Portable C++ for R Packages

by Martyn Plummer

Abstract Package checking errors are more common on Solaris than Linux. In many cases, these errors are due to non-portable C++ code. This article reviews some commonly recurring problems in C++ code found in R packages and suggests solutions.

CRAN packages are tested regularly on both Linux and Solaris. The results of these tests can be found at `http://cran.r-project.org/web/checks/check_summary.html`. Currently, 24 packages generate errors on Linux while 125 packages generate errors on Solaris.[1] A major contribution to the higher frequency of errors on Solaris is lack of portability of C++ code. The CRAN Solaris checks use the Oracle Solaris Studio 12.2 compiler, which has a much more stringent interpretation of the C++ standard than the GCC 4.6.1 compiler used for the checks on on Linux, and will therefore reject code that compiles correctly with GCC.

It seems plausible that most R package developers work with GCC and are therefore not aware of portability issues in their C++ code until these are shown by the CRAN checks on Solaris. In fact, many of the testing errors are due to a few commonly recurring problems in C++. The aims of this article are to describe these problems, to help package authors diagnose them from the Solaris error message, and to suggest good practice for avoiding them.

The scope of the article is limited to basic use of C++. It does not cover the use of the **Rcpp** package (Eddelbuettel and Francois, 2011) and the Scythe statistical library (Pemstein et al., 2011), which are used to support C++ code in some R packages, nor issues involved in writing your own templates.

Before describing the portability issues in detail, it is important to consider two general principles that underlie most portability problems.

Firstly, *C++ is not a superset of C*. The current C standard is ISO/IEC 9899:1999, usually referred to as *C99* after its year of publication. Most C++ compilers support the ISO/IEC 14882:1998 (*C++98*), which predates it.[2] Thus, the two languages have diverged, and there are features in C99 that are not available in C++98.

The g++ compiler allows C99 features in C++ code. These features will not be accepted by other compilers that adhere more closely to the C++98 standard. If your code uses C99 features, then it is not portable.

The C++ standard is evolving. In August 2011, the ISO approved a new C++ standard which was published in September 2011 and is known as *C++11*. This should remove much of the divergence between the two languages. However, it may take some time for the new C++11 standard to be widely implemented in C++ compilers and libraries. Therefore this article was written with C++98 in mind.

The second general issue is that g++ has a permissive interpretation of the C++ standard, and will typically interpret ambiguous code for you. Other compilers require stricter conformance to the standard and will need hints for interpreting ambiguous code. Unlike the first issue, this is unlikely to change with the evolving C++ standard.

The following sections each describe a specific issue that leads to C++ portability problems. By far the most common error message produced on Solaris is 'The function `foo` must have a prototype'. In the following, this is referred to as a *missing prototype error*. Problems and solutions are illustrated using C++ code snippets. In order to keep the examples short, ellipses are used in place of code that is not strictly necessary to illustrate the problem.

C99 functions

Table 1 shows some C functions that were introduced with the C99 standard and are not supported by C++98. These functions are accepted by g++ and will therefore pass R package checks using this compiler, but will fail on Solaris with a missing prototype error.

C99 Function	R replacement
expm1(x)	expm1(x)
log1p(x)	log1p(x)
trunc(x)	ftrunc(x)
round(x)	fprec(x, 0)
lgamma(x)	lgammafn(x)

Table 1: Some expressions using C99 functions and their portable replacements using functions declared in the '<Rmath.h>'

R packages have access to C functions exposed by the R API, which provides a simple workaround for these functions. All of the expressions in the left hand column of Table 1 can be replaced by portable expressions on the right hand side if the header '<Rmath.h>' is included.

A less frequently used C99 function is the cube root function `cbrt`. The expression `cbrt(x)` can be replaced by `std::pow(x, (1./3.))` using the `pow` function defined in the header '<cmath>'.

[1] Patched version of R 2.14.0, on 10 December 2011, x86 platform.

[2] Although a technical corrigendum of the C++ standard was published in 2003, it provided no new features.

C99 macros for special values

The C99 standard also introduced the constants `NAN` and `INFINITY` as well as the macros `isfinite`, `isinf`, `isnan` and `fpclassify` to test for them. None of these are part of the C++98 standard. Attempts to use the macros on Solaris will result in a missing prototype error, and the constants will result in the error message "NAN/INFINITY not defined".

As with the C99 functions above, the R API provides some facilities to replace this missing functionality. The R macros `R_FINITE` and `ISNAN` and the R function `R_IsNan` are described in the R manual "Writing R Extensions" and are accessed by including the header file '<R.h>'. They are not exactly equivalent to the C99 macros because they are adapted to deal with R's missing value `NA_REAL`.

If you need access to a non-finite value then '<R_ext/Arith.h>' provides `R_PosInf`, `R_NegInf` and `R_NaReal` (more commonly used as `NA_REAL`).

Variable-length arrays

A *variable-length array* is created when the size of the array is determined at runtime, not compile time. A simple example is

```
void fun(int n) {
   double A[n];
   ...
}
```

Variable length arrays are not part of the C++98 standard. On Solaris, they produce the error message "An integer constant expression is required within the array subscript operator".

Variable-length arrays can be replaced by an instantiation of the `vector` template from the Standard Template Library (STL). Elements of an STL vector are accessed using square bracket notation just like arrays, so it suffices to replace the definition of the array with

```
std::vector<double> A(n);
```

The STL `vector` template includes a destructor that will free the memory when `A` goes out of scope.

A function that accepts a pointer to the beginning of an array can be modified to accept a reference to an STL vector. For example, this

```
void fun(double *A, unsigned int length) { ... }
```

may be replaced with

```
void fun(std::vector<double> &A) {
   unsigned int length = A.size();
   ...
}
```

Note that an STL vector can be queried to determine its size. Hence the size does not need to be passed as an additional argument.

External library functions that expect a pointer to a C array, such as BLAS or LAPACK routines, may also be used with STL vectors. The C++ standard guarantees that the elements of a `vector` are stored contiguously. The address of the first element (*e.g.* `&a[0]`) is thus a pointer to the start of an underlying C array that can be passed to external functions. For example:

```
int n = 20;
std::vector<double> a(n);
...   // fill in a
double nrm2 = cblas_dnrm2(n, &a[0], 1);
```

Note however that boolean vectors are an exception to this rule. They may be packed to save memory, so it is not safe to assume a one-to-one correspondence between the underlying storage of a boolean `vector` and a boolean array.

Function overloading

The C++ standard library provides overloaded versions of most mathematical functions, with versions that accept (and return) a `float`, `double` or `long double`.

If an integer constant is passed to these functions, then g++ will decide for you which of the overloaded functions to use. For example, this expression is accepted by g++.

```
#include <cmath>
using std::sqrt;

double z = sqrt(2);
```

The Oracle Solaris Studio compiler will produce the error message 'Overloading ambiguity between `std::sqrt(double)` and `std::sqrt(float)`'. It requires a hint about which version to use. This hint can be supplied by ensuring that the constant is interpreted as a double

```
double z = sqrt(2.);
```

In this case '`2.`' is a double constant, rather than an integer, because it includes a decimal point. To use a `float` or `long double`, add the qualifying suffix `F` or `L` respectively.

The same error message arises when an integer variable is passed to an overloaded function inside an expression that does not evaluate to a floating point number.

```
#include <cmath>
using std::sqrt;

bool fun(int n, int m) {
   return n > m * sqrt(m);
}
```

In this example, the compiler does not know if `m` should be considered a `float`, `double` or `long double` inside the `sqrt` function because the return type is `bool`. The hint can be supplied using a cast:

```
return n > m * sqrt(static_cast<double>(m));
```

Namespaces for C functions

As noted above, C99 functions are not part of the C++98 standard. C functions from the previous C90 standard are allowed in C++98 code, but their use is complicated by the issue of namespaces. This is a common cause of missing prototype errors on Solaris.

The C++ standard library offers two distinct sets of header files for C90 functions. One set is called '<cname>' and the other is called '<name.h>' where "name" is the base name of the header (*e.g.* 'math', 'stdio', ...). It should be noted that the '<name.h>' headers in the C++ standard library are *not* the same files as their namesakes provided by the C standard library.

Both sets of header files in the C++ standard library provide the same declarations and definitions, but differ according to whether they provide them in the standard namespace or the global namespace. The namespace determines the function calling convention that must be used:

- Functions in the standard namespace need to be referred to by prefixing `std::` to the function name. Alternatively, in source files (but not header files) the directive `using std::foo;` may be used at the beginning of the file to instruct the compiler that `foo` always refers to a function in the standard namespace.

- Functions in the global namespace can be referred to without any prefix, except when the function is overloaded in another namespace. In this case the scope resolution prefix '`::`' must be used. For example:

```
using std::vector;

namespace mypkg {
   //Declare overloaded sqrt
   vector<double> sqrt(vector<double> const &);
   //Use sqrt in global namespace
   double y = ::sqrt(2.0);
}
```

Although the C++98 standard specifies which namespaces the headers '<cname>' and '<name.h>' should use, it has not been widely followed. In fact, the C++11 standard has been modified to conform to the current behaviour of C++ compilers (JTC1/SC22/WG21 - The C++ Standards Committee, 2011, Appendix D.5), namely:

- Header '<cname>' provides declarations and definitions in the namespace `std`. It may or may not provide them in the global namespace.

- Header '<name.h>' provides declarations and definitions in the global namespace. It may or may not provide them in the namespace `std`.

The permissiveness of this standard makes it difficult to test code for portability. If you use the '<cname>' headers, then g++ puts functions in both the standard and global namespaces, so you may freely mix the two calling conventions. However, the Oracle Solaris Studio compiler will reject C function calls that are not resolved to the standard namespace.

The key to portability of C functions in C++ code is to use one set of C headers consistently and check the code on a platform that does not use both namespaces at the same time. This rules out g++ for testing code with the '<cname>' headers. Conversely, the GNU C++ standard library header '<name.h>' does not put functions in the `std` namespace, so g++ may be used to test code using '<name.h>' headers. This is far from an ideal solution. These headers were meant to simplify porting of C code to C++ and are not supposed to be used for new C++ code. However, the fact that the C++11 standard still includes these deprecated headers suggests a tacit acceptance that their use is still widespread.

Some g++ shortcuts

The g++ compiler provides two further shortcuts for the programmer which may occasionally throw up missing prototype errors on Solaris.

Headers in the GNU C++ standard library may be implicitly included in other headers. For example, in version 4.5.1, the '<fstream>' and '<stdexcept>' headers both include the '<string>' header. If your source file includes either of these two headers, then you may may use strings without an `#include <string>;` statement, relying on this implicit inclusion. Other implementations of the C++ standard library may not do this implicit inclusion.

When faced with a missing prototype error on Solaris, it is worth checking a suitable reference (such as `http://www.cplusplus.com`) to find out which header declares the function according to the C++ standard, and then ensure that this header is explicitly included in your code.

A second shortcut provided by g++ is *argument-dependent name lookup* (also known as *Koenig lookup*), which may allow you to omit scope resolution of functions in the standard namespace. For example,

```
#include <algorithm>
#include <vector>
using std::vector;
```

```
void fun (vector<double> &y)
{
   sort(y.begin(), y.end());
}
```

Since the arguments to the `sort` algorithm are in the `std` namespace, gcc looks in the same namespace for a definition of `sort`. This code therefore compiles correctly with gcc. Compilers that do not support Koenig lookup require the `sort` function to be resolved as `std::sort`.

Conclusions

The best way to check for portability of C++ code is simply to test it with as many compilers on as many platforms as possible. On Linux, various third-party commercial compilers are available at zero cost. The Intel C++ Composer XE compiler is free for non-commercial software development (Note that you must comply with Intel's definition of non-commercial); the PathScale EkoPath 4 compiler recently became open source; the Oracle Solaris Studio compilers may be downloaded for free from the Oracle web site (subject to license terms). The use of these alternative compilers should help to detect problems not detected by GCC, although it may not uncover all portability issues since they also rely on the GNU implementation of the C++ standard library. It is also possible to set up alternate testing platforms inside a virtual machine, although the details of this are beyond the scope of this article.

Recognizing that most R package authors do not have the time to set up their own testing platforms, this article should help them to interpret the feedback from the CRAN tests on Solaris, which provide a rigorous test of conformity to the current C++ standard. Much of the advice in this article also applies to a C++ front-end or an external library to which an R package may be linked.

An important limitation of this article is the assumption that package authors are free to modify the source code. In fact, many R packages are wrappers around code written by a third party. Of course, all CRAN packages published under an open source license may be modified according to the licence conditions. However, some of the solutions proposed here, such as using functions from the R API, may not be suitable for third-party code as they require maintaining a patched copy. Nevertheless, it may still be useful to send feedback to the upstream maintainer.

Acknowledgement

I would like to thank Douglas Bates and an anonymous referee for their helpful comments.

Bibliography

D. Eddelbuettel and R. Francois. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 4 2011. ISSN 1548-7660. URL http://www.jstatsoft.org/v40/i08.

JTC1/SC22/WG21 - The C++ Standards Committee. Working draft, standard for programming language C++. Technical report, ISO/IEC, February 2011. URL http://www.open-std.org/JTC1/SC22/WG21.

D. Pemstein, K. M. Quinn, and A. D. Martin. The scythe statistical library: An open source C++ library for statistical computation. *Journal of Statistical Software*, 42(12):1–26, 6 2011. ISSN 1548-7660. URL http://www.jstatsoft.org/v42/i12.

Martyn Plummer
International Agency for Research on Cancer
150 Cours Albert Thomas 69372 Lyon Cedex 08
France
plummerM@iarc.fr