How to: Cope with C++ Environments

I

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

Edsger Disjkstra

Selected Writings on Computing: A Personal Perspective, 1982.

I.1 Coping with Compilers

Compilers and programming environments are supposed to be our friends. Once mastered, they stay out of the way and let us concentrate on the task at hand: solving problems and writing programs. However, they can be unbelievably frustrating when they don't work as we expect them to (or just plain don't work).

In this How to I'll provide some guidance on using compilers and IDEs (Integrated Development Environments). More complete information can be found at the books's web site:

```
http://www.cs.duke.edu/csed/tapestry
```

For the purposes of this How to, compilers and IDEs fall into three groups as shown in Table I.1. Most compilers are available at very reasonable prices for educational use. In particular, the Cygwin suite of tools is available for both Linux and Windows NT/95/98, and it's free. See `http://www.cygnus.com` for details.

The libraries of code and classes discussed in this text are accessible via the book's web site for each of the compilers listed in Table I.1. I realize that there are other

Table I.1 Compilers and IDEs.

Platform	Compiler/IDE
Windows 95, 98, NT	
	Metrowerks Codewarrior
	Visual C++
	Borland C++ Builder/5.0x
	Cygwin egcs
Linux/Unix	
	g++
	egcs (preferred)
Macintosh	
	Metrowerks Codewarrior

compilers. Many people use Borland Turbo 4.5; although it runs all the examples in this book except for the graphical examples, it doesn't track the C++ standard and it's really a compiler for an older operating system (Windows 3.1). I strongly discourage people from using it.

In theory, all the programs and classes in this book run without change with any compiler and on any platform. In practice compilers conform to the C++ standard to different degrees. The only differences I've encountered in using the code in this book with different compilers is that as I write this, the egcs compilers still use `<strstream>` and `istrstream` instead of `<sstream>` and `istringstream` for the string stream classes. Otherwise, except for the classes `DirStream` and `DirEntry` from *directory.h* which are platform specific, the other code is the same on all platforms.

I.1.1 Keeping Current

Once printed, a book lasts for several years before being revised. Compilers and IDEs have major new releases at least once a year. Rather than being out-of-date before publication, I'll keep the book's web site current with information about the latest releases of common compilers and IDEs. I'll include a general discussion here about the major issues in developing programs that use a library of classes and functions, but detailed instructions on particular compilers and platforms, including step-by-step instructions for the common environments, can be found on the web.

I.2 Creating a C++ Program

The steps in creating a C++ program are explained in detail in Sections 7.2.3, 7.2.4, and 7.2.5. The steps are summarized here for reference, repeating material from those sections, but augmented with explanations of specific compilers/environments.

1. The **preprocessing** step handles all #include directives and some others we haven't studied. A **preprocessor** is used for this step.
2. The **compilation** step takes input from the preprocessor and creates an **object file** (see Section 3.5) for each .cpp file. A **compiler** is used for this step.
3. One or more object files are combined with libraries of compiled code in the **linking** step. The step creates an executable program by linking together system-dependent libraries as well as client code that has been compiled. A **linker** is used for this step.

I.2.1 The Preprocessor

The preprocessor is a program run on each source file before the source file is compiled. A source file like *hello.cpp*, Program 2.1 is translated into a **translation unit** which is then passed to the compiler. The source file isn't physically changed by the preprocessor, but the preprocessor does use **directives** like `#include` in creating the translation unit that the compiler sees. Each preprocessor directive begins with a sharp (or number) sign # that must be the first character on the line.

Where are `include` Files Located? The preprocessor looks in a specific list of directories to find include files; this list is the **include path**. In most environments you can alter the include path so that the preprocessor looks in different directories. In many environments you can specify the order of the directories that are searched by the preprocessor.

Program Tip I.1: If the preprocessor cannot find a file specified, you'll probably get a warning. In some cases the preprocessor will find a different file than the one you intend; one that has the same name as the file you want to include. This can lead to compilation errors that are hard to fix. If your system lets you examine the translation unit produced by the preprocessor you may be able to tell what files were included. You should do this only when you've got real evidence that the wrong header file is being included.

Changing the Include Path

- In Metrowerks Codewarrior the include path is automatically changed when you add a .cpp file or a library to a project. The path is updated so that the directory in which the added file is located is part of the path. Alternatively, the path can be changed manually using the sequence of menus:

  Edit→ Console-App Settings→ Target → Access Paths

- In Visual C++ the include path must often be changed manually, although projects do automatically generate a list of external dependencies that include header files. To change the include path use the sequence of menus below, then chose *Include Files* to specify where the preprocessor looks for files.

  Tools → Options → Directories

- In Borland, the include path is not always searched in the order in which files are given. To change the include path choose the sequence of menus below, then change the include path in the *Source Directories* section.

  Options → Project → Directories

- The include path for g++ and egcs is specified with a `-I` argument on the command line to the compiler or in a Makefile. Multiple arguments are possible. The line below makes an executable named *prog*, from the source file *prog.cpp*, using the current directory and `/foo/code` as the include path (the current directory is always part of the path).

```
g++ -I. -I/foo/code -o prog prog.cpp
```

I.2.2 The Compiler

The input to the compiler is the translation unit generated by the preprocessor from a source file. The compiler generates an **object file** for each compiled source file. Usually the object file has the same prefix as the source file, but ends in .o or .obj. For example, the source file *hello.cpp* might generate *hello.obj* on some systems. In some programming environments the object files aren't stored on disk, but remain in memory. In other environments, the object files are stored on disk. It's also possible for the object files to exist on disk for a short time, so that the linker can use them. After the linking step the object files might be automatically erased by the programming environment.

Libraries. Often you'll have several object files that you use in all your programs. For example, the implementations of `iostream` and `string` functions are used in nearly all the programs we've studied. Many programs use the classes declared in `prompt.h`, `dice.h`, `date.h` and so on. Each of these classes has a corresponding object file generated by compiling the .cpp file. To run a program using all these classes the object files need to be combined in the linking phase. However, nearly all programming environments make it possible to combine object files into a library which can then be linked with your own programs. Using a library is a good idea because you need to link with fewer files and it's usually simple to get an updated library when one becomes available.

I.2.3 The Linker

The linker combines all the necessary object files and libraries together to create an executable program. Libraries are always needed, even if you are not aware of them. Standard libraries are part of every C++ environment and include classes and functions for streams, math, and so on. Often you'll need to use more than one library. For example, I use a library called *tapestry.lib* for all the programs in this book. This library contains the object files for classes `Dice`, `Date`, `RandGen` and functions from `strutils` among many others. The suffix `.lib` is typically used for libraries.

You aren't usually aware of the linker as you begin to program because the libraries are linked automatically. However, as soon as you begin to write programs that use several .cpp files, you'll probably encounter linker errors.

These errors may be hard to understand. The key thing to note is that they are **linker errors**. Programming environments differ in how they identify linker errors, but all environments differentiate between compilation errors and linker errors. If you get a linker error, it's typically because you forgot a .cpp file in the linking step (e.g., you left it out of the project) or because you didn't implement a function the compiler expected to find.

Bibliography

[AA85] Donald J. Albers and G.L. Alexanderson. *Mathematical People*. Birkhäuser, 1985.

[ACM87] ACM. *Turing Award Lectures: The First Twenty Years 1966–1985*. ACM Press, 1987.

[AS96] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. MIT Press and McGraw-Hill, 1996.

[Asp90] William Aspray. *Computing Before Computers*. Iowa State University Press, 1990.

[Aus98] Matthew H. Austern *Generic Programming and the STL*. Addison-Wesley, 1998.

[Ben86] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[Ben88] Jon Bentley. *More Programming Pearls*. Addison-Wesley, 1988.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin Cummings, 1991.

[Boo94] Grady Booch. *Object Oriented Design and Analysis with Applications*. 2nd ed. Benjamin Cummings, 1994.

[BRE71] I. Barrodale, F.D. Roberts, and B.L. Ehle. *Elementary Computer Applications in Science Engineering and Business*. John Wiley & Sons Inc., 1971.

[Coo87] Doug Cooper. *Condensed Pascal*. W.W. Norton, 1987.

[Dij82] Edsger W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.

[DR90] Nachum Dershowitz and Edward M. Reingold. Calendrical calculations. *Software-Practice and Experience*, 20(9):899–928, September 1990.

[(ed91] Allen B. Tucker (ed.). *Computing Curricula 1991 Report of the ACM/IEEE-CS Joint Curriculum Task Force*. ACM Press, 1991.

[EL94] Susan Epstein and Joanne Luciano, editors. *Grace Hopper Celebration of Women in Computing*. Computing Research Association, 1994. HopperBook@cra.org.

[Emm93] Michele Emmer, editor. *The Visual Mind: Art and Mathematics*. MIT Press, 1993.

[G9̈5] Denise W. Gürer. Pioneering women in computer science. *Communications of the ACM*, 38(1):45–54, January 1995.

[Gar95] Simson Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, 1995.

[GHJ95] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides *Design Patterns: Elements of Reusable Object-Oriented Programming* Addison-Wesley, 1995

[Gol93] Herman H. Goldstine. *The Computer from Pascal to von Neumann*. Princeton University Press, 1993.

[Gri74] David Gries. On structured programming - a reply to smoliar. *Communications of the ACM*, 17(11):655–657, 1974.

[GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, 1993.

[Har92] David Harel. *Algorithmics, The Spirit of Computing*. Addison-Wesley, second edition, 1992.

[Hoa89] C.A.R. Hoare. *Essays in Computing Science*. Prentice-Hall, 1989. (editor) C.B. Jones.

[Hod83] Andrew Hodges. *Alan Turing: The Enigma*. Simon & Schuster, 1983.

[Hor92] John Horgan. Claude e. shannon. *IEEE Spectrum*, April 1992.

[JW89] William Strunk Jr. and E.B. White. *The Elements of Style*. MacMillan Publishing Co., third edition, 1989.

[Knu97] Donald E. Knuth. *The Art of Computer Programming*, volume 1 Fundamental Algorithms. Addison-Wesley, third edition, 1997.

[Knu98a] Donald E. Knuth. *The Art of Computer Programming*, volume 2 Seminumerical Algorithms. Addison-Wesley, third edition, 1998.

[Knu98b] Donald E. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison-Wesley, third edition 1998.

[KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[KR96] Samuel N. Kamin and Edward M. Reingold. *Programming with class: A C++ Introduction to Computer Science*. McGraw-Hill, 1996.

[Mac92] Norman Macrae. *John von Neumann*. Pantheon Books, 1992.

[McC79] Pamela McCorduck. *Machines Who Think*. W.H. Freeman and Company, 1979.

[McC93] Steve McConnell. *Code Complete*. Microsoft Press, 1993.

[MGRS91] Albert R. Meyer, John V. Gutag, Ronald L. Rivest, and Peter Szolovits, editors. *Research Directions in Computer Science: An MIT Perspective*. MIT Press, 1991.

[Neu95] Peter G. Neumann. *Computer Related Risks*. Addison Wesley, 1995.

[Pat96] Richard E. Pattis. *Get A-Life: Advice for the Beginning Object-Oriented Programmer*. Turing TarPit Press, 1999.

[Per87] Alan Perlis. The synthesis of algorithmic systems. In *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[RDC93] Edward M. Reingold, Nachum Dershowitz, and Stewart M. Clamen. Calendrical calculations, ii: Three historical calendars. *Software-Practice and Experience*, 23(4):383–404, April 1993.

[Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[Rob95] Eric S. Roberts. Loop exits and structured programming: Reopening the debate. In *Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, pages 268–272. ACM Press, March 1995. SIGCSE Bulletin V. 27 N 1.

[Rob95] Eric S. Roberts. *The Art and Science of C*. Addison-Wesley, 1995.

[Sla87] Robert Slater. *Portraits in Silicon*. MIT Press, 1987.

[Str87] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1987.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[Mey92] Scott Meyers. *Effective C++*. Addison Wesley, 1992.

[Mey96] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.

[Wei94] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin Cummings, 1994.

[Wil56] M.V. Wilkes. *Automatic Digital Computers*. John Wiley & Sons, Inc., 1956.

[Wil87] Maurice V. Wilkes. Computers then and now. In *ACM Turing Award Lectures: The First Twenty Years*, pages 197–205. ACM Press, 1987.

[Wil95] Maurice V. Wilkes. *Computing Perspectives*. Morgan Kaufmann, 1995.

[Wir87] Niklaus Wirth. From programming language design to compiler construction. In *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, 1987.

See Appendix B for suggestions on how to cope with older C++ compilers and with code written for C compilers. 1.2 Learning C++. [notes.learn].Â C++ was designed to be used in a traditional compilation and run-time environment, that is, the C programming environment on the UNIX system. Fortunately, C++ was never restricted to UNIX; it simply used UNIX and C as a model for the relationships between language, libraries, compilers, linkers, execution environments, etc. That minimal model helped C++ to be successful on essen-tially every computing platform. There are, however, good reasons for using C++ in environments that provide significantly more support. C++ Environment Setup - If you are still willing to set up your environment for C++, you need to have the following two softwares on your computer.Â Add the bin subdirectory of your MinGW installation to your PATH environment variable so that you can specify these tools on the command line by their simple names. When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line. I have a C++ application which receives stock data and forward to another application via socket (acting as a server). Actually the WSASend function returns with error code 10055 after small seconds and I found that is the error message. "No buffer space available. An operation on a socket could not be performed because the system lacked sufficient buffer space or because a queue was full".Â I tried to increase the sending buffer SO_SNDBUF using setsockopt function but the same problem still there. How can I solve this problem? is this related to receiver buffer? Sending details: For each complete message I call the send method which uses overlapped sockets. EDIT: Can someone give general guidelines to handle high frequency data in C++? c++ winsock. share | improve this question |.