XPC Basic Libraries Manual

1.1.3

Generated by Doxygen 1.8.17

Chapter 1

XPC Suite of Libraries and Sample Applications

Author

Chris Ahlstrom

Date

2011-12-17 to 2013-01-12

1.1 Introduction

The **XPC** library suite is a collection of "cross-programming" **C** libraries.

The "XPC suite" is a set of very basic libraries (and some simple applications) intended to be useful and instructive.

The libraries provide functionality to help in error-logging, debugging, threading, numerics, internationalization, and unit-testing.

In addition, the code and other project files attempt to be complete examples of how to document, write, format, build, and unit-test a project.

The current document might not include the API documentation. It may just give an overview of the project, and tutorials about using various open-source and Free software tools to perform maintenance tasks.

1.2 A Note About XPC Documentation

The documentation for the **XPC** suite is generated using the very nice *Doxygen* package. This is a *Free software* package available at

http://www.stack.nl/~dimitri/doxygen/

This package works by allowing the user to mark up source code so that formal and detailed documentation can be regenerated from comments in the code.

For our purpose, we also provided stand-alone documents that end in the file-extension `.dox`. Each such document provides a named page of documentation. This documentation is collected by running *doxygen* from the main documentation directory in `xpc_suite/doc/dox`.

In order to build good `PDF` files from *doxygen*, we believe that the following packages are necessary. This list may err on the side of being too long. The names may differ depending on what distribution one uses.

- `doxygen-latex`
- `texlive`, and the following related packages
  - `texlive-base`
  - `texlive-binaries`
  - `texlive-common`
  - `texlive-extra-utils`
  - `texlive-font-utils`
  - `texlive-generic-recommended`
  - `texlive-latex-base`
  - `texlive-latex-recommended`
  - `texlive-latex3`
  - `texlive-luatex` (maybe)
  - `texlive-metapost` (maybe)
- `psutils`

Note that you do *not* need a `texlive-full` package. It is *huge!*

Note

One issue is ordering the documentation pages in a more logical fashion; currently, they appear in the `PDF` in whatever order they are encountered in the directories. To get around this, every document is specified, in the desired order, in the `doxytop.cfg` file.

1.3 XPC Libraries in Summary

The following sections quickly describe each of the sub-projects of the **XPC** suite. Here are the libraries, and their current status:

Name	Library	Status
XPC CUT Library	xpccut-1.1	Complete, self-unit-tested, very usable
XPC C Library	xpc-1.1	Complete, unit-tested, usable

1.3.1 XPC CUT Library Summary

The **XPC CUT** library (libxpccut) is a **C** unit-test library.

The **CUT** library provides a simple framework for unit-testing using only the **C** programming language. This library provides a fairly easy way to write unit-tests that are:

- As limited or as thorough as you like.
- Self-documenting, and capable of creating a log of the test.
- Able to be used as regression tests.
- Easy to debug.

The **CUT** library is not meant to take the place of complex test suites such as *CppUnit*. It just provides an easy way to write unit-tests and group them together. It also provides an easy way to break each unit-test into a number of simpler sub-tests, and to find out which sub-test fails. It provides for timing the tests, controlling the amount of output to the screen, interactive versus non-interactive usage, and more.

The developer writes a set of test-cases that follow a simple function signature. The developer then instantiates a unit-test object and loads each test case into it. The run of the tests is then kicked off.

The fundamental result of the unit-test application is a single status value: 0 (all tests and sub-tests passed) or 1 (at least one test or sub-test failed.)

The **XPC CUT C** library has no dependencies on any other **XPC** libraries.

For a full description of the **CUT** library, see xpccut_introduction_page.

1.3.2 XPC C Library Summary

The **XPC C** library (libxpc) is a **C** library that provides functionality useful in most projects.

- CPU and operating system identification
- Error and message logging
- GNU gettext support
- Numeric support
- Some minor command-line options handling
- Basic INI-file parsing
- Support for using pthreads.
- Other portability functions.

For a full description of the **XPC C** library, see Introduction to the C Unit-Test Library.

1.4 License Terms for the XPC Suite

See Licenses, XPC Library Suite for a description of all of the **XPC** suite licensing.

1.5 References

1. http://xxxxxxx.yyy

1.6 Incomplete Section

Since you're reading this section, you undoubtedly clicked a link to a document or section that is not yet written. My apologies.

Chapter 2

XPC C Utility Library

This file describes the **XPC C** utility library, which is considered the basic library in the **XPC** suite.

2.1 Introduction to the C Unit-Test Library

The **XPC** library is a "cross-programming C" utility library that provides some very basic functionality that just about any application might need. This library is the main (and most basic) library in the **XPC** library suite.

The **XPC C** library (libxpc) is a **C** library that provides functionality useful in most projects.

- CPU and operating system identification
- Error and message logging
- GNU gettext support
- Numeric support
- Some minor command-line options handling
- Basic INI-file parsing
- Support for using pthreads.
- Other portability functions.

MORE TO COME

Although the utility **XPC** library is written and somewhat tested, it is a work in progress and we have not yet had time to describe it at a higher level.

This file describes the **XPC++** utility library, which is considered the basic C++ library in the **XPC++** suite.

2.2 Introduction to the XPC C++ Basic Library

The **XPC++** library is a "cross-programming C++" utility library that provides some very basic functionality that just about any application might need. This library is the main (and most basic) library in the **XPC++** library suite.

Although the utility **XPC++** library is written and somewhat tested, it is a work in progress and we have not yet had time to describe it at a higher level.

Chapter 3

XPC "Hello" Application

This file describes the building of the **XPC** *Hello* application.

This document provides a discussion of the **XPC** Hello project and how it is set up to support the construction of an automake-based application, linked to one of the **XPC** external libraries.

Table of Contents\n

This document is a work in progress.

Note that you can build the detailed-design documentation by running *doxygen* against the `doxygen.cfg` in directory `xpchello-1.0/doc/dox`.

3.1 Introduction

The **XPC** *Hello* application provides a minimalist demonstration of internationalizing an application.

It provides a good example of some of the main steps needed to create an internationalized application.

In addition, a test application is provided to test linkage with other **XPC** libraries. See the next section.

Finally, this project also includes demonstration application for:

- Internationalization
- Usage of bad pointers
- Core dumps
- The setjump() function

This project provides a small application to demonstrate the many facets (pun intended) of internationalizing an application within the context of GNU automake. The application doesn't do much, and we provide only a bad Internet-lookup translation in Spanish.

As an added 'hello', to show how to link to other XPC libraries, this project also provides a test application to verify that it can be linked to the XPC C unit-test library when that library is installed in the conventional GNU manner. The C unit-test library package is called "xpccut-1.0". The 'hello' test program also illustrates installing and linking to a library that is supported by the GNU "pkgconfig" system. It doesn't really do much in the way of testing, though.

For detailed information on the XPC 'hello' project, see the Doxygen documentation generated from the files in the doc/dox sub-directory. To generate this documentation, first install the "doxygen" and "graphviz" projects. (Also install "texlive" if you want to generate the PDF form of the documentation.) Then issue the following commands:

$ cd doc/dox $ doxygen doxygen.cfg

You can then browse the project documentation by load the link at

doc/dox/html/index.html

Windows:

Sorry, at this time, this project does not support Windows, mostly due to gettext. I don't have a lot of motivation to get gettext() working under Windows. If you get it to work, please send me the files or changes you needed.

This project might build under Cygwin, but I'm not sure if all of the components needed (e.g. graphviz and autopoint) are available with Cygwin/X. Again, I will be happy to receive any fixes.

3.2 The XPC Library Suite

There is a second application included in this project. It is a small *Test* program.

The **XPC** *Test* application provides a separate demonstration (and documentation) of how to link to the unit-test library of the **XPC Library Suite**.

The **XPC Library Suite** is a handful of **C** libraries to provide code that I've found useful over the years, all unit-tested as much as we can stomach.

The *Test* application project has much the same structure as the library suite, but it is not part of the suite, even though we keep it in the same directory tree. The *Test* application requires that the **XPC Unit-Test Library** be built and installed separately.

The **XPC Unit-Test Library** includes a lot more than just unit-testing. Be sure to download it from where you got the *Hello* project, and check it out.

Chapter 4

XPC Hello Internationalization How-To

This document provides a discussion of the **XPC** *Hello* project and how it is set up to support the internationalization of an automake-based application.

4.1 Introduction to 'gettext' Internationalization

The **GNU** *gettext* system provides a good way to internationalize an application. However, it is a complex system, and it is difficult to parse tutorial documentation on using it in an automake-based project.

The *Hello* application provides a minimalist demonstration of internationalizing an application. It is even simpler than the **GNU** Hello World"project:

`http://www.gnu.org/software/hello/`

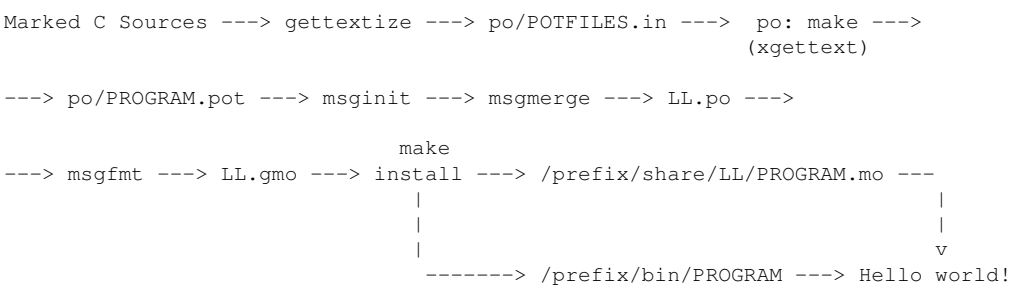
That project is a good example of a self-documenting **GNU** project, but it includes a lot of stuff we don't care about (yet): usage of *Gnulib*; the `getopt_long()` function; *help2man*; *Texinfo*; and **GNU** coding standards. Also, it doesn't really explain everything needed to *create* such a project.

Therefore, we've worked out the procedure in our own *Hello* project, and documented it. In addition, the `bootstrap-po` script, using the `--intl-boot` option, can reconstruct the necessary files from scratch.

4.2 The 'gettext' Process

The following is a modified version of the diagram found at

`http://www.gnu.org/software/gettext/manual/gettext.html#configure_002eac`



where *LL* represents the language code (e.g. "es_ES"), and *PROGRAM* represents the name of the program.

The following usage steps walk the reader through this diagram. It assumes that the basic project files (e.g. configure.ac, Makefile.am, etc.) have already been created so that the project builds properly without internationaliztion.

Usage:\n\n

1. **Marked C Sources**. You must manually mark the translatable message strings in all applicable `C` modules by enclosing them in the `_()` macro. This macro is defined to call `gettext()` to do the actual translation. Marking the sources is ongoing, as new message strings are added to the application. Creating good translatable strings is an art, and GNU documents good ways to do it.

2. **gettextize –force**. Run the *gettextize* program. It adds a `Makefile.in.in` and some helper files to the `po` directory, and some *autoconf* macros to the `m4` directory. It adds a few other files, and modifies some others, making backups. Note that we leave off the `-intl` option. This would cause an `intl/` directory to be created and filled with gettext sources that can be built into the application, so that the end-user doesn't have to have gettext installed on his/her system.

3. **po/POTFILES.in**. Add the list of the translatable **C** and **C++** modules to the `po/POTFILES.in` file. The `build_POTFILES` function in the `bootstrap-po` script does this when the `-boot` option is specified.

4. **po: make *(xgettext)***. Calling *make package.pot* (where `package` is the value of the project's `PACKAGE` macro) in the the `po` directory will call *xgettext*. This application collects all of the marked strings in the sources specified in `POTFILES.in`, and generates a template translation file.

5. **po/PROGRAM.pot**. This is the template translation file. The *make* in the `po` directory causes an error, since none of the languages specified in `LINGUAS` have been handled yet. Therefore, the `bootstrap-po` script punts by using `xpc.pot` from the `contrib/po` directory to create the initial `es.po` file.

6. Other files that need editing are not shown in the diagram:

   - `LINGUAS`. Add the set of supported languages in one line, such as "es de fr". The extra message catalogs "en\@quot" and "en\@boldquot" should also be added. If these are added, then the following files are needed (gettextize will add them):
     - `Rules-quot`
     - `quot.sed`
     - `boldquot.sed`
     - `en@quot.header`
     - `en@boldquot.header`
     - `insert-header.sin`
   - `Makevars`. This file needs to be modified only if your package has multiple `po` directories. The XPC suite does not.
   - `config.guess` and `config.sub`. These files are needed only if `intl/` support is used, and they go into the top-level XPC directory. But the XPC suite does not use that support.
   - `mkinstalldirs`. Apparently not needed anymore, but we will keep it around.

7. **msginit –input=../contrib/po/xpc.pot –output-file=es.po**. The user (or `bootstrap-po` script) calls this program to convert the `xpc.pot` file into a translation file for a given language.

8. **msgmerge**. This program is called as the program is maintained. It merges new messages into the existing language file(s).

9. **LL.po**. This is the translation file for language *LL*. The initial make will fail, since none of the languages specifief in `LINGUAS` have been handled yet. Therefore, the `bootstrap` script punts by copying `es.po` from the `contrib` directory.

10. **msgfmt**. Once the `LL.po` file does exist, the *make* in `po` will generate a `gmo` file.

11. **LL.gmo**. This is a compiled language file that can later be installed as a *mo* file when the program is officially installed.

12. **make install**. This command installs not only the program, but the localization files created above.

13. /***prefix***/**LL**/**PROGRAM.mo**. The *gmo* file is copied here, where *prefix* is usually `/usr/local`. *LL* expands to something like *es/LC_MESSAGES*. If you look in the directory `/usr/local/share/es/LC_MESSAGES`, you may find *mo* files from other installed applications.
14. /***prefix***/**bin**/**PROGRAM**. This is where the program itself goes.
15. **Hello world!**. Running the program will produce this message. If you export `LC_ALL`, etc. to empty strings, and export `LANG="es_ES"`, this message will appear as **!Hola, mundo!**.

4.3 Marking Text and Building Translations

The discussion in the previous section is a bit abstract. Let's make it more concrete.

We have a problem, perhaps. The only way to get a `*.pot` built is to manually change to the `po` directory and then do (for example):

```
$ make libxpccut.pot
```

We are not sure why it isn't built automatically; either PACKAGE isn't defined, or we need to update `po/Makevars`. So keep this additional step in mind; it is included below.

And yet, the above now seems to be wrong, as we've found the `libxpccut.pot` file to be up-to-date!

1. Prep the C source code with invocations of the `_()` and `N_()` macros. Use the former to mark text you want translated, and the latter to mark text that will be ignored. One can assume that unmarked text is also ignored.
2. Create the initial project template file by extracting `_()` invocations from the source code. The first step is for manual execution, while the second step is quicker.
   - xgettext [lists of directories?]
   - Change to the po directory, and then do
     ```
     $ make libxpccut.pot
     ```
     The name 'libxpccut' is the name of the library, and the specific name here is just an example. The project variable used here is PACKAGE="libxpccut".
3. Generate an up-to-date `xx.po` for language `xx`. The *first time* you do this, perform this command:
   ```
   $ cp libxpccut.pot xx.po
   ```
   This step creates the initial `po` file in the developer's language. This file can the be translated in subsequent steps. Again, to be clear, only do this action *once*.
4. The *rest of the time*, do the following command. This step merges in changes and comments out obsolete translations.
   ```
   $ msgmerge -o new.pot es.po libxpccut.pot
   ```
   This command follows the format *msgmerge existing.po uptodatesourcestrings.pot*. It takes the currently-translated strings in the latest version of the translation module, `es.po`, and generates a new file, `new.pot` that contains these translations, plus modifications based on new text strings, or changes of the existing strings within each source-code module.
5. Modify `new.pot` (for language `xx`) to create or extend a translation.
6. Back up the old file (`es.po`) elsewhere, just in case, then .....
7. If it is a new language to the project, add it to the `LINGUAS` file.
8. Convert the `po` file to a machine file (`gmo`) using *msgfmt*.
   ```
   $ msgfmt [option] xx.po
   ```
9. Rebuild and install the project.

4.4 Building the Internationalized Application

Let's recapitulate the whole process from the very beginning, in terms of actual commands:

1. `$ tar xf xpchello-2009-04-23.tar.bz2`
   Unpack the code. Preferably, do this in the `xpc_suite` directory, because that's the home for all **XPC** projects. However, for better testing of the building and the installation of the other **XPC** libraries (e.g. *libxpccut*, then unit-test library), feel free to untar this file somewhere else, such as `~/tmp`.
2. `$ cd xpchello`
   Change to the project directory.
3. `$ ./bootstrap-po -intl-boot`
   Run the script, telling it to create the `po` files. This needs to be done only once. However, the Hello project has been set up to allow it to be built and torn down over and over – the files edited by the user have been preserved in the `contrib` directory. Note that the user must hit *Enter* because running *gettextize* forces a prompt.
4. `$ make`
   Generate the documentation in the `doc` directory, the `es.gmo` and other files in the `po` directory, and the object modules and the `xpchello` executable in the `src` directory.
5. `# make install.`
   Install the documentation, the message catalog(s), and the application. **(WARNING: Currently, the application doesn't install properly yet. Consider this goal a low priority.)** The message catalog `es.gmo` is copied to `/usr/local/share/locale/es/LC_MESSAGES/xpchello.mo`.
6. `$ xpchello`
   Run this newly-installed application, and verify that it outputs **Hello, world!**.
7. `$ export LC_ALL=`
   `$ export LANG=es_ES`
   Disable LC_ALL and change to the Spanish locale.
8. `$ xpchello`
   Run the application again, and verify that it now outputs **!Hola, mundo!**. If it does not, you make have to perform the next step, and try the current step again.
9. `# dpkg-reconfigure locales`
   Check-mark the `es_ES.ISO-8859-1` locale, and hit the `Ok` button. Now the `xpchello` command should work properly.
10. `$ ./bootstrap -intl-clean`
    Clean out all generated files. The project will no longer be buildable. But you can start over again at step 3!

Note

When a program looks up locale-dependent values, it does this according to the following environment variables, in priority order:

1. `LANGUAGE`
2. `LC_ALL`
3. `LC_xxx`, according to selected locale category: `LC_CTYPE`, `LC_NUMERIC`, `LC_TIME`, `LC_COLLATE`, `LC_MONETARY`, `LC_MESSAGES`, ...
4. `LANG`

Variables whose value is set, but is empty, are ignored in this lookup.

LANG is the normal environment variable for specifying a locale.

LC_ALL is an environment variable that overrides all of these. It is typically used in scripts that run particular programs. Some systems, unfortunately, set LC_ALL in /etc/profile or in similar initialization files. As a user, you therefore have to unset this variable if you want to set LANG and optionally some of the other LC_xxx variables.

4.5 More Information

One internationalization package we want to note, for future reference, is the Debian package "iso-codes".

http://packages.debian.org/testing/misc/iso-codes

```
ISO language, territory, currency codes and their translations

This package provides the ISO-639 Language code list, the ISO-4217
currency list, the ISO-3166 Territory code list, and ISO-3166-2
sub-territory lists.

It also (more importantly) provides their translations in .po form.
```

4.6 References

- http://www.gnu.org/software/gettext/manual/gettext.html
- http://inti.sourceforge.net/tutorial/libinti/internationalization.html

Chapter 5

XPC Hello Linkage How-To

This document provides a discussion of the **XPC** *Hello* project and how it is linked to other **XPC** libraries to create the **xpchello** demonstration application.

Table of Contents\n

5.1 Introduction to 'pkgconfig' Linkage

The **xpchello** application code is linked to one of the **XPC** libraries using "*pkg-config* linkage. The **GNU** *pkg-config* system provides a way to do the following things necessary for linking an application:

1. Determine where the `*.h` header files for the library are installed.
2. Determine where the `*.a`, `*.la`, `*.so` and other actual library file are installed.
3. Provide these directories to the automake process so that the files can be used to build the current project.
4. Determine the additional compiler flags that should be used.
5. Determine the additional linker flags that should be used.
6. Provide these flags to the automake process.
7. Provide other information about a library, such as who wrote it, and a description of it.

As is common in open-source projects, this information is stored in a human-readable text file. You can examine one for yourself by viewing one of the files in your `/usr/lib/pkgconfig` directory.

The "Hello" application uses only one of our **XPC** libraries – the **XPCCUT** ("XPC C Unit-Test") library. And actually, only the test application links to it.

Furthermore, the test application build process assumes that the **XPCCUT** library has been installed in one of the expected directories (such as `/usr`, `/usr/local`, or `/opt`.

5.2 The Linkage Process

This section describes how the **autoconf** and **automake** files were set up to link to the install **xpcccut** library.

These steps can be done manually. For example, to obtain the flags necessary to compile and link against the **xpccut** library, the developer could enter these commands:

```
$ MY_CFLAGS=`pkg-config --cflags xpccut`
$ MY_LIBS=`pkg-config --libs xpccut`
```

But the **GNU** tools make it even easier, and also perform a number of tedious checks and setup actions.

By the way, be sure to read the `pkg-config(1)` man page.

```
$ man pkg-config
```

5.2.1 Modifying the configure.ac file

The `configure.ac` file is the template for configuring the whole project. It contains directives to test for various libraries and resource, and directives to set up macros and make-files.

For the purposes of linking to the **xpccut** library using the *pkg-config* system, the following lines are added:

1. `PKG_CHECK_MODULES([XPCCUT], [xpccut-1.0 >= 1.0])`
   This line runs an **m4** macro that looks for the correct version of the **xpccut** library. It calls the *pkg-config* application and creates the following symbols:
   - `XPCCUT_CFLAGS`
   - `XPCCUT_LIBS` You can actually look at the installed `xpccut-1.0.pc` file to see that the **C** flags are something like *-I/usr/local/include/xpc-1.0* and the *ld* flags are something like *-L/usr/local/lib/xpc-1.0 -lxpccut*. Also note that this check looks only at the major and minor version numbers, not the patch-level number.
2. `CFLAGS="${CFLAGS} $COMMONFLAGS $XPCCUT_FLAGS"`. The resultant flags are added to `CFLAGS`.

In the next section, we skip the `src` directory and its make-file, where only the normal *xpchello* application is built, to examine the test application's setup, because that's the application that links to the **xpccut** library.

5.2.2 Modifying tests/Makefile.am

The **automake** template make-file for the tests application has to be told where to find the **xpccut** library. Here are the lines that have to be added to it:

1. `XPCCUT_LIBS = @XPCCUT_LIBS@`
   This line allows *configure* to expose the `XPCCUT_LIBS` variable created by the `PKG_CHECK_MODULES()` module to be added to the `Makefile` that gets generated in the `tests` directory.
2. `xpchello_test_LDADD = @LIBINTL@ -lpthread`
   This line adds something like `-L/usr/local/lib/xpc-1.0 -lxpccut` to the normal linkage specification needed for building the test application.
3. `xpchello_test_lt_LDADD = @LIBINTL@ -lpthread`
   This line is the same as the previous one, but builds a dynamically-linked version of the test application.

Let's recapitulate the whole process from the very beginning, in terms of actual commands. Note that we currently provide tar files with dates in them, instead of version numbers. The current version is 1.0.0. The dates below may not represent the lastest tar files you can find.

1. Install and build the *xpccut* library code:
   (a) `$ tar xf xpccut-2009-04-23.tar.bz2`
   Unpack the code. Preferably, do this in an `xpc_suite` directory, because that's the home for all **XPC** projects.
   (b) `$ cd xpccut-1.0`
   Change to the project directory.
   (c) `$ ./bootstrap`
   By default, this script also configures for a release build. If you want to see the available options, add the `-help` option.
   (d) `$ make`
   Build the library, documentation, and unit-test application.
   (e) `# make install`
   As root, install the library, header files, man pages, pkgconfig file, and the HTML version of the Doxygen documentation.
2. Install and build the *xpchello* application code:
   (a) `$ tar xf xpchello-2009-04-23.tar.bz2`
   Unpack the code. Preferably, do this in the `xpc_suite` directory, because that's the home for all **XPC** projects. However, for better testing of the building and the installation of the other **XPC** libraries (e.g. *xpccut*, then unit-test library), feel free to untar this file somewhere else, such as `~/tmp`.
   (b) `$ cd xpchello-1.0`
   Change to the project directory.
   (c) `$ ./bootstrap -super-boot ...`
   Run this command and others as explained in the The 'gettext' Process section. *gettextize* forces a prompt. If the **xpccut** library has *not* been installed, you will see the following message and an explanation:

```
No package 'xpccut-1.0' found
```

   (d) `$ cd tests`
   Change to the test directory.
   (e) `$ make`
   Generate the *xpchello_test* test application.

Chapter 6

GNU Automake in the XPC Library Suite

This section describes the GNU automake setup for the XPC libraries and test applications.

6.1 GNU Automake

First, note that the main difference between **XPC 1.0** and **XPC 1.1** is that the documentation, *pkgconfig* support, and internationalization support have been consolidated in the top-level directory, instead of being spread over all the library packages. This consolidation makes it a lot easy to maintain **XPC**.

GNU *automake* is a tool for describing project via a simplified set of Makefile templates arranged in a recursive (hierarchical) fashion. *Automake* provides these benefits:

- Shorter, simpler `makefiles` (really!).
- Support for a large number of targets, with almost no effort.
- The ability to configure the files for different development platforms.

Although the term "simplified" is used, *automake* is still complex. However, it is much less complex than writing a bare `makefile` to do the same things an *automake* setup will do. The *automake* setup will do *a lot* more.

Other systems have come along because of the complexity of *automake*. A good example is *cmake* (`http://www.cmake.org/HTML/index.html`). Another good example is *Boost.Build* (`http://www.boost.org/boost-build2/`). However, these systems seem to end up being just as complex as *automake*, since they, too, need to handle the multifarious problems of multi-platform development. We really wanted to transition to *Boost.Build*, but couldn't get it to handle *doxygen* in the way we wanted, even though that product is supposed to be supported.

The present document assumes that you are familiar with *automake* and have used it in projects. The discussion is geared towards what features *automake* supports in the **XPC** projects.

6.2 The 'bootstrap' Script

First, note that one can run the following command to jump right into using the `bootstrap` script:

```
$ ./bootstrap --help
```

Each **XPC** sub-project used to have its own `bootstrap` script, but maintaining them was too difficult. So, for **XPC 1.1**, there is only one `bootstrap` script. In addition, we've off-loaded some of its functionality to a more flexible `build` script.

The **XPC** project provides a `bootstrap` script to handle some common tasks. This script makes it easy to start a project almost from scratch, configure it (if desired), clean it thoroughly when done, pack it up into a tarball, and prepare it for making.

The `bootstrap` script does a lot of what one has to do to set up a project. Thus, it also serves as a description of the operations required in setting up almost any **GNU** *automake* project.

The `bootstrap` script also does a lot more cleanup than the various "make clean" commands provided by *automake*.

One reason the `bootstrap` command does so much is that, even at this late date, we are *still* finding that there are features of *automake* that we don't know how to use.

When run by itself, `bootstrap` first creates all of the directories needed for the project:

```
$ mkdir -p aux-files
$ mkdir -p config
$ mkdir -p m4
$ mkdir -p po
$ mkdir -p include
```

Some of these directories may already exist, of course.

The user will normally create `src` for the library code, and `tests` for unit-tests and regression tests. The user may create other directories if the project is more complex. A `doc` directory is commonly needed to hold documentation. Our projects also include a `doc/dox` directory for **Doxygen** documentation.

`bootstrap` recreates all of the **GNU** management files it needs to create. For **XPC 1.1**, we no longer bother keeping up with the following files:

```
ABOUT-NLS
AUTHORS
NEWS
INSTALL
TODO
```

We avoid them by using the `foreign` keyword in the `Makefile.am` files and in the *automake* call, to avoid *automake* complaining about "missing files".

Some of the files are automatically generated by the **GNU** *autoconf* tools.

Then `bootstrap` makes the following calls:

```
$ PKGCONFIG=yes               (or no if not available)
$ autopoint -f
$ aclocal -I m4 --install
$ autoconf
$ autoheader
$ libtoolize --automake --force --copy
$ automake --foreign --add-missing --copy
$ cp contrib/mkinstalldirs-1.9 aux-files/mkinstalldirs
```

The actions above are the default actions of the `bootstrap` script. Command-line options are available to tailor the actions taken, to implement some common operations:

```
--configure
```

The option above specifies that, after bootstrapping, run the `configure` script with no special options. This is no longer the default action of the `bootstrap` script. It is now preferred to use the `build` script.

```
--no-configure
 -nc
```

The option above specifies to not proceed to run the `configure` script. Simply bootstrap the project. Use this option, and later run `./configure` or `build` for building for debugging, stack-checking, and many other options.

```
--enable-debug
--debug
 -ed
```

The options above cause the `bootstrap` script to run `configure` with the `-enable-debug` and `-disable-shared` flags.

Again, it is now preferred to use the `build` script for this purpose.

```
--cpp-configure
```

The option above specifies to configure the project with the **XPC**-specific `-disable-thisptr` option. This option is good for usage in **C++** code, where the developer is passing unit-test structure objects by reference to **C++** functions. Since the item is a reference to an actual (non-pointer) class member, it is guaranteed to exist, and therefore null `this` pointer checks are unnecessary. This option can be combined with the `-debug` option.

Warning

> This option will make a version of the unit-test (unit_test_test in the tests directory) that will segfault due to attempts to use the null pointers supplied in some of the tests.

In most cases, the C++ developer will want to treat the resulting library as a *convenience library*:

http://sources.redhat.com/autobook/autobook/autobook_92.html

As such, the `-disable-thisptr` version of **XPC** libraries should *not* be installed by the developer. Otherwise, **C** unit-test applications will lose the benefits of null-pointer checks. The **XPC CUT++** build process links in this specially-built version of the **CUT** library into the **CUT++** library, and the install process does make sure that the**C** header files are also installed.

The following `bootstrap` options for **XPC 1.0** no longer exist in **XPC 1.1**.

```
--make, -m
--generate-docs
--no-generate-docs
--no-automake
```

The following options still apply, but they operate on *all* **XPC** projects as whole:

```
--clean
  clean
```

The options above specify to delete *all* derived and junk files from the project.

```
--debian-clean
```

The option above specifies to just remove the files created by running the various debian/rules script options.

The default of the `bootstrap` script is to bootstrap the project. To set up other configurations, manually call the desired `configure` command. For example, to generate (for example) debugging code using static libraries:

```
./bootstrap
./configure --enable-debug --enable-coverage --enable-thisptr --disable-shared
```

Also see the `build` script, which is the new preferred (and easier) method of configuring (and building, and testing) the projects.

6.3 The 'build' Script

First, note that one can run the following command to jump right into using the `build` script:

```
$ ./build --help
```

The `build` script is new with **XPC 1.1**.

It can be run after the `bootstrap` script is run.

It supports the same *configure* process as `bootstrap`, but it also add the following features:

- Build *out-of-source* configurations. This allows the developer to create directories suchs as `release` and `debug` in which to build the code.
- Perform a whole-project *make*, which builds all of the **XPC** source code, `pkgconfig` files, `*.po` files, and documentation.
- Selectively build a single project or the documentation.

The following `build` options cause the projects to be configured with certain options and built in a special output subdirectory:

```
Name           Directory      Build Options

--release      release        None
--debug        debug          --enable-debug --disable-shared
--cpp          cpp            --disable-thisptr
--build name   name           None
```

These options work by creating the directory, changing to it, and running a `../configure` command. This action creates only `Makefiles` that mirror the original directory structure, and cause the modules from the original directory structure to be compiled and built into the new directory structure. These options offer a way to maintain multiple builds of the same code.

The following option supports selecting what to build:

```
--project name
```

The potential values of name are (or will be):

```
all            Build all projects and the documentation. Default.
doc            Just build the documentation.
xpccut         Build the xpccut C project.
xpccut++       Build the xpccut++ C++ project.
xpc            Build the xpc C project.
xpc++          Build the xpc++ C++ project.
xpcproperty    Build the xpcproperty project.
xpchello       Build the xpchello project.
```

The default action, after configuration, is to perform the *make*:

```
--make
--no-make
```

The rest of the options are also found in `bootstrap`. See the `-help` option.

Now, if the `all` project is selected, the following sequence of actions will occur:

(NOT YET FULLY IMPLEMENTED)

1. xpccut
   (a) Build `xpccut` normally in the selected out-of-source directory (preferably `release`, but `debug` or a directory name you pick yourself will also work).
   (b) Build `xpccut` with `-disable-thisptr` in the `cpp` directory.
   (c) Build `xpccut++` normally in the selected out-of-source directory, linking it to the`-disable-thisptr` version of `xpccut`.
2. xpc
   (a) Build `xpc` normally in the selected out-of-source directory (preferably `release`, but `debug` or a directory name you pick yourself will also work).
   (b) Build `xpc` with `-disable-thisptr` in the `cpp` directory.
   (c) Build `xpc++` normally in the selected out-of-source directory, linking it to the`-disable-thisptr` version of `xpc`.
3. Build `xpcompmgr++` normally in the selected out-of-source directory.
4. Build `xpcproperty` normally in the selected out-of-source directory.
5. Build `xpchello` normally in the selected out-of-source directory.

6.4 The configure.ac Script

The **XPC** projects support a number of build options. Some are standard, such as compiling for debugging and internationalization support. Some options are peculiar to **XPC**, such as conditional support for error-logging and pointer-checking.

As usual in **GNU** *automake*, the options are supported by switches passed to the `configure.ac` script, supported by corresponding *m4* macros in the `m4` directory. Here are the options:

```
  Switch             Default  Status/Description

--enable-debug          no    Compiles for gdb
--enable-lp64           no    Very problematic at present, useless
--enable-stackcheck     no    Makes segfaults much more likely, a good test
--enable-thisptr        yes   Activates thisptr() null-pointer check
--enable-errorlog       yes   Provides run-time logging at various verbosities
--without-readline      no    readline() support enabled by default
--enable-coverage       no    Coverage testing using gcov
--enable-profiling      no    Turns on gprof support
--enable-motorola       no    Compiles for MC68020 to check the code
```

The Motorola option requires a version of *gcc* called something like *68000-gcc*, which is a version of *gcc* built for cross-compiling. We still have a lot more to learn about this issue. It isn't ready, and may be wrong-headed anyway.

Some of the options can take additional parameters to tweak how they work. The options that can do that are:

- `-enable-debug`. In addition to the standard `yes` and `no` values, also accepted are `gdb` (the default) and `db` (the BSD/UNIX variant). See Debugging using GDB and Libtool for more information.
- `-enable-coverage`. See Building For 'gcov' Usage for more information.
- `-enable-profiling`. See Building For 'gprof' Usage for more information.

These options are implemented by the `AC_XPC_DEBUGGING` macro call in the **XPCCUT**'s `configure.ac` file.

6.5 Making the XPCCUT Library

This section assumes that you have bootstrapped and configured the library in some manner. It also assumes that you want to install the library for usage by another project.

6.5.1 Making the Library for Normal Usage

There's not much to explain here. Once the **XPC CUT** library is bootstrapped and configured, the rest of the process is very simple and standard at this point:

```
$ make
$ make check   [or "make test"]
# make install
```

This sequence of steps installs the library (`libxpccut` in static and shared-object forms), the `unit_*.h` header files, the HTML version of the documentation, a rudimentary *man* page, a primitive (and usually incomplete) Spanish translation for the error and information messages, and a *pkg-config* file that can be referred to in the `configure.ac` file of projects that want to link properly to the **XPC CUT** library.

There is one special project that is deeply dependent on the **XPC CUT** library, and that is the **XPC CUT++** library and test application. This special case is discussed in the next subsection.

6.5.2 Making the Library for XPC CUT++

The **XPC CUT++** library is a C++ wrapper for the **XPC CUT** library. As such, it basically requires the **XPC CUT** library to be built at the same time. It does not require the **XPC CUT** library to be installed fully, however. Instead, the library is built and linked into the **C++** version of the library, and only the header files are installed.

Rather than have the developer deal with all of this, the `bootstrap` script for the **XPC CUT++** library looks for the **XPC CUT** library source-code in the same base directory (e.g. `xpc_suite`. If it finds it, it bootstraps it configures it with the same user options as passed to the `bootstrap` script, plus a "secret" option to disable the this-pointer checking, for a little extra speed.

Then the **XPC CUT** library is built.

When the **XPC CUT++** library is built, the **XPC CUT** library is then added to the **C++** version of the library, so that **C++** applications need include only one library.

We're also working on a **CUT++** sample application that will help us make sure that the **XPC CUT++** combines properly the **C** and **C++** components, and installs properly the **C** and **C++** header files. See the xpc_cutsample_↩ introduction page for more information.

6.6 Linking to the XPC CUT Library

Using the **XPC CUT** library in an external application requires that the **XPC CUT** library be installed, as noted by the simple instructions in the previous section.

The application project must make some settings in the following project files:

1. `configure.ac`
2. `app/Makefile.am` (as applicable)
3. `tests/Makefile.am` (as applicable)

In the application's `configure.ac` file, the following line is necessary:

```
PKG_CHECK_MODULES([XPCCUT], [xpccut-1.0 >= 1.0])
```

This directive causes the `XPCCUT_CFLAGS` and `XPCCUT_LIBS` macros to be defined as if the following commands were run, and the output assigned to the respective macros:

```
$ pkg-config --cflags xpccut-1.0
$ pkg-config --libs xpccut-1.0
```

The respective outputs of these commands are

```
-I/usr/local/include/xpc-1.0
-L/usr/local/lib/xpc-1.0 -lxpccut
```

These values need to be added to the command lines. First, in `configure.ac`, the `CFLAGS` value needs to be augmented:

```
CFLAGS="$CFLAGS $CFLAGSTD $COMMONFLAGS $XPCCUT_CFLAGS"
```

In the `Makefile.am` in the application directory, the following are needed. First, expose the `-L` and `-l` directives noted above.

```
XPCCUT_LIBS = @XPCCUT_LIBS@
```

Also, provide this flag to the `LDADD` macro, as in the following sample:

```
xpchello_test_LDADD = @LIBINTL@ -lpthread $(XPCCUT_LIBS)
```

Note that the `XPCCUT_CFLAGS` macro is automatically carried through to the application's subdirectory as part of the *automake* process.

For a good example, see the XPC "Hello" Application page and the **XPC** *Hello* application it describes.

6.7 Libtool

This section describes *libtool* and contains notes from other documents in the suite.

6.7.1 Libtool Versioning

We support two types of version in the **XPC** suite:

- Package versioning
- Libtool versioning

Package version indicates when major changes occur in a package, and also indicates what sub-directory the libraries are installed in. For example, the current package version of the **XPC** suite is `1.0.4`. The libraries and header files are included in directories with names of the form `xpc-1.0`. The `4` increments whenever we have made enough changes to make a new release. If the changes modify or delete an existing interface, then the `0` will get incremented to `1`, and the new storage directory will become `xpc-1.1`. If we rewrite any major component of the suite, then the `1` in `1.0` will increment, and the new storage directory will become `xpc-2.0`. We don't anticipate this kind of major change, though.

Although package version is visible to the user, and partly determines the compatibility of the **XPC** libraries, the actual interface versioning is determined by *libtool* versioning. The following discussion is a succinct description of this link:

`http://www.nondot.org/sabre/Mirrored/libtool-2.1a/libtool_6.html`

Imagine a simple version number, starting at 1. Consider each significant change to the library as incrementing this number. This increment reflects a new interface. *libtool* supports specifying that interface versions from `i` to `j` are supported.

libtool library versions are described by three integers:

- `current`. The most recent interface number that this library implements.
- `revision`. The implementation number of the current interface.
- `age`. The difference between the newest and oldest interfaces that this library implements.

The library implements all the interface numbers in the range from `current - age` to `current`.

If two libraries have identical `current` and `age` numbers, then the dynamic linker chooses the library with the greater `revision` number.

The **XPC** suite actually abused the distinction between the package and *libtool* version information. So, for **XPC** package version `1.0.4`, we are resetting the *libtool* version to `0:0:0`

When we make public the next version with any changes at all, the new versions will be:

- **Package**: `1.0.5` (patch++)
- **libtool**: `0:1:0` (revision++)

If, instead of just any simple change, we added an interface:

- **Package**: `1.0.5` (change in patch number)
- **libtool**: `1:0:1` (current++, revision=0, age++)

Our reasoning here is that the additional interface cannot break existing code, so that the same installation directory, `xpc-1.0`, can be used.

If we changed or removed an existing interface, or refactored the whole set of projects, old code will break, so we'll need a new installation directory.

- **Package**: `1.1.0` (minor++, patch=0)
- **libtool**: `1:0:0` (current++, revision=0)

If the changes were really major (a rewrite of the libraries):

- **Package**: `2.0.0` (major++, minor=0, patch=0)
- **libtool**: `1:1:1` (current++, revision++, age++)

6.8 pkgconfig

GNU *pkgconfig* is a way to help applications find out where packages are stored on a system. The common places to store them are (in descending order of likelihood on a Linux system):

```
/usr            Apps installed by the default package manager.
/usr/local      Apps installed from source code.
/opt            Solaris and Java flavored apps such as OpenOffice
$HOME           Apps installed by the user in his/her home directory.
```

As an example, we were trying to install totem (a media player for the Gnome desktop environment) on a Gentoo system. The Gentoo emerge system provided an old version, 1.0.4, but we wanted the latest, which happened to be 1.5. Thus, we went to the totem site and downloaded the source. Running configure revealed a dependency on "iso-codes". However, that package was masked on Gentoo.

Since it was a Debian package, we found it at

http://packages.debian.org/testing/misc/iso-codes

and downloaded the source tarball. We did the `./configure; make; make install` mantra, and then retried the configuration of the totem build.

Still missing. We looked in `/usr/lib/pkgconfig`, but no `iso-codes.pc` file. Nor was it in `/usr/local/lib/pkgconfig`, which is where we expected to find it.

So we look in the iso-codes build area, and see the iso-codes package-configuration file. We copied it to `/usr/local/lib/pkgconfig`.

The totem configuration was then built with no problem.

6.9 Endorsement

A rousing endorsement of Libtool and Pkgconfig from Ulrich Drepper:

http://udrepper.livejournal.com/19395.html

The second problem to mention here is that not all unused dependencies are gone because somebody thought s/he is clever and uses -pthread in one of the pkgconfig files instead of linking with -lpthread. That's just stupid when combined with the insanity called libtool. The result is that the -Wl,–as-needed is not applied to the thread library.

Just avoid libtool and pkgconfig. At the very least fix up the pkgconfig files to use -Wl,–as-needed.

Hmmm, we need to remember that "at the very least" advice....

6.10 References

1. http://wiki.showmedo.com/index.php/LinuxBernsteinMakeDebugSeries Discusses an improved version of automake, called "remake".
2. http://www.gnu.org/software/libtool/manual/automake/VPATH-Builds.html Discusses building the object files in a separate subdirectory.
3. http://www.nondot.org/sabre/Mirrored/libtool-2.1a/libtool_6.html Discusses versioning in libtool.
4. http://www.adp-gmbh.ch/misc/tools/configure/configure_in.html Walks the user through a configure.ac file; very helpful.
5. http://www.geocities.com/foetsch/mfgraph/automake.htm Provides a nice summary of autoconf/automake projects.

Chapter 7

Debugging using GDB and Libtool

This file describes the very basics of debugging using *gdb*.

Table of Contents\n

7.1 Introduction to Debugging

This document describes how to do basic debugging.

It is currently somewhat incomplete.

For code-coverage and profiling, see Unit Test Coverage and Profiling

7.2 Basic Debugging

We won't pretend to teach debugging here; this section is just to get you started.

See some of the items in References for much more information.

7.2.1 Building Without Libtool Shared Libraries

The first thing to note is that it is easier to debug if the project is built with only the static libraries in *Libtool*. With *libtool* and shared libraries, you have to use *libtool* to run the debugger, as described in section Debugging a Libtoolized Application.

However, if the application is built using only the static libraries, then the library code is part of the application, and the application can be debugged normally. To build the application without shared libraries, they must be disabled. Run (or re-run) the 'configure' script with the following options:

```
$ ./configure --enable-debug --disable-shared
```

If you know you want to debug the project before you bootstrap it, then bootstrap it this way:

```
$ ./bootstrap --enable-debug
```

This command sets up the make system, then executes the `./configure` command shown above, including the –disable-shared.

7.2.2 GDB

7.2.3 CGDB

7.2.4 DDD

7.3 Debugging a Libtoolized Application

An application that has been created using Libtool, but that has not yet been official installed, requires some special handling for debugging.

As with a normal application or library, the source-code a build system for a project reside in a small directory hierarchy with directories such as `src`, `include` (or, in our projects, `include/xpc`), and `tests`.

In a normal library, the library file (archive) is created in the `src` directory.

This library is linked to the test application, which is created in the `tests` directory, and is a binary file of ELF format.

But there are some differences once Libtool is used:

- `src`.
  - `libxpccut.la`. Instead of a single static archive (library) that is created, this file is created. It contains some information about the shared library, static library, installation location, and other items of information that describe the newly-created library.
  - `.libs`. The newly created static library is deposited here, instead. Here is what is in this hidden directory:
    - The original, static library, `libxpccut.a`.
    - The shared library, `libxpccut.so.0.1.1`.

    * Links to the static and shared library, similar to what one would see when the library is installed.
    * All of the object modules for each source-code module, `*.o`.
    * A copy of `libxpccut.la`, called `libxcpcut.lai`, that differs only in having a tag *installed* set to *yes* instead of *no*.

- `tests`.
  - `unit_test_test`. Instead of a binary executable file, this item is actually a script. When executed, it looks as if the original application is being executed directly. But it is actually a script that sets up so the shared libraries can be found and accessed while the unininstalled (and hidden) application runs.
  - `.libs`. Just as with the `src/.libs` directory, this hidden directory contains the actual binary for the application. There are two binaries there – we don't yet know what the difference is – a topic for the future.

Try running the following command:

```
$ gdb unit_test_test
```

gdb will complain:

```
"../xpccut-1.1/tests/unit_test_test": not
in executable format: File format not recognized
```

This makes sense, since `unit_test_test` is a script. In order to debug the application, we have to let Libtool execute the debugger:

```
$ libtool --mode=execute gdb unit_test_test
$ libtool --mode=execute cgdb unit_test_test
$ libtool --mode=execute ddd unit_test_test
```

Commands like these can also be used to execute valgrind and strace, too.

7.4 References

- http://heather.cs.ucdavis.edu/~matloff/debug.html Norm Matloff's Debugging Tutorial
- http://nostarch.com/debugging.htm N. Matloff and P.J. Salzman (2008) "The Art of Debugging with GDB, DDD, and Eclipse", a very good tutorial book.

Chapter 8

Unit Test Coverage and Profiling

This file provides a tutorial on using the **GNU** test-coverage and profiling tools.

Table of Contents\n

8.1 Introduction to Coverage Testing and Profiling

This document describes how to use *gcov* (coverage testing) and *gprof* (code profiling), using the **XPCCUT** unit-test project as an example. As befits the purpose of that project, that's where this document can be found.

Coverage testing analyzes the code to determine which functions an application exercises during its run.

Code Profiling analyzes executing code to determine where most of the CPU time is being spent.

Together, both methods ensure that your code is as well-tested and as fast as you can make it.

8.2 Installing 'gcov' and 'gprof'

The means of installing *gcov* and *gprof* vary with your Linux (or BSD) distribution. (We won't even get into *Solaris*.)

gcov is provided as part of the *gcc* package, and *gprof* is provided by *binutils* package (in Debian, anyway).

Other potentially helpful (Debian) packages are available:

- `lcov`. A set of PERL scripts that provide colored HTML output with overview pages and various views of the coverage.
- `ggcov`. A GUI for browsing C test-coverage results.

8.3 The 'gcov' Coverage Application

What is *gcov*? From the GCOV(1) man page.

```
gcov is a test coverage program.  Use it in concert with GCC to analyze
your programs to help create more efficient, faster running code and to
discover untested parts of your program.  You can use gcov as a
profiling tool to help discover where your optimization efforts will
best affect your code.  You can also use gcov along with the other
profiling tool, gprof, to assess which parts of your code use the
greatest amount of computing time.
```

The *GNU gcov* manual can be found at:

`http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

A good PDF version is available in Chapter 9 of the *GNU gcc* manual, PDF version:

`http://gcc.gnu.org/onlinedocs/gcc.pdf`

Also very helpful is *The Definitive Guide to GCC*, Chapter 6. See the References section for more details.

8.3.1 Building For 'gcov' Usage

An application to be analyzed with *gcov* needs to be compiled in a special manner:

- Use a *gcc* compiler.
- Use no optimization.
- Use the `-fprofile-arcs` option.
- Use the `-ftest-coverage` option.

In the **XPCCUT** project, this is all done easily with these commands:

```
$ ./bootstrap --clean                    (if needed)
$ ./bootstrap --no-configure             -or-
$ ./bootstrap -nc
$ ./configure --enable-coverage
```

Note the `-enable-coverage` option. It is implemented as part of the `m4/xpc_debug.m4` script, which is specified by the `AC_XPC_DEBUGGING` macro call in the **XPCCUT**'s `configure.ac` file.

In addition to the normal arguments, you can specify the following forms:

- –enable-coverage=gdb (the default)
- –enable-coverage=db

The first variant is the default, which specifies *GNU* **gdb** compatibility. The **gcc** flags for the **gdb** variant are:

```
-O0 -ggdb
```

The second variant supports the *BSD* **db** program. The **gcc** flags for the **db** variant are:

```
-O0 -g
```

Once the configuration has been set up, run *make* in the `src` directory.

```
$ cd src
$ make
```

Note that there are `*.gcno` files that are created by this compile, one for each C module that is built. Moreover, these are built both in the `src` and the `src/.libs` directories. (The latter is a hidden directory created by *libtool*.

Next, run *make* again in the `tests` directory.

```
$ cd ..
$ cd tests
$ make
```

Note, again, that there are `*.gcno` files that are created by this compile, one for each **C** module that is built in the `tests` directory.

8.3.2 Running for Coverage

Now we want to run the application once, to see how much module coverage we have.

General principles:

- It seems best to run the application from the project root directory, not from its `tests` directory.
- A file *`my_c_module`*`.gcda` file will be created in the `tests` directory by running the application.
- Additional `*.gcda` files will be created in the `src/.libs` directory.
- The run information in this file accumulates with each run. To start fresh, the `*.gcda` files have to be deleted.
- *Question*: What are the `*.bb` and `*.bbg` files referred to in some of the documentation? They don't appear in our project, so are perhaps obsolete.

Run the application (here, called `unit_test_test`).

```
$ ./tests/unit_test_test
```

Note that we are running it from the *root* of the project. This is necessary in order for files in the `src` directory to get their analysis done and their output files created.

The following items are generated:

- Generates `./tests/unit_test_test.gcda`
- In a *libtool* project such as this one, it also generates `./src/.libs/*.gcda` (one for each C module in `src`).
- However, no `*.gcda` files are generated in `src` itself.

8.3.3 Running 'gcov' for Analysis

Once the test data are generated, with *gcov* information, they need to be analyzed using *gcov*.

Here are some commands to try. These commands can be done inside the proper directory (either `src` or `tests`, as appropriate.)

```
$ gcov <i>my_c_module.c</i>
```

The command above must be run for which the analysis of coverage data is desired.

```
$ gcov -b my_c_module.c
```

The command above generates branch probabilities.

```
$ gcov -f my_c_module.c
```

The command above analyzes function calls.

All of the commands above are useful for analyzing a library module.

In the following command, both are done, and output goes to an additional log file that can be examined at leisure.

```
$ cd tests
$ gcov -b -f *.c > output.log
```

The run produces a `unit_test_test.c.gcov` annotated source file. If the `-f` switch was provided, then `output.log` will also contain the summary information.

To make it easier to do all of this, a `gcov` target is provided in the top-level `Makefile.am` make-file. As a synonym, a `coverage` target is also provided.

Although there is a lot to be looked at in the `*.gcov` file, let's limit ourselves to simply checking for lines that did not get executed. Examining the `unit_test_test.c.gcov` annotated source file, we can search for unexecuted lines – they have `#####` in them.

The first one we find is

```
#####:  157:         unit_test_status_pass(&status, true);
```

and the second is

```
#####:  162:            fprintf(stdout, "  %s\n", _("No values to show ...
```

There are more, but let's just try to eliminate these two unexecuted lines by subsequent runs of `unit_test_test` with additional arguments.

```
$ ./tests/unit_test_test --group 2 --case 2
$ gcov -b -f *.c > output.log
$ vi unit_test_test.c.gcov
```

We tried to get the test-skipping mechanism to work, but looking at the `*.gcov` file, we see we did not succeed. Either another option is necessary, or we have a feature that is not completely implemented! Oh well, let's try the next unexecuted line.

```
$ ./tests/unit_test_test --show-values
$ gcov -b -f *.c > output.log
$ vi unit_test_test.c.gcov
```

This time, it worked, and we see that line 162 has now been executed.

We can keep at it, trying more options until we have tried them all, and verified that every line of code in the test application has been touched.

Next, we need to check the coverage of the library modules.

```
$ cd src
$ gcov -o .libs/ *.c                          -or-
$ gcov -b -f -o .libs *.c > output.log
```

This results in `*.gcov` files not in `.libs`, but in `src`. Let's look at just one, `portable_subset.c.gcov`. It has a lot of unexecuted code, because some of the code is meant purely for error conditions that a good coder will avoid easily. We don't really care about testing this module, since it is meant only for internal usage [another library provides more advanced versions of the functions in the `portable_subset.c` module]. However, let's add a test of one of the functions to the `unit_test_test.c` module.

The function to be tested is `xpc_nullptr()`, and it has its own test-case group in the `unit_test_test.c` module, test group *07*. Here is a summary of what we did to rebuild:

```
$ vi tests/unit_test_test.c include/xcp/portable_subset.c
$ cd src
$ make clean
$ make
$ cd ../tests
$ make clean
$ make
$ cd ..
$ ./tests/unit_test_test --show-values
$ cd tests
$ gcov -b -f *.c > output.log
$ cd ../src
$ gcov -b -f -o .libs *.c > output.log
$ vi portable_subset.c.gcov
```

And you can now verify that `xpc_nullptr()` no longer is left out of the test run.

By the way, if you want to start from scratch, change to the project's root directory, run the following command, and repeat the steps above.

```
$ ./bootstrap --clean
```

There is a lot more that can be tested for coverage. You are welcome to do that, and tell me what I have missed!

8.4 The 'gprof' Profiling Application

gprof is another GNU application for analyzing code. It works at run-time, tabulating whether the program spends most of its time. Hence, it is useful for find the places where optimization (or even fixes or design changes) would do the most good.

Other potentially helpful packages are available:

- `qprof`. A set of profiling utilities for Linux.
- `kprof`. A profiling front-end for gprof, Function Check, and Palm OS Emulator.

8.4.1 Building For 'gprof' Usage

An application to be analyzed with *gprof* needs to be compiled in a special manner:

- Use a *gcc* compiler.
- Use the `-pg` option to link in the profiling libraries.
- Add `-g` for line-by-line profiling.
- Use `-finstrument-functions` if you have your own profiling hooks.
- Avoid optimization.

In the **XPCCUT** project, this is all done easily with these commands:

```
$ ./bootstrap --clean                     (if needed)
$ ./bootstrap --no-configure              -or-
$ ./bootstrap -nc
$ ./configure --enable-profiling --disable-shared
```

Note the `-enable-profile` option. It is implemented as part of the `m4/xpc_debug.m4` script, which is specified by the `AC_XPC_DEBUGGING` macro call in the **XPCCUT**'s `configure.ac` file.

Also note the use of the `-disable-shared` option, which is a standard `configure` option. If this flag is not provided to `configure`, attempt to run the **gprof** command on the executable will result in the following error message:

```
gprof: unit_test_test: not in executable format
```

Apparently, using shared libraries will mess up the profiler.

In addition to the normal arguments, you can specify the following forms:

- –enable-profiling=gprof (the default)
- –enable-profiling=prof

The first variant is the default, which specifies *GNU* **gprof** compatibility. The **gcc** flags for the **gprof** variant are:

```
-pg -O0 -ggdb
```

The second variant supports the *BSD* **prof** program. The **gcc** flags for the **prof** variant are:

```
-p -O0 -g
```

If you want to do code-coverage at the same time, you should be able to add the `-ftest-coverage` option. I have not yet tried this, though. Let me know how it works.

Once the configuration has been set up, run *make* in the `src` directory.

```
$ cd src
$ make
```

Then run it in the `tests` directory.

```
$ cd ../tests
$ make
```

8.4.2 Running for Profiling

Change to the application's directory (*unlike for gcov?*) and run the following commands.

```
$ ./unit_test_test
```

The result of this run is a file called `gmon.out` in the `tests` directory.

8.4.3 Running for 'gprof' Analysis

Once the test data are generated, with *gprof* information, they need to be analyzed using *gprof*.

Here are some commands to try. These commands can be done inside the proper directory (either `src` or `tests`, as appropriate.)

```
$ gprof ./unit_test_test
```

One can add the following options:

- The `-b` option for brief output.
- The `-A` option for annotated source code.

8.5 References

- http://gcc.gnu.org/onlinedocs/gcc.pdf GNU gcc manual
- http://gcc.gnu.org/onlinedocs/gcc/Gcov.html gcov – Test Coverage Program
- http://www.apress.com/book/view/9781590591093 (1st edition) and http://book.↩pdfchm.com/the-definitive-guide-to-gcc-second-edition-9569/ von Hagen, William (2006) *The Definitive Guide to GCC*, 2nd edition. Apress, Berkeley, California.

Chapter 9

Making a Debian Package

This file describes how to create a *Debian* package from an *automake*-supported project.

Table of Contents\n

9.1 Introduction

The **XPCCUT** library is a unit-test library for use in C programs. It also serves as the basis for a complementary C++ wrapper library. Its name stands for *Cross-Platform C, C Unit Test*.

The present page describes what we did to create a *Debian* package from this code.

This is our first try at making a Debian package. We welcome any ideas or criticism you have to offer.

9.2 Setup Steps

Here, we have to confess that the *Debian way* covers a lot of variations on a theme, and we are not sure that we've done everything in the proper manner.

The first thing to do is make sure your project is a **GNU autotools** project that builds and installs properly. This build setup can then be supplemented with **Debian** scripts located in a `debian` subdirectory of the project.

9.2.1 Initial Steps

1. Pick the conventions of your project carefully. All projects seem to have different layouts, Some layouts are not handled well by the debhelper tools, resulting in the need for some custom steps to be created.
2. Make sure the various automake targets work, as they are used by the debian scripts to perform these actions:
   - `make`
   - `make install`
3. Verify that the automake creates these items:
   - Directory `/usr/local/include/xpccut-1.1/xpc` (filled with header files)
   - Directory `/usr/local/share/doc/xpccut-1.1/html` (filled with documentation)
   - File `/usr/local/share/man/man1/xpccut.1`
   - File `/usr/local/share/locale/es/LC_MESSAGES/libxpccut.mo`
   - File `/usr/local/lib/libxpccut.a`
   - File `/usr/local/lib/libxpccut.la`
   - File `/usr/local/lib/libxpccut.so`
   - File `/usr/local/lib/pkgconfig/xpccut-1.1.pc`
4. `dh_make -c gpl -e ahlstromcj@gmail.com -n -l -p libxpccut`
5. Edit a bunch of files in the `debian` directory.

9.2.2 Steps for Testing the 'rules' File

1. Bootstrap the project (`./bootstrap`) in order to create the Makefiles, if not already present.
2. Run `fakeroot debian/rules clean` to verify that cleaning results in no errors.
3. Run `debian/rules build` to verify that the source code and documentation build without errors or missing results.
4. Run `fakeroot debian/rules binary` to very that all of the installation steps work. (However, the *fakeroot* man-page says not to do this step! Do it anyway.) See the note below in case this fails.
5. Examine the directories created in `debian/tmp`. They will mirror an actual installation.
6. Examine the `libxpccut1` and `libxpccut-dev` directories created in the `debian` directory. They will also mirror the actual install, though only libxpccut-dev will include the documentation and header-file directories.
7. Examine the `*.deb` files (see below) that are created. You can use *Midnight Commander* (mc) or *Krusader* to see inside them and verify the layout of the installations.

9.2.3 Handling Build Failures

Sometimes building the binary will fail the first time you run it on a host, with a message like this:

```
You needed to add /usr/lib to /etc/ld.so.conf.
Automake will try to fix it for you
/bin/sh: line 6: /etc/ld.so.conf: Permission denied

That didn't work.  Please add /usr/lib to
/etc/ld.so.conf and run ldconfig as root.
/bin/sh: line 13: ldconfig: command not found
```

Just do what it says and try again.

9.2.4 Doing a Complete Rebuild

1. Go to the root directory of the project.
2. Run `dpkg-buildpackage -rfakeroot`

This will create, in the parent directory above the project, the following files:

1. `libxpccut-dev_1.1.0_i386.deb`
2. `libxpccut1_1.1.0_i386.deb`
3. `libxpccut_1.1.0.dsc`
4. `libxpccut_1.1.0.tar.gz`
5. `libxpccut_1.1.0_i386.changes`

Currently, however, we get the following error:

```
dpkg-buildpackage: warning: Failed to sign .dsc and .changes file
```

Fixed yet? If not, tell us how!

9.3 References

We used the following references to figure out what to do, though a lot of backtracking, interpolating, and trial-and-error was necessary:

1. http://debathena.mit.edu/packaging/
2. http://www.debian-administration.org/articles/337
3. https://help.ubuntu.com/ubuntu/packagingguide/C/basic-scratch.html
4. http://www.zoxx.net/notes/index.php/2006/08/09/22-create-a-debian-package-with-dh_make-and-dpkg-buildpackage
5. http://www.mail-archive.com/debian-mentors@lists.debian.org/msg18893.html
6. http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html

Reference 5 was most helpful, while reference 4 goes through the whole process quickly, including how to make your own debian repository. Reference 6 is of great help in getting the conventions correct.

Chapter 10

Using Git

This document provides condensed procedures for using Git in various parts of the version-control workflow.

10.1 Introduction to Using Git

This document is a concise description of various processes using the *Git* distributed version control system (DV↩
CS).

10.2 Setup of Git

Like other version-control systems, Git has configuration items and work-flows that need to be tailored to your needs.

10.2.1 Installation of Git

Obviously, the easiest way to set up Git on Linux is to use your distro's package-management system.

Failing that, or if you want the very latest and greatest, you can get the source code and build it yourself:

`http://git-scm.com/downloads`

On Windows, that same URL will get you an installer executable. After that, you're on your own for monitoring and retrieving updates. Another option is to use the *msysGit* project: `http://msysgit.github.io/` Actually, that gets you the very same executable!

10.2.2 Git Configuration Files

There are three levels of configuration for Git. Each succeeding level overrides the previous level.

- `/etc/gitconfig`. The system-wide configuration. Using `git -system config` affects this file. On MSYS for Windows, the `msys/etc/gitconfig` file is used.
- `~/.gitconfig` (Linux) or `$HOME/.gitconfig` (Windows, where HOME is `C:/Users/$USER`. The user's pan project configuration. Using `git -global config` affects this file.
- `myproject/.git/config`. The project's own configuration.

10.2.3 Setup of Git Client

For our purposes, the command-line Git client is quite sufficient. Start with the `gitk` package if you are interested in a GUI.

The `git config –global ...` command will set up your preferences for ignoring certain files, for coloring, email, editors, diff programs, and more. Specific instances of this command are shown below.

1. *Setting up Git features*. This process includes setting up preferred handling for:
   (a) Whitespace.
   (b) Line-endings.
   (c) Differencing code.
   (d) Merging code.
   (e) Programmer's editor.
   (f) Aliases for Git commands.
   (g) Colors.
   (h) Template of commit message.
   (i) And much more.
2. *Setting up 'git-ignore'*. This process provides a list of files, file extensions, and directories, that you do not want Git to track, as well as subsets of those that you do want Git to track anyway.

10.2.4 Setting Up Git Features

There are additional configuration options. Rather than discuss them, we will just list the commands and show the `~/.gitconfig` file that results.

```
$ git config --global user.name "Chris Ahlstrom"
$ git config --global user.email ahlstrom@bogus.com
$ git config --global core.excludesfile ~/.gitignore
$ git config --global core.editor vim
$ git config --global commit.template $HOME/.gitmessage.txt
$ git config --global color.ui true
$ git config --global core.autocrlf input
$ git config --global diff.tool gvimdiff
$ git config --global merge.tool gvimdiff
$ git config --global alias.d difftool
```

`commit.template`:

```
subject line

what happened

[ticket: X]
```

Another option is `core.autocrlf` (what about `core.eol`?). If set to 'true', then it will convert LF into CR-LF line-endings. Useful for those unfortunate Windows-only teams. For Linux teams, or for mixed-OS teams, see the "input" setting above. That setting will leave you with CR-LF endings in Windows checkouts but LF endings on Mac and Linux systems and in the repository.

There are many more helpful configuration items, such as those that deal with OS-specific line-endings or the disposition of white space. Other "config" options to check out: `core.pager`; `core.whitespace`; `user.signingkey`; `color.*`; and, on the server side, `receive.fsckObject`, `receive.denyNonFastForwards`, and `receive.denyDeletes`.

After this, your `.gitconfig` should look like this:

```
   [user]
      name = Chris Ahlstrom
      email = ahlstrom@bogus.com
   [core]
      excludesfile = /home/ahlstrom/.gitignore
```

And this is the file that results:

```
[user]
   name = Chris Ahlstrom
   email = ahlstrom@bogus.com
[core]
   excludesfile = /home/ahlstrom/.gitignore
   editor = vim
[diff]
   tool = gvimdiff
[difftool]
   prompt = false
[alias]
   d = difftool
[color]
   ui = true
```

You can check your current settings with this command:

```
   $ git config --list
```

10.2.5 Setting Up the Git-Ignore File

Building code generates a lot of by-products that we don't want to end up in the repository. This can happen if the `git add` command is used on a directory (as opposed to a file).

First, look at the sample git-ignore file `contrib/git/dot-gitignore`. Verify that the file extensions all all to be ignored (not checked into source-code control), and that the list includes all such files you can conjur up in your imagination. Next, copy this file to the `.gitignore` file in your home directory.

Finally, run

```
   $  git config --global core.excludesfile ~/.gitignore
```

10.2.6 Setting Up Git Bash Features

It is very useful to have your command-line prompt change colors and show the Git branch that is active, whenever the current directory is being managed by Git.

Note these two files in the `contrib/git` directory in this project:

- `dot-bashrc-git`, to be carefully inserted into your `~/.bashrc` file.
- `dot-git-completion`, to be copied to `~/.git-completion`, which is sourced in that .bashrc fragment.

What these script fragments do is set up your bash prompt so that it will show the Git branch that is represented by the current directory, in your command-line prompt, if the directory is part of a Git project.

10.3 Setup of Git

There are three "locations" where a Git repository can be created:

- Your personal computer
- Your local network
- A remote network (on the Internet, hosted by a hosting site)

10.4 Setting Up a Git Repository on Your Personal Computer

A restrictred, but still useful use-case is to set up a Git repository on the same computer you use for writing and building code.

Assuming you have a nascent project directory in existence, and want to start tracking it in Git, you perform the following steps (under your normal login):

1. `cd` to the project's root directory. Make sure there are no files that you don't want to track, or that they are covered by your system or user git-ignore file.
2. Run the command `git init`. The result will be a hidden `.git` directory.
3. Now run `git add *` to add all of the files, including those in subdirectories, to the current Git project.
4. Run `git status` to verify that you have added only the files that you want to track.
5. Run a command like `git commit -m "Initial project revision."`

10.5 Setting Up a Git Repository on Your Personal Computer

If your personal machine is on a home/local network, you can clone your new Git repository on another home machine. The following command assumes you use SSH even at home to get access to other home computers.

```
$ git clone ssh://homeserver/pub/git/mls/xpc_suite-1.1.git
```

This repository will be referred to as a remote called "origin". This remote assumes that "homeserver" has the IP address it had when the clone was made. However, outside of your home network, you may have to access "homeserver" from a different IP address. For example, at home, "homeserver might be accessed as '192.168.↩ 111.18', the IP address of 'homeserver' in `/etc/hosts`, while outside your home network, it might be something like '77.77.77.77', accessed from the same client machine remotely as 'remoteserver'.

In this case, you need to add a remote name to supplement 'origin'. For example:

```
$ git remote add remoteaccess ssh://remoteserver/pub/git/mls/xpc_suite-1.1.git
```

So, while working on your home network, you push changes to the repository using:

```
$ git push
```

or

```
$ git push origin currentbranch
```

while away from home, you push changes to the repository using:

```
$ git push remoteaccess currentbranch
```

10.6 Setup of Git Remote Server

10.6.1 Git Remote Server at Home

Setting up a Git "server" really means just creating a remote repository that you can potentially share with other developers.

The repository can be accessed via the SSH protocol for those users that have accounts on the server. It can also be accessed by the Git protocol, but you'll want to run that protocol over SSH for security. A common way to access a remote Git repository is through the HTTP/HTTPS protocols.

10.6.2 Git Remote Server at GitHub

Let's assume that one has a body of code existing on one's laptop, but nowhere else, and that no Git archive for it already exists. Also, let's assume we want to first set up the archive at GitHub, and then clone it back to the laptop to serve as a remote workspace for the new GitHub project. Finally, let's assume you've already signed up for GitHub and know how to log onto that service.

The process here is simple:

1. Convert your local project into a local Git repository.
2. Create your empty remote repository on GitHub.
3. Add your GitHub repository as a remote for your local Git repository.
4. Push your master branch to the GitHub repository.

Convert local project to local Git repository.

1. Change to your projects source-code directory, at the top of the project tree.
2. Run the command `git init` there to create an empty git repository..
3. Run the command `git add .` to add the current directory, and all sub-directories, to the new repository.
4. Run the command `git status` if you want to verify that all desired files have been added to the repository.
5. Run the command `git commit -m "short message"` to add the current directory,

Create new repository at GitHub.

1. Sign up for a GitHub account, if needed, then log into it. Also be sure to add your computer's public key to GitHub, if you haven't already. (How to do that? We will document that later.)
2. On your home page, click on the "plus sign" at the upper right and select "New Repository".
3. Verify that you are the owner, then fill in the repository name and description.
4. Make sure it is check-marked as public (unless you want to hide your code and pay for that privilege). There's no need to initialize the repository with a README, since you are importing existing code. No need to deal with a `.gitignore` file or a license, if they already exist in your project.
5. Click the "Create Repository" button.
6. Once the repository is created, take note of the URLs for it:

- **HTTPS**. https://github.com/username/projectname.git
- **SSH**. git@:github.com:username/projectname.git

7. Add this new remote repository to your local copy and make it the official copy, called "origin"; `git remote add origin git@github.com:username/projectname.git`
8. Push your current code to this repository: `git push -u origin master`

The `-u` (or `-set-upstream`) option sets an upstream tracking reference, so that an argument-less `git pull` will pull from this reference, Once you push the code, you will see a message like:

```
Branch master set up to track remote branch master from origin.
```

Note that the URL for the home-page for the project is

```
https://github.com/username/projectname
```

This URL can be used when adding GitHub as a remote.

To clone the new GitHub repository to another computer:

```
git clone git@github.com:username/projectname.git
```

Note that some of this information is also available in Scott Chacon's book, "Pro Git", from Apress.

TODO: talk about pushing and pulling your own changes, and pulling changes made by people who have forked your project.

10.7 Git Basic Commands

We've already covered the `config` and `init` commands, and touched on a few other commands. In this section, we briefly describe the most common commands.

You can get information using `git help`. Adding the `-a` option lists the many command of git. Adding the `-g` option lists some concepts that can be presented. Try the "workflow" concept:

```
$ git help workflows
```

The result is a man page summarizing a workflow with Git.

There is also a nice man page, gittutorial(1).

10.7.1 git status

First, note that git commands can be entered in any directory or subdirectory of a project. Yet the command will still operate on all files and directories in the project, starting from the root directory of a project. If files are to be displayed, their paths are shown relative to the current working directory.

The `git status` command examines the files and determines if they are new, modified, or unchanged. If new or modified, they are shown in red. The command also determines if they have been added to the commit cache. If so, then they are shown in green.

10.7.2 git add

When files are shown as red in the `git status` command, that means they have been modified. If you think they are ready, you can use the `git add` command to add them to the commit cache; they will then show up as green in the `git status` listing.

You can add any or all modified files to the commit cache. If you add a file, and then modify it again, you will have to add it again.

Now, if some files are still red, but you want to commit the green files anyway, it is perfectly fine to do so.

10.7.3 git commit

This command takes the files in the git commit cache (they show up as green in `git status`) and commits them. The most common form of this command is:

```
$ git commit -m "This is a message about the commit".
```

Keep the message very short, around 40 characters. One can leave off the message, in which case one can add this message, plus a much longer description, in the text editor that one set up git to use.

10.7.4 git stash

This command stores your current local modifications and brings the files back to a clean working directory. Once finished, you can recall the stash and continue onward. Great for double-checking the older revision of the code.

10.7.5 git branch

Branching is a complex topic, and not every workflow can be documented. Here, we describe a straightforward workflow where there's a master branch, and occasionally one side-development branch that ultimately gets merged back to master. We will assume for now that no conflicts occur, and that branches occur serially (one developer, who finishes the branch and merges it back before going on to the next branch.)

10.7.5.1 Create a branch

Our repository is at a certain commit in `master` at the current latest commit in a repository. We will call this commit `cf384659`, after the hypothetical checksum of that commit.

The `HEAD` pointer points to `master`, which points to `cf384659`.

Let's create a branch called "feature":

```
$ git branch feature
```

Now, `feature` points to `cf384659`, just like `master` does. `HEAD` still points to `master`. This means that we are still working in `master`.

To switch to the "feature" branch so that we can work on it:

```
$ git checkout feature
```

Now `HEAD` points to `feature`.

There's a git trick that let's you create a new branch and check it out in one command:

```
$ git checkout -b 'feature'
```

Verify that you're in the new branch; the asterisk will point to the new branch:

```
$ git branch
* feature
  master
```

In this branch, let's edit a file or two and commit them.

```
$ vi file1.c file2.c
$ git commit -m "Added part 1 of feature."
```

Now there's a new commit just beyond `cf384659`, we'll call it `f1958ccc`.
`feature` now points to `f1958ccc`, and `HEAD` still points to `feature`.

We're not done with `feature`, but we suddenly realize we want to add something to `master`. Let's set `HEAD` back to `master`

```
$ git checkout master
```

That was fast! Now we can make some edits, commit them, and then go back to working on `feature`:

```
$ git checkout feature
```

Now, git won't let you change branches if files are uncommitted, to protect from losing changes. Before you change branches, you must either *commit* the changes or *stash* them.

After a number of edits and commits on the "feature" branch, we're ready to tag it for ease of recovery later, and then merge "feature" back into "master".

10.7.6 git ls-files

This command is convenient for finding files, especially ones that did not get added to the archive because they were unintentionally present in a `.gitignore` file. The following command shows those "other" files:

```
$ git ls-files -o
```

10.7.7 git merge

TODO

10.7.8 git push

TODO

10.7.9 git fetch

TODO

10.7.10 git pull

TODO

10.7.11 git amend

TODO

10.7.12 git svn

TODO

10.7.13 git tag

TODO

10.7.14 git rebase

TODO

10.7.15 git reset

TODO

10.7.16 git format-patch

TODO

10.7.17 git log

TODO

10.7.18 git diff

TODO

10.7.19 git show

TODO

10.7.20 git grep

TODO

10.8 Git Workflow

TODO

10.9 Git Tips

```
git status
git status -sb
git checkout -b
git log --pretty=format:'%C(yellow)%h %C(blue)%ad%C(red)%d
   %C(reset)%s%C(green) [%cn]' --decorate --date=short
git commit --amend -C HEAD
git rebase -i
```

Also see the section on git_svn_page for using the Git-to-Subversion bridge.

Chapter 11

XPC Suite with Mingw, Windows, and pthreads-w32

This file provides information about building XPC in the Mingw environment, along with how to use pthreads in Mingw and Windows.

11.1 Introduction

The **XPC** library suite can be built in **Windows** as well as **Linux**. But rather than use **Microsoft**'s *Visual Studio* to build it for **Windows**, we will use the *Mingw* project to support it.

And, at first, we will build the **Windows** support from with **Linux** (**Debian** and **Gentoo**), rather than in **Windows** itself.

(We will probably add a `vs2010` directory to hold a solution and a number of `*.vcxproj` files, but it is a low priority.)

11.2 Installing Mingw

Mingw and its related tools can be built from source code in various environments, then be installed. This build is a lot of work.

However, prebuilt packages exists for most environments, and that is what we'll use.

11.2.1 Installing Mingw in Debian

In **Debian GNU**/**Linux**, a number of packages exist to support *Mingw*:

- gcc-mingw32
- mingw-w64
- mingw32-binutils
- mingw32-runtime

To build **XPC** for *mingw*, try these commands for configuration:

```
$ ./configure --host=i586-mingw32msvc --with-mingw32=/usr/i586-mingw32msvc/
$ ./configure --host=amd64-mingw32msvc --with-mingw32=/usr/amd64-mingw32msvc/
```

11.2.2 Installing Mingw in Gentoo

Gentoo provides a script called *crossdev* that knows how to configure and build the various parts of *mingw*:

- sys-devel/crossdev. This package builds and installs
  - dev-util/mingw-runtime
  - dev-util/mingw64-runtime
  - dev-util/w32api

If you use *layman* on **Gentoo**, be sure to check out potential issues with it:

http://www.gentoo-wiki.info/MinGW

You might want to add this line to `/etc/portage/package.use`:

```
cross-i686-pc-mingw32/gcc doc gcj multilib nptl openmp libffi -gtk
      -mudflap -objc -objc++ -objc-gc -selinux
```

Supported architectures:

- `i686-pc-linux-gnu`: The default x86 tuple for PCs.
- `x86_64-pc-linux-gnu`: The default tuple for 64-bit x86 machines (such as the AMD 64 and IA64 architectures).
- `powerpc-unknown-linux-gnu`: Support for PowerPC PCs, e.g. Apple Macintoshes.
- `arm-unknown-linux-gnu`: Support for embedded devices based on ARM chips.
- `arm-softfloat-elf`: For embedded devices with ARM chips without hardware floating point.
- `arm-elf`: For embedded devices with ARM chips with hardware floating point.
- `i686-pc-mingw32`: Supports cross-compiling for 32-bit Windows (toolchain based on mingw32).
- `i686-w64-mingw32`: Supports cross-compiling for 32-bit Windows (toolchain based on mingw64).
- `x86_64-w64-mingw32`: Supports cross-compiling for 64-bit Windows (toolchain based on mingw64).
- `avr`: Supports cross-compiling for Atmel AVR-MCU's.

As `root`, run

```
# emerge crossdev
```

Decide the target for which you want to build. Also decide where the cross-compiler is to be installed. Then run a command like the following, adjusting for your choices:

```
# crossdev --target i686-pc-mingw32 --ov-output /usr/local/portage
```

This action will cause about 6 packages to be built. We had to run it twice; the first time this warning came up:

```
!!! WARNING - Cannot auto-configure CHOST i686-pc-mingw32
!!! You should edit /usr/i686-pc-mingw32/etc/portage/make.conf
!!! by hand to complete your configuration
 * Log: /var/log/portage/cross-i686-pc-mingw32-binutils.log
 * Emerging cross-binutils ...
```

You can also run *crossdev* with the `-help` option to see what else you can do.

Here's what was installed on our **Gentoo** system in /usr/local/portage/cross-i686-pc-mingw32:

```
binutils -> /usr/portage/sys-devel/binutils
gcc -> /usr/portage/sys-devel/gcc
gdb -> /usr/portage/sys-devel/gdb
insight -> /usr/portage/dev-util/insight
mingw-runtime -> /usr/portage/dev-util/mingw-runtime
w32api -> /usr/portage/dev-util/w32api
```

To build **XPC** for *mingw*, try this command for configuration:

```
$ ./configure --host=i686-mingw32msvc --with-mingw32=/usr/i686-mingw32msvc/
```

11.2.3 Installing Mingw in Windows

We want install both *mingw* and *MSYS*.

TODO TODO TODO TODO TODO

11.3 Installing Pthreads

This section does not talk about installing the normal **Linux** *pthreads* package. It talks about installing *pthreads-win32* in **Windows** and in **Linux**.

11.3.1 Installing pthreads in Debian

On our **Debian** *squeeze* machine, *mingw* is installed in two locations:

```
/usr/amd64-mingw32msvc
/usr/i586-mingw32msvc
```

11.3.2 Installing pthreads in Gentoo

11.3.3 Installing pthreads in Windows

11.4 References

1. http://en.gentoo-wiki.com/wiki/Crossdev
2. http://www.gentoo.org/proj/en/base/embedded/handbook/
3. http://www.gentoo.org/proj/en/base/embedded/handbook/cross-compiler.xml
4. http://metastatic.org/text/libtool.html
5. http://dev.gentoo.org/~vapier/CROSS-COMPILE-HOWTO
6. http://en.gentoo-wiki.com/wiki/Mingw
7. http://www.gentoo-wiki.info/MinGW
8. http://wiki.njh.eu/Cross_Compiling_for_Win32

Chapter 12

Nice Regular Classes and Coding Conventions

This file provides a collection of advice from various sources about the creation of good **C++** classes and functions. This document describes the coding conventions for a *nice* and *regular class*.

12.1 Introduction

This section provides a fast tutorial on writing class functions and friendly global functions. In addition, it illustrates coding conventions pretty similar to what one might like to use in everyday code.

It declares all the items of a "nice" class and a "regular" class. Demonstrates the basics of some useful techniques for improving the consistency and maintainability of a class.

For this discussion, assume that all the following occur as inline functions in the class declaration

```
class X : public B {};
```

Let class **T** represent a built-in type; let **B** and **Y** represent any other type. Let instances of each class be denoted by the same letter, in lower case.

The following code fragments represent some of the more important constructors, operators, and destructors that can be written, and the things needing to be done by each of them for efficiency, safety, and maintainability.

12.2 Background

Margaret Ellis and her colleagues group the functions in the categories of "regular" and "nice". No, "regular" is *not* what's normal for you. These two terms summarize classes that have behavior consistent with that of the built-in classes. There should be few surprises in the usage of a regular or nice class.

12.3 Characteristics of Nice and Regular Classes

Regular classes provide the following functions:

```
copy constructor           Construct an X whose value is the same as x
destructor                 Destroy this X object
principal assignment op    Set this object to x & return a reference to it
equality op                Return true if & only if x1 & x2 have same value
inequality op              Return true if & only if x1 & x2 not the same
```

Nice classes provide the following functions:

```
default constructor        Construct an object with a null value
copy constructor           Construct an X whose value is the same as x
destructor                 Destroy this X object
assignment op              Set this object to x and return a reference to it
equality op                Return true if & only if x1 and x2 have same value
```

As noted in http://en.wikibooks.org/wiki/C%2B%2B_Programming/Classes/Nice_↩Class, "nice" classes can be called "container-safe" classes.

Note that we provide a relatively simple class declaration, followed by a well-documented inline definition for each function. This should give you quite a jump start on your own nice classes. Thanks to the books of Bjarne Stroustrop, Scott Meyers, and Stan Lippman, as well as code by Paul Pedriana, and an article by Ellis.

12.4 Semantics of Copying and Assignment

Here, we discuss the copy constructor, the principal assignment operator (PAO), a copy() function, and a clone() function. The copy() and clone() functions are useful in containing common code used in the writing of the copy constructor and principle assignment operator.

Item	copy constructor	principle assignment	copy()	clone()
Destination exists	no	yes	yes	no
Need to delete old items	no	yes	yes	no
Need to create new items	yes	yes	yes	yes

Note the need for creation and deletion. This means you should also write a pair of functions such as "allocate()" and "deallocate()" or "create()" and "destroy()". Often, all of these functions are also useful in the other constructors or in the destructor. See, for example, Principal assignment operator.

You might call this "the Tao of PAO". In any case, GUI frameworks like MFC and OWL often hide the copy constructor and assignment operator because writing them correctly can be hard work!

12.5 Return-Value Optimization

"Return value optimization" (RVO) is a technique for avoiding temporary variables created by the compiler to return results to the caller of a function.

See http://www.cs.cmu.edu/~gilpin/c++/performance.html#returnvalue for a discussion of one form of RVO. This isn't what is normally meant by RVO, though.

It involves the difference between these two calls:

```
const T operator + (const T & lhs, const T & rhs)
{
   T result(lhs);             // copy lhs into result
   return result += rhs;      // add rhs to it and return it
}
```

versus

```
const T operator + (const T & lhs, const T & rhs)
{
   return T(lhs) += rhs;      // return value optimization
}
```

Here may be a good link for general C++ optimization issues:

http://www.tantalon.com/pete/cppopt/main.htm

Another, more esoteric, link: http://blogs.msdn.com/slippman/archive/2004/02/03/66739.aspx This article provides an in-depth discussion of "named" RVO, and how the two most widely used compilers, Visual C++ and GNU C++, did not implement it for a long while.

12.6 Coding Conventions

Lots of lessons packed into this module:

1. Use the correct header style on all modules, including the project files. See the top of this module for one example.
2. Avoid tab characters, but set the tab stops of your editor to increments of 3 or 4. Some projects use 8. We use 3. Set up your editor or IDE right now!
3. Use white space wisely and to good effect. Surround operators with a space when possible.
4. Use the carriage return often to increase readability and keep the lines no more than 80 characters. (You be glad you did when your company hands you a laptop or a PDA with a too-small screen!)
5. Comments.
   (a) Use short side comments when possible, and line them up at the same tab stop.
   (b) We have two ways to comment lines of code. Pick whichever works best for the kind of code your writing.
      i. // Put the comment right above code. Good with one-liners. category = NICE_CLASS;
      ii. // Embed the comment as a paragraph before a number of lines of code. This style is good when you want to explain a number of lines of code using a single, long comment. Extra blank comment lines can be added if desired.

```
category = NICE_CLASS;
for (int ci = 0; ci < category; ci++)
{
      much_code();      // a routine thing to do!
}
```

(c) Comment on and explain all parameters and return values. Don't allow the reader to make incorrect guesses as to what the parameters means.

6. Put the return type of functions on a line by itself. This habit makes it easier to find declarations with a real programmer's editor (one that can search for the beginning of a line), and also makes long declarations easier to read.

7. Naming conventions. This area seems to garner the most controversy, so the following items are just suggestions.

   (a) Try to use lower case for local variables.

   (b) Avoid use of warts unless it is deeply ingrained in your psyche. Some common warts:

   - sz string terminated by an ASCII zero
   - lpsz long pointer to string terminated by an ASCII zero
   - b boolean variable
   - i integer variable – holy FORTRAN!
   - m_ a member variable of a Microsoft class
   - _ Appended or prepended to certain members

   (c) Use mixed case for members of a class, or whereever it will distinguish variables with good effect.

8. Adopt a consistent and clean function indentation style. Here are two. The first one makes it easy to comment on the parameters, and is perhaps easier to read.

```
static int
integers_to_ascii_array
(
   int firstone,
   int secondone,
      ...
)
{
   ... define the function ...
}
```

All statement blocks should be indented in this manner, too.

```
static int integers_to_ascii_array (
   int firstone,
   int secondone, ... )
{
   ... define the function ...
}
```

12.7 Error Handling

There are two main camps on the topic of error handling. The first camp believes in using and checking return values. The second camp believe in using exception handling. Return values require great diligence in the codernaught, so that all values get checked. Exception handling requires great diligence in making constructors exception-safe (no leaks), and exceptions can generate some unwanted overhead.

Should you have a single error-handling convention? I don't know. You should probably use both methods, depending on the module involved, but use except only for the more arduous errors.

12.8 A Nice Pseudo-Class

This sample class should give you an idea of a nice class.

12.9 Declarations

```
#if !defined NiceClassCode_h
#define NiceClassCode_h

#include "U_class.h"               // U, a user-written or library class

typedef int B;                     // B is a built-in type (e.g. integer)
typedef int T;                     // T is any type; includes U or B

/*********************************************************************
// Class member versions of declarations
//
// Note the "friend" function.  Friend functions are a good way to
// extend the interface of a class without adding to the complexity of
// the class.  Also, overloaded operator functions need to be friends.
//
// The rest of this class declaration shows the best function
// signatures for various member functions.  The functions themselves
// are fleshed out and described below.
//
*********************************************************************/

class X : public U                 // note the base class, U
{
   friend const X operator + (const X & x1, const X & x2);   // binary operator

public:

   X ();                           // default constructor
   X (T t = 0);                    // default constructor (default parameter)
   X (const X & x);                // copy constructor
   operator C ();                  // conversion operator (inherited) named C
   operator C () const;            // conversion operator (safer)
   X & operator = (const X & x);   // principal assignment operator
   virtual ~X ();                  // destructor (virtual for base classes)

   bool operator < (const X & x) const;    // less-than operator
   bool operator > (const X & x) const;    // greater-than operator
   bool operator == (const X & x) const;   // equality operator
   bool operator != (const X & x) const;   // inequality operator
   bool operator <= (const X & x) const;   // less-than-or-equal-to operator
   bool operator >= (const X & x) const;   // greater-than-or-equal-to operator

   int operator ! ();                  // unary operator
   X & operator += (const X & x);      // assignment version of operator
   const X operator + (const X & x);   // member version of binary operator
   X & operator ++ ();                 // prefix increment operator
   const X operator ++ (int);          // postfix increment operator
   X & operator [] (T index) const;    // read subscript (must be non-static)
   X & operator [] (T index);          // write subscript (must be non-static)
   X operator () () const;             // parentheses operator
   void operator () ();                // parentheses operator
   Y * operator -> () const;           // indirection (dereference) operator
   Y & operator * () const;            // indirection operator

private:

   B built_in_type_member;
   T another_type_member;

};

/*********************************************************************
```

```
// Global versions of declarations
//
// Any of the following functions that need to be implemented in
// terms of private members of class X must be declared friends of
// class X, as shown above for
//
//       const X operator + (const X & x1, const X & x2);
//
// Also note that there is a member version of operator +() declared
// above.   See the definition's comments way below for the reasoning.
//
/*********************************************************************

bool operator < (const X & x1, const X & x2) const;   // less-than operator
bool operator > (const X & x1, const X & x2) const;   // greater-than operator
bool operator == (const X & x1, const X & x2);        // equality operator
bool operator != (const X & x1, const X & x2);        // inequality operator
bool operator <= (const X & x1, const X & x2) const;  // less-than-or-equal operator
bool operator >= (const X & x1, const X & x2) const;  // greater-than-or-equal operator

int operator ! (X & x);                               // unary operator

const X operator + (const X & x1, const X & x2);      // binary operator
const X operator + (const X & x1, const T t);         // overloading on Tr
const X operator + (const T t, const X & x);          // commutativity
```

12.10 Definitions

This section briefly discusses the implementation of class members.

12.10.1 Default constructor

The default constructor creates and initializes all of the resources a class will need (though with default values, which have the option of disabling the class). It can often be made more robust with a reusable "create()" or "allocate()" function as discussed in Semantics of Copying and Assignment.

1. Be sure to include the member initializations in the order of their declaration in the class. The compiler will call them in that same order, and you won't get confused in the debugger.
2. Be very sure to make any constructed objects exception-safe (e.g. by using the auto_ptr template class). See "More Effective C++" for details. This safety requires great care and a lot of work.
3. Prefer initialization in the initializer list to assignment in the body of the constructor; it's more efficient and often more clear.
4. Avoid writing code that calls the default constructor and follows the call with a bunch of member initializations using public accessor functions.
   (a) Exposing the members publicly is dangerous.
   (b) The code is less clear and definitely less efficient. The caller is forced to maintain copies of member values to use for the initialization. [Microsoft's MFC violates this advice a lot; Borland's OWL does a better job.]
5. Remember that some container classes require that a default constructor be supplied. If the compiler's version won't do the right thing, then you must provide this function.

Here is a skeleton of a default constructor.

```
inline
X::X ()
 :
   U(...),        // base class construction
   member(...)    // member initializations
{
   // Other code; Microsoft and other software often assigns to members
   // here, unfortunately.  This is redundant and less readable.
   //
   // Don't forget about the issue of <i>exception safety</i>.
   // Googoo for...  I mean Google for it.
   //
   // If a create() or allocated() function written for reuse exists, it may
   // be possible to call it here.
}
```

12.10.2 Principal constructor

The term *principal constructor* is simply our term for the main constructor of a class. This function provides one or more non-default parameters, and is the constructor most likely to be called.

This term is analogous to **principal assignment operator** , and was coined merely to distinguish the main constructor from the other ones.

12.10.3 Conversion via default constructor using default parameter

```
inline
X::X (T t = 0)
 :
   U( ... ),      // base class construction
                  // member initializations
{
   // other code
}
```

12.10.4 Copy constructor guidelines

```
  -#  Consider implementing a reference-counting scheme.
  -#  The copy constructor is a regular and nice function.
  -#  Another valid form is X::X(const X & x, T t = 0)
```

Note that this does not cover the issue of copying the base class portion properly. Not sure what to do, but see the principal assignment operator for an idea, using the = operator. Also see Stroustrop's discussion of clone() or copy() functions.

It might be that the base classes appear first in the member initialization list, as in:

```
      U(x)
```

which should initialize the U part of the new X with the U part of the x parameter.

Note that we have some ways of detecting an error in a constructor call:

1. Checking for a null pointer or other badly-initialized member.
2. Setting an "is-error" member, providing a public function to check it, and actually calling it after construction.

3. Catching a thrown exception.

```
inline
X::X (const X & x)
 :
   U( ... ),      // base class construction
   each_member(x.each_member),
   .  .  .
{
   // Let xm be a pointer allocated in the constructor.  Then the following
   // stylized code could be used [many ways of copying can be employed]:

   xm = new (nothrow) Y;      // allocate space for the object
   if (xm != 0)               // make sure the allocation worked first
      *xm = *x.xm;            // copy the object
}
```

12.10.5 Copy constructor (another version)

```
  -# Found this alternate method in Borland's OwlSock code.  Ye gods!
  -# Note the use of operator =().
  -# This implementation mixes up the semantics of the copy
     constructor and the principal assignment operator.  The
     copy semantics are something like
     -# Create the object and its members.
     -# Allocate any memory, if applicable.
     -# Copy from the members of the source class, and, if applicable
        from the memory allocated in the source class.
  -# The assignment semantics are different, because the destination
     already exists:
     -# Make sure the source and destination are not the same.  If the
        same, skip to step e.
     -# Delete any allocated memory, if applicable.
     -# Re-allocate any new memory, if applicable.
     -# Copy from the members of the source class, and, if applicable
        from the memory allocated in the source class.
     -# Return "*this".
  -# Do not write functions like those below.


inline
X::X (const X & x)
{
   operator =(x);            // bad semantics, and should cast to (void)
}

inline
X::X (const T & t)
{
   operator =(t);            // bad semantics, and should cast to (void)
}
```

12.10.6 Principal assignment operator

The principal assignment operator (operator =) is a great example of a function that checks its parameters, deallocates existing data, creates a new area for it, copies data, and then returns a reference to itself. It uses the functions mention in Semantics of Copying and Assignment – copy(), clone(), allocate()/deallocate() or create()/destroy(). Thus, it provides many opportunities for re-use and robustness.

1. Must be a member function, to avoid absurdities.
2. Must return a reference to X, to permit the chaining of assignments. [Scott Ladd's old book erroneously declared this operator as "void".]

3. Check to be sure that assignment to self is not done. Might need to implement an identity scheme, especially when multiple inheritance is involved, since there's no guarantee that the address applies to the part of the object desired. Here are the easy, but sometimes insufficient, alternatives:

```
if (this != &x)            check pointer values [a fast method]
if (*this != x)            check object value
```

4. Not an inherited function. Derived class's must provide there own version. The version provided by the compiler assigns only the members of the derived class.

5. Delete any existing resources in the object, if necessary.

6. Note how the base class's operator can be called to increase the consistency of the code, but only if the operator has been explicitly declared. Otherwise, a cast to a reference can be used. Both alternatives are shown below.

7. Allocate any necessary resources and copy the values from the source. A copy() function can contain code common to both the copy constructor and the principla assignment operator. See Stroustrop for advice.

8. Assign the derived-class members.

9. Return ∗this, always.

```
inline X &
X::operator = (const X & x)
{
   if (this != &x)               //  make sure not assigning to self
   {
      // Let xm be a pointer allocated in the constructor.  Then the
      // following stylized code could be used:

      if (xm != 0)               // current object already have one?
         delete xm;              // yes, delete it (might need [] in some cases)

#if base class U explicitly declares an assignment operator
         U::operator =(x);       // copy the U portion
#else
         ((U &) *this) = x;      // copy the U portion
#endif

      each_member = x.each_member;  // copy the members

      xm = new Y;                // allocate space for the new object
      *xm = *x.xm;               // copy the object
   }
   return *this;                 // take note of this!
}
```

12.10.7 Destructor

```
  -# Be sure to make the destructors of base classes virtual.  In this
     way, the base class destructor will be called when the derived
     class's destructor finishes.
  -# Even if a virtual destructor is pure, a definition must be
     coded, since a base class's virtual destructor will be called
     when the derived class's destructor finishes.
  -# Do not define a destructor unless one is necessary.  Otherwise
     code is wasted, and the reader might be confused as to why
     a do-nothing destructor is defined.
  -# "More Effective C++" discusses a method of controlling the
     propagation of exceptions, as hinted at in the code below.


inline
X::~X ()
{
   try
```

```
   {
      // code that might cause exceptions
   }
   catch (...)          // catch any exception
   {
      // do nothing; we just want to stop the exception from propagating
      // further
   }
}
```

12.10.8 Conversion operator (an inheritable function)

```
  -# The conversion operator is useful with smart pointers.
  -# Converts from a class to a basic type (e.g. a built-in type
     or a structure).
  -# By convention, this kind of function has no explicit return type.
     The return type is implicitly the name of the operator.
  -# Warning:  The compiler can call the conversion implicitly.
     Meyers recommends eschewing this operator.  Instead,
     replace the built-in type T with a function name that has
     less syntactic magic (e.g. "AsDouble" versus "double").
     The keyword "explicit" can be used to avoid unintended
     implicit conversions in compilers that are up to the latest
     C++ specifications.
  -# This function is inherited by derived classes.
  -# Note the safer "const" version.
  -# It might be better to reinterpret_cast<T *>(this) below,
```

```
inline
X::operator T ()
{
   return *((T *) this);
}

inline
X::operator T () const               // the safer version
{
   return *((T *) this);
}
```

12.10.9 Equality operator

```
  -# This is a regular and nice function.
  -# Use this operator to implement the operator !=().
```

```
inline bool
X::operator == (const X & x) const
{
   bool result = false;

   // implement the desired definition of equality.  For example...

   if (member == x.member)
      result = true;

   return result;
}
```

12.10.10 Inequality operator

```
  -# This is a regular and nice function.
  -# Note how we use operator ==() to implement this operator.
     This improves the correctability/maintainability of the code.


inline bool
X::operator != (const X & x) const
{
   // implemented in terms of operator ==

   return (*this == x) ? false : true ;
}
```

12.10.11 Less-than operator

For containers, this is an important operator, because it not only defines the less-than operation, but containers defined the equality, inequality, greater-than, greater-than-or-equal, and less-than-or-equal functions in terms of less-than.

To make our comparison operators consistent with each other, following the discussion of Stroustrop, 3rd edition, section 17.1.4.1, we can define equivalence (equiv) in terms of our less-than comparison operator (cmp):

```
  equiv(x, y) = not [ cmp(x, y) or cmp(y, x) ]
```

It is easy to invert these to get the definition of operator !=() We don't have to worry, we just have to define operator $<$.

Note here that we follow the advice of Scott Meyers, and write the comparison operators as member functions.

```
inline bool
X::operator < (const X & x) const
{
   return key < x.key;           // more complex for complex classes
}
```

If there are two members, *a* and *b*, where *a* takes predence in the less-than calculation, then do the checks in this order:

```
inline bool
X::operator < (const X & x) const
{
   if (a < x.a)
      return true;
   else if (a == x.a)
      return b < x.b;
   else
      return false;
}
```

Given the less-than operator, we can write the equivalence operator in terms of it. Normally, the STL container classes do this for you, but here is a good definition:

```
inline bool
X::operator == (const X & x) const
{
   return ! (x < *this || *this < x);
}
```

The inequality operator is trivially implemented in terms of operator $<$:

```
inline bool
X::operator != (const X & x) const
{
   return ! (x < *this || *this < x);
}
```

The greater-than operator is even more trivially implemented in terms of operator $<$:

```
inline bool
X::operator > (const X & x) const
{
   return x < *this;
}
```

The less-than-or-equal operator is trivially implemented in terms of operator $<$:

```
inline bool
X::operator <= (const X & x) const
{
   return ! (x < *this);
}
```

The greater-than-or-equal operator is trivially implemented in terms of operator $<$:

```
inline bool
X::operator >= (const X & x) const
{
   return ! (*this < x);
}
```

12.10.12 Unary operator

```
inline int
X::operator ! ()
{
   // LATER, I'm getting tired
}
```

12.10.13 Assignment version of binary operator

```
  -# Also applies to -=, /=, *=, etc.
  -# When writing the other operators based on this operator [that
     is, operator +() and operator ++(), prefix and postfix
     versions], implement them in terms of this operator, as shown
     later, to improve the consistency of the code.
  -# When using the operators, note that the assignment version
     of a binary operator is more efficient than the binary
     operator (the "stand-alone" version of the operator).


inline X &
X::operator += (const X & x)
{
   X result;

   // implement the desired addition operation

   return result;
}
```

12.10.14 Prefix increment operator

```
  -# Applies to operator --() also.
  -# Note that this implementation uses operator +=() for
     consistency.
  -# Use the prefix version to implement the postfix version
     (shown below).
  -# The prefix operator is more efficient than the postfix operator.


inline X &
X::operator ++ ()
{
   // One way to increment (if overloaded on a built-in integral type):

   *this += 1;                          // (other forms possible)

   return *this;
}
```

12.10.15 Postfix increment operator

```
  -# Applies to operator --() also.
  -# Note how we use the prefix version to implement this version to
     promote consistency and maintainability.
  -# The prefix operator is more efficient than this operator.
  -# Note the "const" in the return value.  This prevents the
     programmer from trying something like x++++, which is illegal
     for integers, and would increment x only once anyway.
  -# The preceding precept illustrates that adage "when in doubt,
     do as the ints do."


inline const X
X::operator ++ (int)
{
   X oldvalue = *this;               // save it for the return
   ++(*this);                        // use prefix increment
   return oldvalue;
}
```

12.10.16 Subscript operator

```
  -# Must be a non-static member function.
  -# Note the two versions, the const one being for reads.  The
     2nd edition of Scott Meyer's book makes the return value of
     the read version a const.
  -# Designing a good subscript operator is a deep subject.  Refer
     to Meyers' "More Effective C++".
  -# Rather than making your own array class, consider using the
     STL vector class instead.


inline const X &
X::operator [] (T index) const
{
   // access code here
}

inline X &
X::operator [] (T index)
{
   // access code here
}
```

12.10.17 Parenthesis operator

The parentheses operator is also called the "function call" operator. It must be a member function.

1. Useful for making function objects.
2. Useful for obtaining substrings from a string class.
3. Useful as a subscripting operator for multi-dimensional arrays.
4. Again, a deep topic; see Meyers and see Stroustrop.

```
inline X
X::operator (T & t) () const
{
   // The t parameter is the same type T as X::t_member.  This particular
   // parentheses operator would be called as X(t), and would cause the
   // parametr to be modified. An example:

   t += t_member;

}
```

12.10.18 Indirection (field dereference) operator

```
  -# Useful in dereferencing a smart pointer to get access to one
     of its member functions.
```

```
inline Y *
X::operator -> () const
{
   // code needed here
}
```

12.10.19 Indirection (object dereference) operator

```
  -# Useful in dereferencing a smart pointer.
```

```
inline Y &
X::operator * () const
{
   // code needed here
}
```

12.11 Global versions

Some of the member functions above can also be written as global functions. Sometimes this way of writing the function has advantages, such as (surprise!) improved encapsulation.

Many of the same comments and format as for their class-member brethren apply, and are not repeated here.

12.11.1 Equality operator

```
inline bool
operator == (const X & x1, const X & x2)
{
   bool result;

   // Implement the desired definition of equality or write in terms of
   // operator <.

   return result;
}
```

12.11.2 Inequality operator

```
inline bool
operator != (const X & x1, const X & x2)
{
   return !(x1 == x2);
}
```

12.11.3 Unary operator

```
inline int
operator ! (X & x)
{
   // your code needed here
}
```

12.11.4 Binary operator (member version and global versions)

```
  -# The global version is preferable to the member version because
     conversions can generally be made that support commutativity.
  -# Note the implementation in terms of the class version of
     operator +=().  For example, assume that there exists a conversion
     from an integer to a class X.  Then, given the declaration


   X x(1);              // create an X instance


     only one of the following is legal if operator +() is a member of
     X:


   X result = x + 2;    // fine, calls x.operator+(2)
   X result = 2 + x;    // fails, there is no 2.operator+(const X &)


  -# Stroustrop prefers to implement operator +=() as a member of
     the class X, then to implement operator +() as a global function,
     as shown in the second version.
  -# Note the usage of the copy constructor in the return-value-
     optimizable version.
  -# By not creating any named automatic (temporary) variables,
     we give the compiler a chance for return-value optimization.
     This could save constructor calls.
  -# The const return value avoids code such as x + + y.
  -# This function can be made efficient by keeping it defined
     as an inline function.
  -# Also applies to -, /, and *, etc.
  -# Never perform this kind of overloading for operator &&(),
```

```
     operator ||(), or operator ,() [the infamous comma operator].
     For && and ||, short-circuit evaluation is not used in the
     overloaded versions, and this could cause confusion.   For
     the comma operator, the behavior of "evaluate left expression
     first, then right expression, and return the right expression"
     cannot be mimicked in an overload.
  -# const X operator + (B b, B b) is illegal, because it would
     change the built-in operator.  [Recall that we defined B as
     a built-in type].
  -# Meyers uses "lhs" and "rhs" (left- and right-hand side variable)
     in place of "a" and "x", respectively).


inline const X
X::operator + (const X & x)               // first version
{
   return operator +=(x);                 // calls this->operator+()
}

inline const X
operator + (const X & a, const X & x)     // second version, preferred
{
   X result = a;
   return a += x;                         // calls a.operator+()
}

inline const X
operator + (const X & a, const X & x)     // return-value-optimizable version
{
   return X(a) += x;
}
```

12.11.5 Overloading on T and Commutativity

```
inline const X
operator + (const X & x1, const T t)
{
   // code needed here
}

inline const X
operator + (const T t, const X & x)
{
   // code needed here
}

#endif      NiceClassCode_h
```

Chapter 13

Philosophy, XPC Library Suite

This file describes a little bit about our coding style and philosophy.

13.1 Philosophy of the XPC Library Suite

This document is a start at explaining my own philosophy about code development.
This document is incomplete. At first, we just want to explain some coding principles in the **C** and **C++** code that some may find odd, incorrect, or substandard in what we do.
Finally, note how much this document parallels the "license" document, Licenses, XPC Library Suite. What a coincidence!

13.2 XPC Application Philosophy

For the moment, see XPC Library Philosophy instead. All of the applications in this suite are console applications that provide a large number of command-line arguments.

13.3 XPC Library Philosophy

My overriding philosophy about libraries is that they should:

- Be as simple as possible.
- Be as flexible and unrestricting as possible.
- Be as airtight, logical, and robust as possible.
- Be able to provide useful diagnostic output when necessary.
- Not duplicate functionality that is better provided by existing libraries, while ...
- ... Conforming to my own quixotic ways of doing things.

A good idea is to contrast my (or your) coding philosophy with the GNU coding philosophy. From my vantage, the GNU coding philosophy:

- Supports freedom and access.
- Avoids needless restrictions in implementation, and takes great pains to avoid hard-wiring.
- Has its own quirks in coding style and implementations that some (including myself) may not relish.

The present document is a work in progress. We don't want to bore anyone, but we do have a few oddities to explain in the next sections, just so you don't think we are stupid.

13.3.1 XPC's Usage of the Shell

There are a few things "wrong" with our shell-scripting.
First, we use *bash*. While this is nice for **Linux** users, it is not so nice for other environments, which may not have *bash* installed, and use a shell that does not support all the *bash* syntax that we use.
Second, we do our **Boolean** flags in a stilted manner. We define a flag as being off by setting it to "no". We test a flag by testing it against "yes" or "no".

```
MYFLAG="no"
if [ $1 == "--myflag" ] ; then
   MYFLAG="yes"
fi
if [ $MYFLAG == "yes" ] ; then
   echo "MYFLAG is on"
else
   echo "MYFLAG is off"
fi
```

We also could have written line 2 through 4 as

```
[ $1 == "--myflag" ] && MYFLAG="yes"
```

As a result, our scripts may be a bit longer, slower, less portable, and more difficult to read than they should be.
For now, we prefer to optimize our **C**/**C++** code.

13.3.2 XPC Library Make System

Although the **GNU** *automake* system takes care of cleaning and packing all of the project files, we still let the `bootstrap` script take care of the cleaning – it deletes some files that the "make clean" process misses. And we currently do the packing manually, by cleaning the project, and then running *tar*.
One reason we do this is that we're not fully proficient in adding files to the correct lines of `Makefile.am`, as in the following examples:

```
MAINTAINERCLEANFILES = Makefile.in dox-warnings.log
EXTRA_DIST = doxygen.cfg xpc_licensing.doxx . . .
```

This is not a good thing to do, and our `bootstrap` script has been a crutch that prevents us from learning to walk the full **GNU** *automake* walk.
We'll throw away that crutch someday.

13.3.3 XPC Library Command-line Options

All of my libraries provide their own command-line options. Each application that uses a number of these libraries will call the help facilities of each of the libraries it uses, so that very comprehensive help for the application can be provided with little effort.
The parsing of the command-line options is very primitive. Here are its undesirable features:

- An explicit `while` loop for parsing the options.
- Explicit handling of short versus long forms of command-line options.

Contrast this to GNU's getopt(3) [run `man 3 getopt` to learn more.]
We tried to use `getopt()`, and we found it to be not much more brief and no easier to understand (when reading the code) than our style. Or so we rationalized as we went on to do it the brute-force way.
It would be cool to extend `getopt()` to support help text, and then we might consider using it. If we are being silly, let me know.
For now, my hard-coded loops are easy to read and easy to maintain (until I accrue more than a dozen or two options in each library).
XPC command-line processing has room for improvement, but it suffices for now.

13.3.4 XPC Library Standard Output

When reading some of the **XPC C++** code, one will notice how often we use `fprintf()` instead of `std::cout`, `cerr`, and `operator >>`.
Why? Surely we should stick with `iostream` in **C++** code!
Well, yes. But trying to do formatted output with `iostream` can be very difficult to read and difficult to get right in appearance.
Consider the following sample from `unit_test.c`:

```
fprintf
(
   stdout,
   "%d %s (%d %s). %d %s.\n"
   "  %s: %d (%s %d, %s %d, %s %d)\n"
   ,
   unit_test_count(tests), _("tests completed"),
   unit_test_subtest_count(tests), _("sub-tests encountered"),
   tests->m_Total_Errors, _("failed"),
   _("First failed unit-test number"),
   tests->m_First_Failed_Test+1,
   _("Group"), tests->m_First_Failed_Group,
   _("Case"), tests->m_First_Failed_Case,
   _("Sub-test"), tests->m_First_Failed_Subtest
);
```

It is very easy to see this is a single line of output going to `stdout`. It might be even easier to read, if more expansive, by putting each argument value on a separate line with some extra indenting.

```
fprintf
(
   stdout,
   "%d %s (%d %s). %d %s.\n"
   "  %s: %d (%s %d, %s %d, %s %d)\n"
   ,
   unit_test_count(tests),
      _("tests completed"),
   unit_test_subtest_count(tests),
      _("sub-tests encountered"),
   tests->m_Total_Errors,
      _("failed"),
      _("First failed unit-test number"),
   tests->m_First_Failed_Test+1,
      _("Group"),
   tests->m_First_Failed_Group,
      _("Case"),
   tests->m_First_Failed_Case,
      _("Sub-test"),
   tests->m_First_Failed_Subtest
);
```

Compare it to a **C++** `iostream` implementation:

```
std::cout
   << unit_test_count(tests) <<  _("tests completed")
   << "(" << unit_test_subtest_count(tests) << _("sub-tests encountered")
   << "). " << tests->m_Total_Errors << _("failed") << std::endl
   << "  " << _("First failed unit-test number") << ": "
   << tests->m_First_Failed_Test+1
   << "(" << _("Group") << tests->m_First_Failed_Group
   << ", " << _("Case") << tests->m_First_Failed_Case
   << ", " << _("Sub-test") << tests->m_First_Failed_Subtest
   << ")" << std::endl
   ;
```

Which is easier to understand? And note that the **C++** example would be even more convoluted to read if one had to add `setw` macros to control the widths of the fields.
"But what about type safety?", you may cry. Well, does it really matter in this short output sample? If it did, the **GNU** *gcc* covers that issue, because it explicitly warns about any arguments that do not match the corresponding `printf()` format specification.
One disadvantage of using `printf()` in **C++** code is that one must use `std::string::c_str()` to output the string. Again, though, *gcc* will warn you if you forget to do this.

Now, if you have a lot of output, or output that is impossible to format for the screen, then it makes sense to use `iostream`, because you really do need the type safety as a back stop, and there's no pretty way to output, say, 256 columns of data.
Here's an ever more extreme example, showing how to make the format show the output layout better, at the expense of extra lines of code:

```
fprintf
(
   stdout,
   "\n"
   "%d %s.\n"
   "%s.\n"
   ,
   unit_test_subtest_count(tests), _("sub-tests encountered"),
   _("Tests summarized, not performed")
);
```

The format is written with one real line per newline. The data arguments are layed out to exactly match the format exactly.
If you are a vertical speed reader, you might like this layout. Or are you of the following school?

```
fprintf(stdout, "\n" "%d %s.\n" "%s.\n" , unit_test_subtest_count(tests), _("sub-tests encountered"), _("Te
```

Here is a case that is common in our code, where the vertical-school method really stands out:

```
fprintf
(
   stdout,
   "- unit_test_status_t:\n"
   "-    m_Test_Options:           %p\n"
   "-    m_Group_Name:             %s\n"
   "-    m_Case_Description:       %s\n"
   "-    m_Subtest_Name:           %s\n"
   "-    m_Test_Group:             %d\n"
   "-    m_Test_Case:              %d\n"
   "-    m_Subtest:                %d\n"
   "-    m_Test_Result:            %s\n"
   "-    m_Subtest_Error_Count:    %d\n"
   "-    m_Failed_Subtest:         %d\n"
   "-    m_Test_Disposition:       %d\n"
   "-    m_Start_Time_us.tv_sec:   %d\n"
   "-    m_Start_Time_us.tv_usec:  %d\n"
   "-    m_End_Time_us.tv_sec:     %d\n"
   "-    m_End_Time_us.tv_usec:    %d\n"
   "-    m_Test_Duration_ms:       %g\n"
   ,
   (void *) status->m_Test_Options,
   status->m_Group_Name,
   status->m_Case_Description,
   status->m_Subtest_Name,
   status->m_Test_Group,
   status->m_Test_Case,
   status->m_Subtest,
   status->m_Test_Result ? _("true") : _("false"),
   status->m_Subtest_Error_Count,
   status->m_Failed_Subtest,
   status->m_Test_Disposition,
   (int) status->m_Start_Time_us.tv_sec,
   (int) status->m_Start_Time_us.tv_usec,
   (int) status->m_End_Time_us.tv_sec,
   (int) status->m_End_Time_us.tv_usec,
   status->m_Test_Duration_ms
);
```

That's about as readable as it can get!
Here's another example, where we are separating the format information from the output data

```
if (cut_not_nullptr(message))
   fprintf(stderr, "? %s\n", message);
else
   fprintf(stderr, "? null message pointer in xpccut_errprint()";
```

It is easy to understand the parallel between the two messages. However, this sample, actual code, is still wrong. We left out the newline in the second `fprintf()` statement! Also, we forgot to internationalize the second message. Let's fix it, and also treat the two messages more uniformly, and save an `fprintf()` call, too:

```
if (cut_is_nullptr(message))
   message = _("null message pointer in xpccut_errprint()");

fprintf(stderr, "? %s\n", message);    // one uniform format
```

Note that only the message, and not the format, is wrapped in the **GNU** internationalization macro, `_()`.
If the two messages aren't of the same format, it still pays to separate the format and the output:

```
if (cut_not_nullptr_2(message, extra))
   fprintf(stderr, "? %s: %s\n", extra, message);
else
   fprintf(stderr, "? %s\n", _("null pointer(s), xpccut_errprint_ex()"));
```

These code samples illustrate the following principles:

- Make the output format in the code resemble the output format in the output as much as possible.
- Separate output strings from the format strings.
  - This makes the formatting easier to follow.
  - It allows for more and better usage of the internationalization macro. (The **GNU** documentation for `gettext()` explains well how to prepare one's code for internationalization.)
- Make sure the spacing of output is consistent from module to module, to allow the output of all the help text (for example) to have a seamless look no matter how many help functions are concatenated.
- Always use the output descriptor, even if writing to `stdout`.
  - Makes the destination more obvious.
  - Makes the output calls more uniform.
  - Makes it easier to retrofit output to a log file in the future. The tiny bit of inefficiency should not matter in most cases.

However, these items are merely principles. Feel free to deviate from them whenever it is more convenient to do so. We do!
By the way, when obtaining samples for this section, we found that we had coded these samples with two mistakes:

1. We had left off the newline character (`\n`) from a lot of the constant message strings.
2. We had output these message strings as is, with "?" embedded in them.

By being so compulsive about the code, we were able to find some mistakes and improve the adherence of the code to principle, and use them for demonstration above. The more time you can devote to combing through your code, the better your code will be.

13.4 XPC Documentation Philosophy

I have nothing to say on this topic, yet, except that I think documentation should be completely free, while being protected from malicious modification.

13.5 XPC Affero Philosophy

I have nothing to say on this topic, yet.

13.6 XPC Philosophy Summary

As one spends years and decades programming, one's style and philosophy tends to change much as time goes on. Sometimes one can go for years with the same style, then decide to throw it all away for something better. Guess who is guilty of that.

Chapter 14

Licenses, XPC Library Suite

This file is a copy of the main license file.

14.1 License Terms for the XPC Library Suite

These licenses apply to each sub-project and file artifact in the **XPC** library suite.
Wherever the term **XPC** is encountered in this project, it refers to the **XPC Development Suite** on **SourceForge.net**, which is also called the **XPC Library Suite**, and may be provided at other sites.

14.2 XPC Application License

The **XPC** application license is the **GNU GPLv3**.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.
This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
You should have received a copy of the GNU General Public License along with this program; if not, write to the

```
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301, USA.
```

The text of the GNU GPL version 3 license can also be found here:
http://www.gnu.org/licenses/gpl-3.0.txt

14.3 XPC Library License

The **XPC** library license is the **GNU LGPLv3**.

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.
This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Lesser Public License for more details.
You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the

```
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301, USA.
```

The text of the GNU LGPL version 3 license can also be found here:
http://www.gnu.org/licenses/lgpl-3.0.txt

14.4 XPC Documentation License

The **XPC** documentation license is the **GNU FDLv1.3**.
Copyright (C) 2008-2022 by Chris Ahlstrom
This documentation is free documentation; you can redistribute it and/or modify it under the terms of the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3 of the License, or (at your option) any later version.
This documentation is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Free Documentation License for more details.
You should have received a copy of the GNU Free Documentation License along with this documentation; if not, write to the

```
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301, USA.
```

The text of the GNU FDL version 1.3 license can also be found here:
http://www.gnu.org/licenses/fdl.txt

14.5 XPC Affero License

The **XPC** "Affero" license is the **GNU AGPLv3**.
Copyright (C) 2008-2022 by Chris Ahlstrom
This server software is free server software; you can redistribute it and/or modify it under the terms of the GNU Affero General Public License as published by the Free Software Foundation; either version 1.3 of the License, or (at your option) any later version.
This documentation is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Free Documentation License for more details.
You should have received a copy of the GNU Affero General Public License along with this server software; if not, write to the

```
Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301, USA.
```

The text of the GNU AGPL version 3 license can also be found here:
http://www.gnu.org/licenses/agpl-3.0.txt
At the present time, no **XPC** project uses the "Affero" license.

14.6 XPC License Summary

Include one of these licenses in your Doxygen documentation with one of the following Doxygen tags:

```
\ref xpc_suite_license_subproject
\ref xpc_suite_license_application
\ref xpc_suite_license_library
\ref xpc_suite_license_documentation
\ref xpc_suite_license_affero
```

For more information on navigating GNU licensing, see this page:

```
http://www.gnu.org/licenses/
```

Copies of these licenses (and some logos) are provided in the `licenses` directory of the main project (or you can search for them at *gnu.org*).