Item 1: Switching Streams

ITEM 1: SWITCHING STREAMS — DIFFICULTY: 2

What's the best way to dynamically use different stream sources and targets, including the standard console streams and files?

1. What are the types of `std::cin` and `std::cout`?
2. Write an `ECHO` program that simply echoes its input and that can be invoked equivalently in the two following ways:

```
ECHO <infile >outfile

ECHO infile outfile
```

In most popular command-line environments, the first command assumes that the program takes input from `cin` and sends output to `cout`. The second command tells the program to take its input from the file named *infile* and to produce output in the file named *outfile.* The program should be able to support all of the above input/output options.

SOLUTION

1. What are the types of `std::cin` and `std::cout`?

The short answer is that `cin` boils down to:

```
std::basic_istream<char, std::char_traits<char> >
```

and cout boils down to:

```
std::basic_ostream<char, std::char_traits<char> >
```

The longer answer shows the connection by following some standard typedefs and templates. First, cin and cout have type std::istream and std::ostream, respectively. In turn, those are typdef'd as std::basic_istream<char> and std::basic_ostream<char>. Finally, after accounting for the default template arguments, we get the above.

Note: If you are using a prestandard implementation of the iostreams subsystem, you might still see intermediate classes, such as istream_with_assign. Those classes do not appear in the standard.

2. Write an ECHO program that simply echoes its input and that can be invoked equivalently in the two following ways:

```
ECHO <infile >outfile

ECHO infile outfile
```

The Tersest Solution

For those who like terse code, the tersest solution is a program containing just a single statement:

```
// Example 1-1: A one-statement wonder
//
#include <fstream>
#include <iostream>

int main( int argc, char* argv[] )
{
  using namespace std;

  (argc > 2
    ? ofstream(argv[2], ios::out | ios::binary)
    : cout)
  <<
  (argc > 1
    ? ifstream(argv[1], ios::in | ios::binary)
    : cin)
  .rdbuf();
}
```

This works because of two cooperating facilities: First, basic_ios provides a convenient rdbuf() member function that returns the streambuf used inside a given stream object, in this case either cin or a temporary ifstream, both of which are derived from `basic_ios`. Second, `basic_ostream` provides an `operator<<()` that accepts just such a `basic_streambuf` object as its input, which it then happily reads to exhaustion. As the French would say, "*C'est ça*" ("and that's it").

Toward More-Flexible Solutions

The approach in Example 1-1 has two major drawbacks: First, the terseness is borderline, and extreme terseness is not suitable for production code.

Guideline

Prefer readability. Avoid writing terse code (brief, but difficult to understand and maintain). Eschew obfuscation.

Second, although Example 1-1 answers the immediate question, it's only good when you want to copy the input verbatim. That may be enough today, but what if tomorrow you need to do other processing on the input, such as converting it to upper case or calculating a total or removing every third character? That may well be a reasonable thing to want to do in the future, so it would be better right now to encapsulate the processing work in a separate function that can use the right kind of input or output object polymorphically:

```
#include <fstream>
#include <iostream>

 int main( int argc, char* argv[] )
 {
   using namespace std;

   fstream in, out;
   if( argc > 1 ) in.open ( argv[1], ios::in  | ios::binary );
   if( argc > 2 ) out.open( argv[2], ios::out | ios::binary );

   Process( in.is_open()  ? in  : cin,
            out.is_open() ? out : cout );
 }
```

But how do we implement `Process()`? In C++, there are four major ways to get polymorphic behavior: virtual functions, templates, overloading, and conversions. The first two methods are directly applicable here to express the kind of polymorphism we need.

Method A: Templates (Compile-Time Polymorphism)

The first way is to use compile-time polymorphism using templates, which merely requires the passed objects to have a suitable interface (such as a member function named `rdbuf()`):

```
// Example 1-2(a): A templatized Process()
//
template<typename In, typename Out>
void Process( In& in, Out& out )
{
  // ... do something more sophisticated,
  //     or just plain "out << in.rdbuf();"...
}
```

Method B: Virtual Functions (Run-Time Polymorphism)

The second way is to use run-time polymorphism, which makes use of the fact that there is a common base class with a suitable interface:

```
// Example 1-2(b): First attempt, sort of okay
//
void Process( basic_istream<char>& in,
              basic_ostream<char>& out )
{
  // ... do something more sophisticated,
  //     or just plain "out << in.rdbuf();"...
}
```

Note that in Example 1-2(b), the parameters to `Process()` are not of type `basic_ios<char>&` because that wouldn't permit the use of `operator<<()`.

Of course, the approach in Example 1-2(b) depends on the input and output streams being derived from `basic_istream<char>` and `basic_ostream<char>`. That happens to be good enough for our example, but not all streams are based on plain `char`s or even on `char_traits<char>`. For example, wide character streams are based on `wchar_t`, and *Exceptional C++* [Sutter00] Items 2 and 3 showed the potential usefulness of user-defined traits with different behavior (in those cases, `ci_char_traits` provided case insensitivity).

So even Method B ought to use templates and let the compiler deduce the arguments appropriately:

```
// Example 1-2(c): Better solution
//
template<typename C = char, typename T = char_traits<C> >
void Process( basic_istream<C,T>& in,
              basic_ostream<C,T>& out )
{
  // ... do something more sophisticated,
  //     or just plain "out << in.rdbuf();"...
}
```

Sound Engineering Principles

All of these answers are “right” as far as they go, but in this situation I personally tend to prefer Method A. This is because of two valuable guidelines. The first is this:

Guideline
Prefer extensibility.

Avoid writing code that solves only the immediate problem. Writing an extensible solution is almost always better—as long as we don’t go overboard, of course.

Balanced judgment is one hallmark of the experienced programmer. In particular, experienced programmers understand how to strike the right balance between writing special-purpose code that solves only the immediate problem (shortsighted, hard to extend) and writing a grandiose general framework to solve what should be a simple problem (rabid overdesign).

Compared with the approach in Example 1-1, Method A has about the same overall complexity but it’s easier to understand and more extensible, to boot. Compared with Method B, Method A is at once simpler and more flexible; it is more adaptable to new situations because it avoids being hardwired to work with the iostreams hierarchy only.

So if two options require about the same effort to design and implement and are about equally clear and maintainable, prefer extensibility. This advice is *not* intended as an open license to go overboard and overdesign what ought to be a simple system; we all do that too much already. This advice is, however, encouragement to do more than just solve the immediate problem, when a little thought lets you discover that the problem you’re solving is a special case of a more general problem. This is especially true because designing for extensibility often implicitly means designing for encapsulation.

Guideline
Prefer encapsulation. Separate concerns.

As far as possible, one piece of code—function or class—should know about and be responsible for one thing.

Arguably best of all, Method A exhibits good separation of concerns. The code that knows about the possible differences in input/output sources and sinks is separated from the code that knows how to actually do the work. This separation also makes the intent of the code clearer, easier for a human to read and digest. Good sepa-