C++ on a NeXT System

The purpose of this guide is to help developers who already have a large investment in C++ code easily integrate it into the NeXT development environment. This guide isn't intended to be a C++ tutorial, since there are many C++ books that cover this subject in depth.

The C++ and Objective C languages have their respective strengths and weaknesses, depending on the type of application you're building. For the types of applications we build at NeXT (libraries and tools that are intended to be reused in a wide variety of situations), Objective C works great since it supports run-time binding and dynamic loading of objects, which is a key feature in order to take full advantage of the power of Interface Builder. We also find developers have a much easier time learning Objective C, since it only adds a few new constructs to C.

We would recommend that you use the Objective C language if you are developing your application from scratch. See the section ªPorting Code From Other Platforms: More Informationº for details.

Objective C and C++ Design Goals

When NeXT added C++ to its development environment, it had these goals in mind:

- To allow both Objective C and C++ to coexist in the same compiler.
- Not to alter the semantics of either language.

Benefits: Developers will be able to compile their existing C++ applications and conveniently use the NeXTstep™ development tools in conjunction with the C++ programming language.

C++ Utilities Available in NeXTstep Release 2

The following utilities are available in NeXTStep Release 2:

cc++ A C++ compiler based on the GNU G++ compiler that has been extended to recognize Objective C. The compiler is based on *G++ version 1.36.4*, which implements *version 2.0 of the C++ language*, as specified by AT&T. This is a native compiler, not a translator, which means:

- Turnaround time is better.
- Debugging is more convenient.
- Optimizations are possible.

Note that **cc++** accepts all the compiler flags that are specified for **cc**. See the UNIX® manual page on **cc** for more details.

nm++ An enhanced symbol table tool. It also displays type information for function prototypes. **nm++** accepts all the options that are specified for **nm.** See the UNIX® manual page on **nm** for more details.

gprof++ A profiler for C++ code. Many non-NeXT systems don't support this.

See the UNIX manual page on **gprof** for more details.

gdb A terminal-based debugger with integrated support for both C++ and Objective C. For documentation, see the *NeXT Developer's Library* (accessible through the NeXT Developer target in Digital Librarian™).

Porting Code From Other Platforms: More Information

Here are three key points about porting C++ applications:

- Interface Builder™ doesn't contain any support for C++. See the next section on how to integrate C++, Objective C, and Interface Builder files in one application.
- The Application Kit™ will continue to be based on Objective C.
- NeXT doesn't ship any C++ libraries with NeXTstep Release 2. However, libraries are available from other sources.

The Free Software Foundation (which is responsible for the GNU project) has written a large library of routines for C++ called **libg++**, but under current Free Software Foundation rules **libg++** can only be used by programs that won't be distributed commercially. For further information about how to obtain **libg++**, contact the Free Software Foundation at:

Free Software Foundation, Inc.
675 Massachusetts Avenue
Cambridge, MA 02139
U.S.A.

Telephone: (617) 876-3296
Electronic mail: **gnu@prep.ai.mit.edu**

The GNU C++ (**libg++**) class library includes support of the following (frequently requested) features:

- Streams and other I/O
- Complex numbers
- String support

Other third parties offer C++ libraries as well. AT&T offers a library that's a subset of **libg++** for a fee. Some developers have chosen to develop their own C++ libraries.

How to Integrate Interface Builder, Objective C, and C++

To integrate Interface Builder, Objective C, and C++ files in one single application, you must do three things: compile appropriately, modify your Interface Builder files, and get the two languages to talk to each other in your code.

Compiling C++ Files

You must use the C++ compiler for source files that contain either C++ alone or both C++ and Objective C. If you want to use the C++ compiler for *all* of your source files, including the Objective C sources, add the following line to your **Makefile.preamble**:

```
CC=cc++
```

If you want to use the C++ compiler for C++ files only, add the following to your **Makefile.postamble** rather than overriding the CC directive in the **Makefile.preamble**:

```
.c.o:
    cc++ $(CFLAGS) -c $*.c -o $(OFILE_DIR)/$*.o
```

When running **make**, all your ".c" files will then be compiled with **cc++**. Note that you'll need to add the compiler flags "-g -DDEBUG" to the above line as follows, in order to debug your C++ files with gdb:

```
.c.o:
    cc++ $(CFLAGS) -c -g -DDEBUG $*.c -o $(OFILE_DIR)/$*.o
```

You'll also need to modify those ".c" files as described below.

The C++ ªlinkageº directive serves two purposes (when importing interface files that contain straight ANSI-C/Objective C code). The directive:

- Allows you to link with libraries that haven't been compiled with the C++ compiler. Since libraries on a NeXT computer are compiled with the Objective C compiler (**cc**, not **cc++**), you must use the C++ linkage directive when

importing interface files that represent NeXT libraries (or any library that isn't compiled with **cc++**).

- Tells the compiler to ignore C++ keywords that will result in syntax errors when importing ANSI-C or Objective C interface files. The linkage directive essentially tells the C++ compiler to treat keywords (such as the method names ªnewº, ªdeleteº, etc.) as normal identifiers.

Now that you're using the C++ compiler, you have to notify the compiler when your header files contain non-C++ code. For Objective C header files embedded in a C++ source file, encapsulate your **#import** directives like this:

```
extern "Objective C"
{
#import <appkit/Application.h>
#import <appkit/Panel.h>
#import <appkit/TextField.h>
#import <appkit/Button.h>
}
```

For regular C header files embedded in a C++ source file, encapsulate your **#import** directives like this:

```
extern "C"
{
#import <appkit/publicWraps.h>
#import <objc/error.h>
#import <objc/NXStringTable.h>
#import <strings.h>
}
```

Interface Builder

Within Interface Builder you need to add the C++ ª.cº and ª.hº files to your project. Add the files separately in the Project InspectorÐª.cº files go in the ª.cº category, and ª.hº files go in the ª.h (other)º category.

If you already have a ª_main.mº file, make sure that the option in Interface Builder for generating the main file is turned *off*. Then, remove the void declaration of the main procedure by replacing:

```
void main(int argc, char *argv[]) {
```

with:

```
main(int argc, char *argv[]) {
```

Note: You only need to change the ª_main.mº file if you use the C++ compiler for *all* your source files.

Modifying Source Code

Since the nib files generated by Interface Builder are based on the Application Kit, and since Interface Builder generates source templates in Objective C, you need to envision your application such that Objective C and nib files are the foundation of the program, and the C++ code is a supporting library.

Now that you can compile, you need to get an Objective C object and a C++ object to pass messages to one another. Basically, in either case you use the language constructs of the object to which you are referring, and embed them in the source file of the other language.

As an example, suppose that you've created two objectsÐa C++ object and an

Objective C object. This is how you might refer to the C++ object from Objective C and tell it to ªdo somethingº:

```
implementation SomeObjCClass:Object

class CalcEngine *cplus_object;

- init {
    cplus_object = new CalcEngine;
    cplus_object->doSomething();
    return self;
 }
```

C++ objects are implemented as regular C structures, so to access public instance variables, or public methods of a C++ object, you dereference the object with the -> syntax as you would a structure member. This is how you might refer to an Objective C object from C++:

```
void SomeCPlusClass::functionName()  {
    id objectiveObj;
    objectiveObj = [[ObjectiveObjClass alloc] init];
    [objectiveObj doSomethingElse:what];
}
```

Using pswrap

Since **pswrap** generates C code, the C++ compiler won't work with ª.cº files generated by **pswrap**. To integrate ª.pswº files with C++ files, you need to create a **Makefile.postamble** and tell **make** to use **cc** when compiling **pswrap**'ed files. For example:

```
.psw.o:
    $(PSWRAP) $(PSWFLAGS) -a -h $*.h -o $*.c $*.psw
    cc $(CFLAGS) -c $*.c -o $(OFILE_DIR)/$*.o
    rm $*.c
```

You have to remove the ".c" files generated by **pswrap** after the compilation, so that in subsequent makes, the C++ compiler will not try to compile these files. In general, you can override the default **make** rules to suit your environment. See also **/usr/lib/nib/Makefile.common** for NeXT's default rules.

Programming Example

There's an example located in **/NextDeveloper/Examples/CalculatorLab++** that illustrates the integration of Interface Builder nib files, Objective C source code, and C++ source code into one program.

How to Compile libg++ on a NeXT System

Here's an example of how to compile version 1.37.0 of **libg++** on a NeXT machine. Note that we haven't tested the compilation of other versions of **libg++**, but the steps should be similar.

Modifying the Makefile in libg++ and tests Directories

- First, you must turn off the inclusion of **libg++**'s **malloc** routines. You can do this by setting a compiler flag. Edit the Makefile and search for MALLOC. Remove the comment sign (#) on the line that sets XTRAFLAGS to -DNO_LIBGXX_MALLOC. It will look like this when you're finished:

```
# Use this to disable placing libg++ version of malloc in libg++.a
XTRAFLAGS = -DNO_LIBGXX_MALLOC
```

Make the same change to the **Makefile** in the **tests** directory.

- Also, if you aren't using gnu **make**, you should change this line in the Makefile:

```
#PWD := $(shell pwd)
```

to point to the directory in which the sources live, for example:

```
PWD=/me/C++/Libg++-1.37.0
```

- Set GXX to **cc++** and set VERBOSITY_FLAGS to **-Wall**, removing the **-v** flag. (Changing verbosity isn't necessary, but with **-v** set you will get lots of useless messages.)

```
GXX = cc++
VERBOSITY_FLAGS = -Wall
```

Modifying Time.h in the libg++ Directory

- There are some conflicts between the **time.h** file that's shipped with **libg++** and the **time.h** that comes with a NeXT system. Change the first few lines of **time.h** in the **g++-include** directory to look like this:

```
#ifndef time_h
#define time_h 1
#ifdef NeXT
#define _TIME_H 1
#endif
```

After you've done all these steps, version 1.37.0 of the GNU **libg++** will compile on a NeXT machine.

Compiling Your Source Files

- Remember to specify the GNU libraries in your Makefile. For example:

```
LIBS = -L/me/C++/Libg++-1.37.0/src -lg++
```

```
CFLAGS = -I/me/C++/Libg++-1.37.0/g++-include -g -Wimplicit
```

Some Known Gotchas

This section contains miscellaneous question/answer pairs (formerly in NeXTanswers™) that describe workarounds to some problems with the C++ compiler in NeXTstep Release 2. (We expect that these problems will be fixed in the next major release of our software.) This section also includes other peculiarities you should be aware of when integrating C++. This material will be updated periodically as we collect more information on the usage of C++ on a NeXT system.

C++ Initialization of Static Class Members

You shouldn't call Application Kit methods during the static initialization of your C++ class members, since this code is executed even before the Application Kit calls **[Application new]** to create an Application object, NXApp. This could create

unexpected problems, such as crashes or null pointers, because of a non-existent Application object.

C++ enum Declaration

Q: When I try to compile a C++ program that contains this header file:

```
typedef enum {
    aaa,
    bbb,
    ccc
    } Type;

class myClass {
    private:
        Type myType;
    public:
        void setType(Type newType);
    };
```

I get the link error:

```
/bin/ld:Undefined symbols:_setType__7myClass3$_1
```

and the compilation stops. The same program would compile fine on a non-NeXT C++ compiler. What's the problem?

A: There's indeed a bug in the current C++ compiler, but the workaround is easy.

The problem is with the declaration:

```
typedef enum {
    aaa,
    bbb,
    ccc
    } Type;
```

This causes problems with C++'s name mangling, since this enum has no tag. The current compiler doesn't know how to encode this type, so it simply uses an integer for the number of the unencodable type. There's no reason to think that this will match the encoding used in some other file that may have had a different number of unencodable types or the same types but in a different order. This is indeed what was happening in your case, and explains why the link failed.

You should just use this declaration instead:

```
enum Type {
   aaa,
   bbb,
   ccc
   };
```

In C++, a typedef is automatically published for each struct, union, or enum with the same name as the tag, so you can continue to use **Type** rather than **enum Type**.

This bug will be fixed in a future release of the C++ compiler.

C++ gdb Bug Workaround

Q: When I tried to trace my C++ program with **gdb**, **gdb** just crashes with a Bus Error message. Why is this happening?

A: There's a problem with using **gdb** and C++, when you forward declare a class, then try to use the class as a static member of another class. The following code snippet shows the problem, and how to work around it:

```
class A;

class B {
    static   A *a;
    public:
             B(){};
    };


main()
{
    int i;
    i=2;
}

class A{
    int x;
    };
```

Note that class B has a static class variable (or in the C++ jargon, a static class

member), but no instance variable (or in the C++ jargon, no member). The workaround is to add a dummy instance variable to the class definition:

```
class A;

class B {
    static   A *a;
    A *dummy;
    public:
             B(){};
    };


main()
{
    int i;
    i=2;
}

class A {
    int x;
};
```

C++ Compiler Crash Inline Functions

Q: When I try to compile some of my C++ files using the command **cc++ -g**, the compiler just crashes, with a fatal signal 10. What can I do?

A: This problem can be caused by your inline functions not having an argument name in the function definition. The workaround is to name the argument. For example, the following code snippet shows the problem and how to correct it:

```
class Foo {
    public:
    inline void* operator new(size_t)
    {
    return FooBucket::free_list->allocate();
    };
```

The function **new** doesn't have an argument name. The fix is simply to add an argument name:

```
class Foo {
    public:
    inline void* operator new(size_t fooMember)
```

```
{
return FooBucket::free_list->allocate();
};
```

C++ new Error Handler

Q: In C++, how can I override the error handler for the **new** operator? I can do this with other compilers, but it doesn't work with the NeXT C++ compiler.

A: On a NeXT system, you can override the error for the **new** operator in the following way. Define a routine, say, **newErrorHandler()**:

```
static void newErrorHandler()
{
    fprintf(stderr, "new: Failed to acquire sufficient memory.");
    fflush(stderr);
}
```

Then, somewhere in your initialization routines call **set_new_handler()**:

```
set_new_handler(newErrorHandler);
```

newErrorHandler() will be called whenever an error is encountered during the **new** operation.

C++ Bug fatal signal 11 collect

Q: I'm getting the following error message from the C++ compiler:

```
cc++: Internal compiler error. Program collect got fatal signal 11.
```

What's going on?

A: You have encountered a known bug in the C++ compiler. The bug will be fixed in an upcoming release. It's a bug in the program **collect** that relates to the use of **alloca()**. Essentially, **collect** runs out of stack when ªcollectingº your application, causing it to crash.

There's a workaround. Just increase the stacksize in the shell. Use the **limit** command to increase your stacksize, like this:

```
% limit stacksize 10000
```

In NeXTstep Release 2, this workaround doesn't work when used with the **make** utility. **make** hardwires the stacksize rather than reading it from the users default. So, in Release 2 you must link by hand, or write a little shell script to do it for you.

You can get a fixed version of **collect** by sending e-mail to **ask_next@next.com.**

Destroying Stack-Allocated Arrays

Q: The following code snippet doesn't execute properly, and the compiler generates a core dump:

```
class X {
   public:
        X() { printf("Initializing %x\n", this); };
        ~X() { printf("Destroying %x\n", this); };
};

main()
{
 X foo[2];
```

```
 return 0;
}
```

The program output would show:

```
Initializing 3fffa7a
Initializing 3fffa7b
Destroying ffffffff
```

Note that the destructor should be called twice, and it should be destroying the objects with the same addresses as the ones displayed with initializing.

A: You have encountered a known bug in the C++ compiler. When allocating arrays of class objects on the stack, these arrays do not get destroyed properly at the exit of your program. The bug will be fixed in our next major software release. To work around this problem for now, change the code as follows:

```
class X {
  public:
        X() { printf("Initializing %x\n", this); };
        ~X() { printf("Destroying %x\n", this); };
};

main()
```

```
{
   {
   X foo[2];
   }
 return 0;
}
```

Note that there is an extra level of braces around the array, so that the array is not destructed at the same level as that of the return statement.

Written by Mai Nguyen and Mary Mc Nabb. Thanks to Steve Naroff, Adrienne Wong, and Doug Keislar.