemented other `` `operator X*' `` functions, you would have to implement an `` `operator id' `` since otherwise the compiler would not know which conversion to look for. Actually, the compiler prefers `` `operator id' `` at all time, if it can find it.

- **Unaligned text.** (i386 only) Using the new flag `-munaligned-text` all alignment for instructions will be turned off. Ocasionally this may be interesting if the code size is significant.

Changes

- **Implicit cast from int to enums**. The compiler used to implicitly allow casts from int to any enum type. According to ANSI C++ and ANSI C, this is not correct behavior, so a warning will be issued when this is needed. It is a problem because you cannot be sure the integer value lies in the range of the enum.

- **Assignment used as conditional.** With the -Wall flag turned on, the compiler will issue a warning for assignments used as conditionals in if, for, and while statements. For example the code:

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

  Will generate a warning suggesting an extra set of parenthesis around the assignment, like:

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

  The idea is that this warning will catch situations where you really meant test for equivalence == and not assignment =.

  Note: There has been much discussion and many comments on this warning. Many developers from NeXT have mailed me or come intoo my office complaining about it, or asking if it was going to stay. I have decided that it will stay in the compiler anyway, because several people have come back to me and changed their mind. The first time it actually catches a real bug, you will be glad to have it.

- **Pointers to members.** The C++ compiler used to allow the case in the following code which is now marked as an error. The compiler will issue and error message if those occur.

```
class X { public: void f(int); };
class Y : public X { public: void f(int); };

g (Y* y, X* x)
{
     void (X::*xmp)(int) = &X::f;
     void (Y::*ymp)(int) = &Y::f;

     (y->*xmp)(4); // ok
     (x->*ymp)(5); // error (1)
     ymp = &Y::f;  // ok
     xmp = &Y::f;  // error (2)
}
```

Statement (1) is an error because, intuitively you cannot be sure that an X* ``responds to'' a Y member pointer. The assignment in (2) is an error, because in general you cannot be sure that some member of a derived class (in this case Y::f) is available in any of it's base classes. Even though ymp is initialized to an X member pointer, it cannot safely be applied to an X*. *This was reported as a problem by Mike Monegan.*

- **Implicit cast from void* to object pointer.** The C++ compiler used to implicitly allow casts from void* to any C++ Object pointer type. According to ANSI C++, this is not correct behavior, so a warning will be issued when this is needed. *This was reported as a problem by Mike Monegan.*

- **Incrementing enums.** The C++ compiler used to allow increment and decrement operations on enums. According to

ANSI C++, this is not correct behavior, so a warning will be issued for this. It is a problem because you cannot be sure the resulting integer value lies in the range of the enum. *This was reported as a problem by Mike Monegan.*

- **`volatile` and `const` declarations must match definitions.** The C compiler used to allow a declaration and it's definition to mis-match on volatile and const-ness. This is no longer allowed, and the compiler will issue an error when such appears.

- **-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. On the 3.2hp release, both forms are accepted.

- **-fkeep-inline-functions.** Preious versions of the compiler would eliminate unused static inline functions. Using this flag, you can make sure they get compiled into the image.

- More **Strict Objective-C Syntax Checking.** The compiler now is more strict with syntax checking, so you are not allowed to nest `@interface` and `@implementation` blocks. This is done to be able to check that the user allways remembers closing `@end`.

- **C++ multiple virtual inheritance.** Improvements have been made to the C++ compiler so that the dispatch of virtual functions is right most of the times except for the following case.

```
class A                          { void f(); }
class B : public virtual A   { virtual void f(); }
class C : public virtual A   { virtual void f(); }
class D : public B, public C { virtual void f(); }
class E : public D           { virtual void f(); }
```

Consider the following calls,

```
 foo() {
   E* e = new E;
   B* b = e;
   C* c = e;
   D* d = e;

   c->f();  // Wrong - calls D::f()
   b->f();  // Right - calls E::f()
   d->f();  // Right - calls E::f()
}
```

However, if the above hierarchy is modified such that the non-virtual function `f` in A is made virtual, the dispatch works correctly. The workaround is therefore to make `f` virtual through out the hierarchy.

```
class A                      { virtual void f(); }
class B : public virtual A   { virtual void f(); }
class C : public virtual A   { virtual void f(); }
class D : public B, public C { virtual void f(); }
class E : public D           { virtual void f(); }
```

In any event, the compiler now warns if there is a posibility that wrong code is being generated:

```
warning: method `A::f()' redeclared as `virtual B::f()'
```

This warning is issued whenever a non-virtual method is redeclared in a subclass as virtual.

Known Bugs and Limitations

The following bugs or limitations are worth noting for the GNU C and C++ Compilers for Release 3.3, since they were not there for earlier releases.

· **Constant pointers (40775).** The C++ compiler does not handle const pointers, i.e. `` `*const' `` correctly. Occationally, this may cause an internal compiler error. Since this is only an optimization issue, this can be replaced by simply `` `*' ``.

· **Conditional expressions (39034).** The C++ compiler does not handle conditional expressions yielding an object or struct value correctly. Occationally, this may cause an internal compiler error. To fix it, just change the code to not use conditional expressions.

· **Constant expressions (no bug number).** Some expression, such as the difference between the address of any two symbols are not properly recognized by the compiler as valid initializer values. There is currently no workaround for that. This is actually not a bug in ANSI-C sense (just a limitation), because the ANSI-C standard does not specify any behavior for this.

- **Complex numbers (40546).** The GNU compiler is documented as supporting complex numbers. This part of the compiler is untested, and have been found to be errerous in some cases. It is not recommended that this is being used.

- **Long doubles (41950).** Using immediate long doubles sometimes doesn't work. Using the form `23450.0L` which is used to write long doubles immediate values may cause the compiler to report an internal error.