ISOcxx: The C++ Portability Package

Walter E. Brown

Fermi National Accelerator Laboratory, Batavia, Illinois, USA

Abstract

The level of C++ compilers' adherence to the ISO C++ standard varies considerably from compiler to compiler. This variability has significantly hindered users' attempts at standard-compliant C++ coding practices. `ISOcxx` is a software package that addresses such deficient aspects of users' C++ development environments. This portability package:

- probes an environment to identify areas of non-compliance ("defects") with the standard, and
- supplies, where possible, "compliance code" so as to mitigate ("cure") the ill effects of the detected defects.

Each defect typically results from a feature that is required by the ISO C++ standard, but that a particular environment omits entirely, provides only incompletely, matches to an outdated draft of the standard, or otherwise incorrectly supports.

A cure is applicable if test programs demonstrating the corresponding defect can be successfully compiled and run when the compliance code is incorporated. Where no compliance code is available, client code is nonetheless made aware of the defect and can thus avoid the offending construct.

Thus, this package allows client code to be maximally compliant with the international C++ standard, yet still be acceptable to many otherwise-defective environments.

Keywords: `ISOcxx`, C++, portability, programming, portable programming

1 Project Overview

The *C++ Portability Project* at Fermilab grew out of informal discussions in November 1998, "to standardize C++ library and language feature access across multiple platform/compiler combinations." Given the abbreviation `ISOcxx` (International Standards Organization C++), the project's formal goals are to:

- document known defects of C++ development environments for supported Run II platforms;
- provide, to the extent possible, C++ standard-compliant fixes and workarounds that serve as cures for known defects' consequences;
- provide a framework for testing known defects and for testing compliance code proposed as cures for defects;
- provide a centralized source of information regarding known problems and possible cures for various platforms; and
- provide a framework for easily adding both newly-discovered defects and new (or improved) defect cures to the package.

In brief, as stated in the original project proposal, "[t]he main goal of the new package is to allow developers at all levels of activity to program in a way as close to standard C++ as possible with a minimum of intrusion into user code."

To accomplish this, `ISOcxx` incorporates fixes to C++ development platforms to account for environments[1] that do not comply with the ISO C++ language and library standard. Each C++ language or library feature that is not compliant is known as a *defect*. Typically, such a defect results from a feature that is required by the ISO C++ standard, but that a particular environment omits entirely, provides only incompletely, matches to an earlier (outdated, superseded) draft of the standard, or otherwise incorrectly supports (in whole or in part).

In this package, each defect is defined by one or more test programs that demonstrate a specific area of non-compliance. To the extent possible, each such defect (or group of related defects) is accompanied by *compliance code*, additional software that addresses the defect and attempts to *cure* (remedy or ameliorate) some or all of its consequences. The success of any cure is evaluated by the extent to which all test programs demonstrating the corresponding defect can be successfully compiled, linked, and run when the compliance code is incorporated and correctly used.

2 An Example

As a small sample of a defect cured by this package, consider the following client code fragment:

```
for ( int k = 0;  k < N;  ++k )  {
  // ...
}
int k;
```

This code complies with the C++ Standard[2], but is rejected by compilers that conform to earlier rules. In particular, the 1998 Standard specifies, for names declared in `for`-statements, scope rules that differ from those originally detailed in the ARM[3]. Under the older rules, the above code fragment will not compile, because the `k` declared in the `for` will still be in scope when the second `k` is defined. However, using this `ISOcxx` portability package, the fragment will compile, without change, according to the Standard's scope rules (the first `k`'s scope ends at the `for`'s closing brace) in compliant and noncompliant environments alike.

3 Usage by Client Software

Client source code interfaces with this portability package via several simple practices that provide the desired objective of maximally portable C++ software. For example, all client source code first connects to this portability package by inserting, at the top of each source file, the directive:

```
#include "ISOcxx.h"
```

In every client source file, any code ahead of this directive will not receive the full benefits of any of the package's available cures. Therefore, it is strongly recommended that the directive come first. (Leading comments are, of course, OK.)

Beyond the presence of this directive, much of this portability package's compliance code is *transparent*: inspection of client source code reveals no trace that compliance code was needed or used. Such code typically takes the form of replacements for keywords or library headers in which defects have been demonstrated. Implicit activation of such code is made possible via the cooperation of the `ISOcxx` header and the compilation and linking practices described below.

Unfortunately, not all known defects can be cured transparently. In these cases, the compliance code does require client code cooperation to achieve its cure, typically in the form of specific

[1] For purposes of this package, such a development environment is composed of: a hardware platform, a version of an operating system, a version of a C++ compiler, a version of a C++ run-time library (if distinct from the compiler version), and a set of options used to preprocess, compile, and link target software.

[2] *International Standard ISO/IEC 14882: Programming languages — C++*. 1998-09-01.

[3] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*, 1990. ISBN 0-201-51459-1.

macro calls in lieu of (or in addition to) otherwise standard C++.

Due to limitations in the state of the art, some known defects do not have associated compliance code. We therefore recommend to avoid the use of the language or library feature that engenders such defects. If such avoidance is undesirable, it may be possible for client software to provide two code fragments, one that operates in the presence of the defect while the other operates in the defect's absence.

To facilitate this, this portability package shares its knowledge of the presence or absence of each defect in a given environment. For each defect, the package provides a unique symbol (of the form `DEFECT_*`) that is defined if and only if the defect is present, and a unique symbol (of the form `ISOcxx_*`) that is defined if and only if the defect is absent. Via these symbols, client code has the compile-time ability to interrogate the status of any defects of interest, and to take direct control via code conditioned on the presence of any corresponding symbol. Documentation for each defect clearly identifies the pair of symbols with which the defect is associated.

Because this portability package is designed to bridge any gaps between an environment and the C++ language standard, it is typically necessary to make certain adjustments to compilation and link commands to enable seamless cooperation between the package and the environment. The main adjustment is to those flags in the compile command that govern the order in which directories are searched to locate system header files. In particular, `ISOcxx` directories must be searched for these headers ahead of any other (environment-specific) directory.

Fermilab's Run II build tool, SoftRelTools (SRT), is already environment-dependent, hence client code compiled and linked via SRT needs little or no adjustment as described above. Absent SRT, the needed adjustments will inherently vary by environment.

4 Package Configuration and Maintenance

Configuration consists of running a script (`configure`) that probes the environment to determine its behavior and so determine what, if any, cures the package needs to enable. Thus, the behavior of this portability package is tied to the environment in which it is built, which must then be identical in every respect to the environment in which it is employed. A change in a single command-line switch, for example, may very well be sufficient to change the behavior of the environment, hence in the behavior of this package. Therefore this package must be reconfigured (a matter of two to five minutes) whenever any component of the environment changes. Multiple configurations (*e.g.*, one for debugging, another for optimization, *etc.*) are supported via SRT.

Internally, the configuration process consists primarily of a sequence of attempts to compile, link, and execute (in the defined environment) all of `ISOcxx`'s test programs that define the defects this package addresses. For each such attempt, there is printed a message of the form:

```
checking <defect>... <status>
```

in which `<defect>` identifies the test under consideration, and `<status>` identifies the result of the test. In many cases, `<defect>` includes the name of the specific test program being attempted. The primary `<status>` information messages are:

- `ok` (the defect is not present in the local environment),
- `defective` (the defect is present and we will try the proposed cure),
- `cured` (the compliance code for this defect is efficacious), and
- `PROBLEM!` (the compliance code does not cure the defect as expected).

The last of these may arise while attempting to maintain the package (by extending the defect coverage or by porting to a new environment), but not otherwise.

The consolidated results of this environment-probing process are recorded in a single file (`ISOcxx.h`). As described above, this file represents the public interface to `ISOcxx` client software.

It must therefore be installed in a directory that is automatically searched whenever client software is built.

This portability package is configured via SRT by issuing the command:

```
gmake ISOcxx.include
```

Outside a Fermilab SRT environment, it is necessary to build this portability package by running the `configure` script (found in the package's top-level directory) by hand. Then, the resulting `ISOcxx.h` file must be installed in a suitable directory that will be implicitly or explicitly searched when client software is built. (The `ISOcxx` package has not as yet been tested outside the SRT environment; detailed instructions will be available once such testing has been completed.)

This portability package is maintained via the use of the `autoconf` utility (version 2.13 or higher). This software is designed to automate the production of the script that is run during configuration to customize a package to its local environment. For `ISOcxx` purposes, the script produced in this way is named `configure`, and may be found in the `ISOcxx` package's top-level directory. Note that `autoconf` is not needed in order to distribute, configure, or use `ISOcxx`; `autoconf` (and its prerequisite software, `m4`) is only needed for package maintenance.

5 Current Status and Future Plans

The alpha release of this C++ Portability Package is currently available via the FPCLTF ("Zoom") project repository[4]. This release addresses approximately 30 defects that are primarily language-related and approximately 20 defects that are primarily library-related. This is in addition to the important issue of outdated, nonstandard, and nonportable library header names (*e.g.*, `<iostream.h>`).

This release supports operations only within a SoftRelTools environment. A beta release, including support for non-SRT environments, is expected within a few weeks. Once proven stable, `ISOcxx` may be incorporated into CLHEP (preliminary discussions have yielded favorable comments) and thus become available through the usual CLHEP channels.

6 Acknowledgments

Many individuals at Fermilab contributed to the list of defects this project addresses. In particular, the pioneering work of Robert Kennedy is gratefully acknowledged, as are contributions from David Adams, Chris Green (co-author of the original proposal), Scott Snyder, Gordon Watts, and the Zoom[4] project. Non-Fermi sources included the Blitz++[5], CLHEP[6], and STLport[7] projects.

7 Call for Participation

Contributions to the C++ Portability Project are invited and welcomed. Questions, concerns, suggested improvements, additional defects, or other extensions to this package may be sent via email to `zoom-support@fnal.gov`. When reporting a new defect, please also furnish a (small!) program that demonstrates the defect in isolation, together (if possible) with a suggested cure; kindly also identify the specific environment that first manifested the defect.

[4] Mark Fischler, *et al.*: `www.fnal.gov/docs/working-groups/fpcltf/fpcltf.html`.
[5] Todd Veldhuizen: `oonumerics.org/blitz`.
[6] Evgueni Tchernaiev, *et al.*: `wwwinfo.cern.ch/asd/lhc++/clhep/index.html`.
[7] Boris Fomitchev: `www.STLport.org`.