Windows to UNIX porting, Part 2: Internals of porting C/C++ sources

From compiler attributes to commonly encountered problems

Skill Level: Intermediate

Rahul Kumar Kardam (rahul@syncad.com)
Senior Software Developer
Synapti Computer Aided Design Pvt Ltd

Arpan Sen (arpansen@gmail.com)
Independent author

06 Nov 2007

Part 1 of this series covered the typical C/C++ project types you work with in a Microsoft® Visual Studio® environment and introduced the processes of porting dynamic and static library project variants to a UNIX® platform. Part 2 delves into some of the compiler options used to build Visual C++ projects and the UNIX and g++ equivalents, takes a closer look at the g++ attribute mechanism as it relates to porting, and examines some common problems you might encounter while porting from a 32-bit Windows® environment to a 64-bit UNIX environment. It concludes with an overview of concepts for porting multithreaded applications and an example project that shows you how to pull all these pieces together.

Comparing and contrasting compiler options

The `cl` compilers in both Visual C++ and GNU g++ come with several options. While you can use `cl` as a standalone tool for compilation purposes, Visual C++ provides a neat integrated development environment (IDE) to set compiler options. Software developed with Visual Studio® often uses compiler-specific and platform-dependant features that are controlled using compiler or linker options. When you are porting sources across platforms with different compilers or tool chains, it is important to understand compiler options. This section explores some of

the most useful compiler options.

Enable string pooling

Consider the following code snippet:

```
char *string1= "This is a character buffer";
char *string2= "This is a character buffer";
```

If the string-pooling option, [/GF], is enabled in Visual C++, then a single copy of the string is kept in the program image during execution and `string1` equals `string2`. It is interesting to note that the g++ behavior is the exact reverse of this and by default `string1` equals `string2`. To disable string pooling in g++, you must add the `-fwritable-strings` option to the g++ command line.

Using wchar_t

The C++ standard defines the `wchar_t` wide character type. If the `/Zc:wchar_t` option is passed to the compiler, Visual C++ treats `wchar_t` as a native type. Otherwise, implementation-specific headers, such as `windows.h`, or standard headers, such as `wchar.h`, need to be included. g++ supports the native `wchar_t` type, and no specific headers need to be included. Note that the size of `wchar_t` varies across platforms. You can use the `-fshort-wchar g++` option to force the `wchar_t` size to two bytes.

Support for C++ Run Time Type Identification

If the sources don't use `dynamic_cast` or the `typeid` operator, Run Time Type Identification (RTTI) might be disabled. By default, RTTI is on in Visual Studio 2005 (that is, the `/GR` switch is on). Using the `/GR-` switch disables RTTI in a Visual Studio environment. Disabling RTTI might help produce an executable of a smaller size. Note that disabling RTTI in a code that contains `dynamic_cast` or `typeid` might produce undesirable effects, including code crash. Consider the snippet in Listing 1.

Listing 1. Code snippet for demonstration of RTTI

```
#include <iostream>
struct A {
  virtual void f()
    { std::cout << "A::f\n"; }
  };

struct B : A {
  virtual void f()
    { std::cout << "B::f\n"; }
  };

struct C : B {
```

```
  virtual void f()
    { std::cout << "C::f\n"; }
  };

int main (int argc, char** argv )
  {
  A* pa = new C;
  B* pb = dynamic_cast<B*> (pa);
  if (pb)
    pb->f();
  return 0;
  }
```

To compile this code snippet in the standalone `cl` compiler outside the Visual Studio IDE, the `/GR` switch needs to be explicitly turned on. Unlike `cl`, the g++ compiler does not need any special option to turn RTTI on. However, just like the `/GR-` option in Visual Studio, g++ has the `-fno-rtti` option that explicitly turns off RTTI. Compiling this snippet in g++ with the `-fno-rtti` option causes compilation errors to be reported. However, `cl` compiles this code even without the `/GR` option, but the generated executable crashes at run time.

Exception handling

To enable exception handling in `cl`, use the `/GX` compiler option or `/EHsc`. Without either of these options, the `try` and `catch` code can still execute, and the system does not call the destructors of local objects up to the `throw` statement. Exception handling has a performance penalty. Since the compiler generates the code for stack unwinding for every C++ function, this requirement results in a larger executable and slower running code. There might be areas of a particular project where this performance penalty is not acceptable, and you need to turn off the feature. To disable exception handling, you need to remove all try and catch blocks from the sources and compiling the code using the `/GX-` option. The g++ compiler, by default, has exception handling enabled. Passing the `-fno-exceptions` option to g++ has the desired effect. Note that using this option with sources that have `try`, `catch`, and `throw` keywords might result in compilation errors. You still have to manually remove the `try` and `catch` blocks, if any, from the sources and then pass this option to g++. Consider the code in Listing 2.

Listing 2. Code snippet demonstrating exception handling

```
#include <iostream>
using namespace std;

class A { public: ~A () { cout << "Destroying A "; } };
void f1 () { A a; throw 2; }

int main (int argc, char** argv ) {
  try { f1 (); } catch (...) { cout << "Caught!\n"; }
  return 0;
  }
```

Here is the resulting output from `cl` and g++ with and without the options described earlier in this section:

- `cl` with the `/GX` option: `Destroying A Caught!`
- `cl` without the `/GX` option: `Caught!`
- g++ without the `-fno-exceptions`: `Destroying A Caught!`
- g++ with the `-fno-exceptions`: Compile time error

Loop conformance

For loop conformance, consider the code snippet in Listing 3.

Listing 3. For loop conformance

```
int main (int argc, char** argv )
  {
  for (int i=0; i<5; i++);
  i = 7;
  return 0;
  }
```

This code is not supposed to compile according to the ISO C++ guidelines, because the scope of the `i` local variable declared as part of the loop is limited to the loop body and cannot be accessed outside the loop. By default, `cl` passes this code without any error. However, using the `/Zc:forScope` option with `cl` causes compilation failure. The g++ behavior is the exact reverse of `cl` and produces the following error for this test:

```
error: name lookup of 'i' changed for new ISO 'for' scoping
```

To suppress this behavior, you can use the `-fno-for-scope flag` during compilation.

Using g++ attributes

Both Visual C++ and GNU g++ come with non-standard extensions to the language. The g++ attribute mechanism is suitable for use in porting platform-specific features in Visual C++ code. The attribute syntax is of the form `__attribute__ ((attribute-list))`—the attribute list is a comma-separated list of attributes. Individual elements of the attribute list can either be a word or a word followed by possible parameters for the attribute in parentheses. This section looks at some possible uses of attributes in porting.

Function calling convention

You can use specific Visual Studio keywords, such as `__cdecl`, `__stdcall` , and `__fastcall`, to indicate calling conventions for functions to the compiler. Table 1 sums up the details.

Table 1. Calling conventions in a Windows environment

Calling convention	Implied semantics
__cdecl (cl option: /Gd)	Arguments to the called function are pushed onto the stack from right to left. Once executed, the calling function pops the arguments off the stack.
__stdcall (cl option: /Gz)	Arguments to the called function are pushed onto the stack from right to left. Once executed, the calling function pops the arguments off the stack.
__fastcall (cl option: /Gr)	The first two arguments are passed in ECX and EDX registers, while all other arguments are pushed from right to left. The callee function cleans the stack post execution.

The g++ attributes to replicate the same behavior are `cdecl`, `stdcall`, and `fastcall`. Listing 4 reveals the subtle difference in attribute declaration styles in Windows® and UNIX®.

Listing 4. Attribute declaration styles in Windows and UNIX

```
Visual C++ Style Declaration:
double __stdcall compute(double d1, double d2);

g++ Style Declaration:
double __attribute__((stdcall)) compute(double d1, double d2);
```

Structure member alignment

The `/Zpn` structure member alignment controls the alignment in memory for structures. For example, `/Zp8` aligns structures in 8-byte boundaries (also the default), whereas `/Zp16` aligns structures in 16-byte boundaries. You can use the `aligned`g++ attribute for specifying variable alignments, as shown in Listing 5.

Listing 5. Structure member alignment in Windows and UNIX

```
Visual C++ Style Declaration with /Zp8 switch:
struct T1 { int n1; double d1;};

g++ Style Declaration:
struct T1 { int n1; double d1;}  __attribute__((aligned(8)));
```

The effectiveness of aligned attributes, however, is limited by inherent linker limitations. On many systems, the linker is only able to align variables up to a certain maximum alignment.

Visual C++ declspec nothrow attribute

This attribute is a hint to the compiler that the function declared with this attribute and the subsequent functions it calls do not throw an exception. Using this feature is an optimization that reduces overall code size because, by default, even if the code does not throw exceptions, `cl` still generates stack unwinding information for C++ sources. You can use the `nothrow`g++ attribute for similar purposes, as shown in Listing 6.

Listing 6. The nothrow attribute in Windows and UNIX

```
Visual C++ Style Declaration:
double __declspec(nothrow) sqrt(double d1);

g++ Style Declaration:
double __attribute__((nothrow)) sqrt(double d1);
```

A more portable option is to use the standard defined style: `double sqrt(double d1) throw ();`.

Parallels between Visual C++ and g++

Apart from the earlier examples, several parallels exist between Visual C++ and g++ attribute schemes. For example, the `noinline`, `noreturn`, `deprecated`, and `naked` attributes enjoy support from both compilers.

Potential pitfalls in porting from a 32-bit Windows to a 64-bit UNIX environment

C++ code developed in Win32 systems is based on the ILP32 model, where `int`, `long`, and pointer types are 32 bits. UNIX systems follow the LP64 model, where `long` and pointer types are 64 bits, but `int`s remain 32 bits. It's primarily this change that causes most code breakages. This section takes a brief look into two of the most basic problems you might encounter. Porting from 32- to 64-bit systems is an extensive field of study in itself. For more information about this topic, see the Resources section.

Difference in data type sizes

It makes good sense to use data types that are the same across both the ILP32 and LP64 models. In general, you should avoid `long` and `pointer` data as much as possible. Also, it's common to use the data types defined in the `sys/types.h` standard header file, but the size of the individual data types in this file, such as `ptrdiff_t, size_t` and so on, change from 32- to 64-bit models and you need to use with caution.

Memory requirements of individual data structures

The memory requirements of individual data structures might get altered, depending upon the way packing is implemented in the compiler. Consider the snippet in Listing 7.

Listing 7. Flawed struct membership alignment

```
struct s {
          int var1;  // hole between var1 and var2
          long var2;
          int var3; // hole between var3 and ptr1
          char* ptr1;
       };
// sizeof(s) = 32 bytes
```

In an LP64 model, the `long` and `pointer` types are aligned to a 64-bit boundary. Also, the size of the structure is aligned to the size of the largest member within it. In this example, the structure *s* is aligned to an 8-byte boundary, and so is the *s.var2* variable. This causes holes to be created inside the structure, thus inflating memory. The realignment in Listing 8 causes the structure to be of 24 bytes.

Listing 8. Correct struct membership alignment

```
struct s {
          int var1;
          int var3;
          long var2;
          char* ptr1;
       };
// sizeof(s) = 24 bytes
```

Porting multithreaded applications

Technically, a thread is as an independent stream of instructions that the operating system can schedule to run. In both environments, a thread exists within a process and uses the process resources. It has its own independent flow of control, as long as its parent process exists and the operating system supports it. It might share the process resources with other threads that act independently (or dependently), and it dies if the parent process dies. Here is an overview of some typical application program interfaces (APIs) that you can use to make a project multithreaded in both Windows and UNIX environments. The interface of choice is the C run time routines, as opposed to WIN32 APIs, and routines conforming to Portable Operating System Interface (POSIX) threads for simplicity and clarity.

Note: Due to space constraints, we aren't able to provide details of other ways in which to write such an application.

Creating a thread

Windows uses the `_beginthread` API from the C run time library function. There are other Win32 APIs that you can use to create a thread but, moving forward, you are dealing only with C run time library functions. As the name suggests, the `_beginthread()` function creates a thread that executes the routine, where the pointer to that routine is specified as the first argument. This routine uses the `__cdecl` C declaration calling convention and returns void. When the thread returns from that routine, it terminates.

In UNIX, the same is achieved using the `pthread_create()` function. The `pthread_create()` subroutine returns the new thread ID using a thread argument. The caller can use this thread ID to perform various operations on the thread. Check this ID to ensure that the thread exists.

Destroying a thread

The `_endthread` function terminates a thread created by `_beginthread()`. Threads terminate automatically when their sequential execution is finished. The `_endthread()` function is useful for conditional termination from within a thread.

In UNIX, the same result is achieved with the `pthread_exit()` function. This function exits a thread if the normal sequential execution hasn't finished. If `main()` finishes before the threads it has created and exits with `pthread_exit()`, the other threads continue to execute. Otherwise, they are automatically terminated when `main()` finishes.

Synchronization in a thread

For achieving synchronization, you can use mutexes. In Windows, `CreateMutex()` creates mutexes. It returns a handle that can be used by any function that requires a mutex object because all access rights to this mutex are provided. `ReleaseMutex()` is called when the owning thread no longer needs the mutex, and it can be conveniently released to the system. If the calling thread has no ownership of this mutex, this function fails.

In UNIX, a mutex is created dynamically with the `pthread_mutex_init()` routine. This method allows you to set mutex object attributes. Otherwise, it can be created statically when it is declared by a *pthread_mutex_t* variable. To free a mutex object that is no longer needed, `pthread_mutex_destroy()` is used.

Working example of porting a multithreaded application

Now that you're armed with the information presented earlier in this article, let's go over a small example of a program that prints to the console using different threads

executing within the main process. Listing 9 is the source code for multithread.cpp.

Listing 9. Source code for multithread.cpp

```
#include <stdio.h>
#include <stdlib.h>

#ifdef WIN32
  #include <windows.h>
  #include <string.h>
  #include <conio.h>
  #include <process.h>
#else
  #include <pthread.h>
#endif

#define MAX_THREADS 32

#ifdef WIN32
  void InitWinApp();
  void WinThreadFunction( void* );
  void ShutDown();

 HANDLE  mutexObject;
#else
  void InitUNIXApp();
  void* UNIXThreadFunction( void *argPointer );

  pthread_mutex_t mutexObject = PTHREAD_MUTEX_INITIALIZER;
#endif

int     threadsStarted;             // Number of threads started

int main()
{
  #ifdef WIN32
    InitWinApp();
  #else
    InitUNIXApp();
  #endif
}

#ifdef WIN32
void InitWinApp()
  {

  /* Create the mutex and reset thread count. */
  mutexObject = CreateMutex( NULL, FALSE, NULL );   /* Cleared */
  if(mutexObject == NULL && GetLastError() != ERROR_SUCCESS)
    {
    printf("failed to obtain a proper mutex for multithreaded application");
    exit(1);
    }
  threadsStarted = 0;
  for(;threadsStarted < 5 && threadsStarted < MAX_THREADS;
       threadsStarted++)
    {
    _beginthread( WinThreadFunction, 0,  &threadsStarted );
    }
  ShutDown();
  CloseHandle( mutexObject );
  getchar();
  }

void ShutDown()
  {
  while ( threadsStarted > 0 )
```

```
    {
    ReleaseMutex( mutexObject ); /* Tell thread to die. */
    threadsStarted--;
    }
  }

void WinThreadFunction( void *argPointer )
  {
  WaitForSingleObject( mutexObject, INFINITE );
  printf("We are inside a thread\n");
  ReleaseMutex(mutexObject);
  }

#else
void InitUNIXApp()
  {
  int count = 0, rc;
  pthread_t threads[5];

  /* Create independent threads each of which will execute functionC */

  while(count < 5)
    {
    rc = pthread_create(&threads[count], NULL, &UNIXThreadFunction, NULL);
    if(rc)
      {
      printf("thread creation failed");
      exit(1);
      }
    count++;
    }

  // We will have to wait for the threads to finish execution otherwise
  // terminating the main program will terminate all the threads it spawned
  for(;count >= 0;count--)
    {
    pthread_join( threads[count], NULL);
    }
  //Note : To destroy a thread explicitly pthread_exit() function can be used
  //but since the thread gets terminated automatically on execution we did
  //not make explicit calls to pthread_exit();
  exit(0);
  }

void* UNIXThreadFunction( void *argPointer )
  {
   pthread_mutex_lock( &mutexObject );
   printf("We are inside a thread\n");
   pthread_mutex_unlock( &mutexObject );
  }

#endif
```

We tested the source code for multithread.cpp with Visual Studio Toolkit 2003 and Microsoft Windows 2000 Service Pack 4 using the following command line:

```
    cl multithread.cpp /DWIN32 /DMT /TP
```

We also tested it on a UNIX platform with g++ compiler Version 3.4.4 using the following command line:

```
g++ multithread.cpp -DUNIX -lpthread
```

Listing 10 is the output of the program in both environments.

Listing 10. Output of multithread.cpp

```
We are inside a thread
We are inside a thread
We are inside a thread
We are inside a thread
We are inside a thread
```

Conclusion

Porting across two completely different platforms, such as Windows and UNIX, calls for knowledge in several areas, including understanding compilers and their options, platform-specific features such as DLLs, and implementation-specific features such as threads. This series of articles serves as an introduction to the numerous facets of porting. For more advanced information about this topic, see the Resources section.

Resources

Learn

- Windows to UNIX porting: Check out other parts in this series.
- "Porting MFC applications to Linux" (developerWorks, April 2002): Read this article for a step-by-step guide to using wxWindows.
- 64-Bit Programming Models: Why LP64?: Visit this page to learn more about the LP64 programming model.
- "Porting Linux applications to 64-bit systems" (developerWorks, April 2006): Read this article for tips and techniques for porting Linux applications to 64-bit systems.
- *Emulating Windows Registry in Unix* (BYTE.com, July 1999): Get installation-specific information for the application here.
- *Multithreading with C and Win32*: Read this article on the Microsoft Developer Network Web site for more information about multithreading with Visual C++.
- pthread: The Open Group provides comprehensive documentation regarding pthread.
- GCC online documentation: This Web site provides manuals for the latest releases of gcc/g++.
- AIX and UNIX: The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- New to AIX and UNIX?: Visit the "New to AIX and UNIX" page to learn more about AIX and UNIX.
- AIX Wiki: A collaborative environment for technical information related to AIX.
- Search the AIX and UNIX library by topic:
  - System administration
  - Application development
  - Performance
  - Porting
  - Security
  - Tips
  - Tools and utilities

    - Java™ technology
    - Linux®
    - Open source

- Safari bookstore: Visit this e-reference library to find specific technical resources.
- developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.
- Podcasts: Tune in and catch up with IBM technical experts.

Get products and technologies

- IBM trial software: Build your next development project with software for download directly from developerWorks.

Discuss

- Participate in the developerWorks blogs and get involved in the developerWorks community.
- Participate in the AIX and UNIX forums:
    - AIX—technical forum
    - AIX 6 Open Beta
    - AIX for Developers Forum
    - Cluster Systems Management
    - IBM Support Assistant
    - Performance Tools—technical
    - Virtualization—technical
    - More AIX and UNIX forums

About the authors

Rahul Kumar Kardam

Rahul Kardam is a senior software developer specializing in complex C++-based electronic design automation tools such as simulators for hardware designs. He has programming experience in both Windows and UNIX platforms. Rahul enjoys tinkering with open source software,

which he uses as a framework for designing robust, scalable code for the design automation tools he works on.

Arpan Sen

Arpan Sen is a lead engineer working on the development of software in the electronic design automation industry. He has worked on several flavors of UNIX, including Solaris, SunOS, HP-UX, and IRIX as well as Linux and Microsoft Windows for several years. He takes a keen interest in software performance-optimization techniques, graph theory, and parallel computing. Arpan holds a post-graduate degree in software systems. You can reach him at arpansen@gmail.com.

Trademarks