contents

credits & contacts

Editor:

John Merrells,
merrells@acm.org
808 East Dana St,
Mountain View,
CA 94041, U.S.A

Advertising:

Pete Goodliffe, Chris Lowe
ads@accu.org

Membership:

David Hodge,
membership@accu.org
31 Egerton Road
Bexhill-on-Sea, East Sussex
TN39 3HJ, UK

Readers:

Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Website: http://www.accu.org/

Membership fees and how to join:

Basic (C Vu only): £15
Full (C Vu and Overload): £25
Corporate: £80
Students: half normal rate
ISDF fee (optional) to support Standards work: £21
There are 6 journals of each type produced every year.
Join on the web at www.accu.org with a debit/credit card, T/Polo shirts available.
Want to use cheque and post - email membership@accu.org for an application form.
Any questions - just email membership@accu.org.

Editorial

This issue of Overload has been a struggle for us to put together. At the deadline we had only 11 pages of articles. The minimum we need is 24 pages. I wanted to just skip this issue, but the ACCU committee was determined that we should publish an issue, even if it was much shorter than usual. Thanks to the last minute efforts of a few individuals we actually managed to author, source, review, revise, and publish the rest of this issue within a week.

This is not sustainable. Over the past few years each issue has become harder and harder for us to fill with quality articles. This is not because we reject low quality articles, but because we receive fewer and fewer submissions.

The quality of the articles in this publication is due to the work of the Overload editorial board readers. They do not act as a content filter rejecting submissions. They read, comment, discuss, and re-read, re-comment, and re-discuss, until they feel that an article is up to the standard that the Overload readership has come to expect. Given the choice between filling space with an item of low relevance, or of dubious technical quality, and an empty space they go with the blank page every time.

There is no quality hurdle that an author needs to clear. We gratefully accept all submissions and guide the author through the development process from draft to published article. We offer far more advice and help then any author would get from a commercial journal.

Some of the content shortfall is due to regular contributors moving on to other things. I think that's great. The ACCU alumni are writing for commercial journals, writing books, speaking at conferences, being consultants, and contributing to open standards. We shouldn't be relying on them to fill the magazine. I like it that members can practice and hone their talents within the ACCU and then move on to greater things.

Note that the ACCU fees are not magazine subscription fees. They are membership fees. That's an important distinction. The ACCU is a community of peers gathered together to benefit from each other's experiences. If Overload always contains the same few voices we're only going to be hearing from a small subset of our community.

I'm sure you're tired of reading CVu and Overload editorials asking for articles so I'll try to be brief. You get out of this organisation what you put into it. You grow by challenging yourself. You really understand something if you can explain it to somebody else. You do have something interesting to say.

Your action item is to email me now. I want to hear what you want to read about in Overload magazine. I want to hear about what you want to write about.

John Merrells

Copy Deadline

All articles intended for publication in *Overload 53* should be submitted to the editor by January 1st 2003, and for *Overload 54* by February 14th 2003.

Note the earlier than usual deadline for *Overload 54* - this is to allow us to produce the April journals in time for the conference.

Build Systems

by Allan Kelly

In my last essay I discussed the structure of the directory tree containing source code, here I want to discuss the systems we use to build source code. The directory tree structure plays an important part in this. The structure we use to hold our source code, and the mechanism we use to convert that source code into programs are all part of the same problem.

Source code is the representation of everything you know about the problem and solution domain, creating it is a process of blood, sweat and tears, but unless you can turn it into executable programs it is worthless. Before it can generate revenue it must be built.

First and foremost we must be able to build our source code. This may seem obvious but there is nothing more depressing than being given source code and finding we can't build it. The `.dsp` won't load, or make bombs out immediately - or there just isn't a make file.

It helps to ask two questions:

- Given a new PC how long does it take to get it building our source?
- If we hire a new developer how difficult is it for them to build the system?

One company I encountered considered it a rite of passage for a new developer to build the server, this is quite adversarial and depressing to a new developer.

Objectives

Let's consider a few objectives for our build system:

- **Reliable**: no point in having a system that is slightly random.
- **Repeatable**: having a system that only works three out of four weeks is no good.
- **Understandable**: build systems tend to become increasing cryptic over time, especially if you use some of the more obscure syntax of make.
- **Fault aware**: things will go wrong with our build system, we don't expect fault tolerance but we can aim for fault awareness. When a build fails we would like as much information on why and how as possible.

There are more objectives we may like to add over time: multi-platform capable, automated, multiple build types (e.g. debug, release), fast and so on but let's start with a short list.

We need to think of the build system as part of the source code. Makefiles are as much part of your system as any other source code file, be it a `.cpp`, or `.hpp`. They should be subject to the same source code control as any other file. Your makefiles are essential to your applications.

And everyone should use the same build process, and the same makefiles. It is counterproductive if you using one set of makefiles and Joe using another set. What if you set the `/GX` options and he sets the `/O2` options? Subtle differences will emerge, subtle faults will appear - they will even appear and disappear at "random" intervals.

Subtlety is bad

If it is different make it obvious - Write it BIG

Unfortunately this leads to a contradiction in the build system. Developers want a system that is easy to use, integrates with our tools, is fast, and does the minimal rebuild possible. However, the build master wants a 100% repeatable process - no typing make again if it fails the first time - and their definition of "easy to use" is different from ours.

Where we have an overnight batch build we have slightly different objectives again: speed is less of an issue, but automation is paramount.

Some of these difference can be easily resolved (e.g. have the build master use the same build process as the overnight build, maybe even take a release candidate from the overnight build), other difference aren't reconcilable and compromise is needed.

The clean build

When developing a build system my first milestone is:

- **Given a clean machine, I should be able to install a set of packages from a given list, get files from source code control, and perform a build.**

If we can't do this, do we have any hope of ever building our software? Consider the alternatives. Many of us have been there, you start work at a new company and try and build the code. It fails, someone says "Yes, you need to install Python", and you install Python. It fails. Then someone says "Yes, you have to install version 2.1." Next time it fails: "Can't find `\boost\any.hpp`", after much searching you discover that everyone else has `BOOST_ROOT` set in the environment but has forgotten about it. And so on.

Once I can rebuild on a clean machine look to repeatability:

- **Automate the build to happen at 1am every night.**

(Does 1am sound OK to you? Or are your developers frequently working until 2am? Are you really sure you want them checking in when they are bleary eyed? I'm told that Apple used to have a rule against check-ins later than 10pm. Perhaps more problematic than insomniac developers is what to do when your development spans multiple time zones, or when the build takes a long, long time.)

If you can't automate your build how do you know it is repeatable? How do you know it doesn't rely on Pete blowing the tree away every day at 3pm? Even if you document your build process ("install Visual C++, download Boost, build the utils library, and build the application") how do you know the document is accurate and up to date? The process should be the documentation.

While creating a repeatable build process you need to strive to make the build aware of potential problems. Have it issue meaningful messages if a package is missing, have it log output to a file, even comment the makefiles!

With these elements in place we are half way to meeting the objectives outlined above.

Environment variables

Variables, as always, are the key to capturing variance and the same is true with environment variables. In the build system we use them to express configuration details (where to find the external tree), options (build debug or release), set defaults (developers will default to debug builds, build master will default to release and the overnight build should do both) and communicate between the makefiles.

Unix folks have always been friendly with environment variables while Windows people, on the other hand, see environment variables as a hangover from the DOS day - who needs them when you have the registry? However, they are just as important on Windows machines as Unix machine for two reasons. First, they communicate with the compiler and make much better than the registry, and second, they provide a common mechanism so scripts can be used on Unix too.

You can use an environment variable like `PROJECT_ROOT` almost as well in Windows as in Unix. The project configuration in the Microsoft IDE does its best to confuse you, the "project settings" dialog is even more fiddly than the control panel and uses the `$(PROJECT_ROOT)` syntax like make and bash, and unlike the Windows command line `%PROJECT_ROOT%` syntax.

Here is a section of my current `.bashrc` file:

```
export PROJECT_ROOT=f:/xml
export EXTERNAL_ROOT=f:/external/lib
export BOOST_ROOT=f:/external/boost_1_27_0
export GSOAP_ROOT=$EXTERNAL_ROOT/soapcpp-win32-
                                          2.1.3
export BERKELEYDB_ROOT=$EXTERNAL_ROOT/db-4.1.15
```

I'm running the Cygwin toolkit for Windows which provides a Unix-like environment. This gives me a rich shell environment - far superior to the command shell provided by Microsoft. It is also, give or take a drive reference, compatible with the bash shell on my Linux box. Even when I'm not engaged in cross-platform work I run a Unix shell, currently Cygwin bash but MKS also do a good Korn shell.

Returning to the `.bashrc` section, I'm mostly defining variables for third party tools and libraries in terms of previous variables. Not always, because (a) I'm a little bit random myself, (b) I sometimes try new versions of things and they may live in a different place until I decide what to do with them. The important thing is I have flexibility.

Now, take a look at a section from one of my make files:

```
ifeq ($(PROJECT_ROOT),)
$(error Error: PROJECT_ROOT not set)
endif

ifeq ($(EXTERNAL_ROOT),)
export EXTERNAL_ROOT=$(PROJECT_ROOT)/external
$(warning Warning: EXTERNAL_ROOT not set \
             using default $(EXTERNAL_ROOT))
endif

ifeq ($(BOOST_ROOT),)
export
BOOST_ROOT=$(EXTERNAL_ROOT)/boost_1_26_0
$(warning Warning: BOOST_ROOT not set using \
                     default $(BOOST_ROOT))
endif
```

The makefile will check to see if it can find the important variables. If it can't find `PROJECT_ROOT` then all bets are off, give up now. If it can't find `EXTERNAL_ROOT` then make a guess, as it happens the guess it will make here is bad but as long as I've set in my environment it doesn't really matter. What is important is that the system is aware of a potential problem; it issues a warning but attempts to carry on.

Next it moves on to check a long list of third party sources. Again, if it can't find them it makes an educated guess and gives me a warning. In this way I don't need to set a large number of variables to get the build up and running, just a couple of key ones, although I have the option to do things differently, to put my external libraries somewhere else.

Importantly this is self documenting, if I need to know what third party packages I should download I can look at the makefile, the same file which will be used to find them when I do the build, the makefile is the documentation.

Secrets of Make

The original Unix make program must have the most cryptic syntaxes developed before the advent of Perl. Over the years it has put many developers off using it - and sent many running into the arms of Microsoft and their crippled make system using `.dsw` and `.dsp` files.

There are two reasons to ditch your Microsoft build system in favour of real make: first it is massively more flexible and powerful (although it is also easier to shoot yourself in the foot). Secondly: it allows you to construct true hierarchies, and perform recursive builds. In contrast Microsoft provides only one level of depth - that of grouping `.dsp` files in a `.dsw`.

There are several secrets about make you should be aware of before continuing:

- GNU make (sometimes called gmake) is a very different creature to the original Unix make, the old syntax is still there but there is new simpler syntax which makes it far easier to program.
- Rules and variables (confusingly called macros) are the key to a good makefile.
- Rules specify targets.
- Variables can be picked up from your shell environment, set on the command line invoking make, or set within the make script.
- Rules can be specified in terns of environment variables. If the environment variables change, the rules change, this includes targets.
- By default, make will execute the first rule (target) it sees when processing a makescript. Unless another target is specified on the command line only the first rule will be processed, of course, the first rule may cause other rules to be invoked. (By convention this rule is usually called all.)
- The order of rules is not important - exceptions being the first rule, duplicate rules and includes.
- You can include other makefiles in your makefile.
- If you include a file that does not exist, make will turn in on itself to see if it knows a rule to create the file you want included before it executes the first rule.
- Except rules invoked to find or generate include files, no rules will be executed until all includes have been processed, then the first rule encountered will be processed.
- Make comes with a set of in built, default, rules which are useful for small programs but not much else. Many of these rules are legacy rules (e.g. compiling `.pas` files) and it is usually easier to disable them (with `.SUFFIX`) and specify your own.
- If things are too difficult to do in make you can always run a shell script from within make.

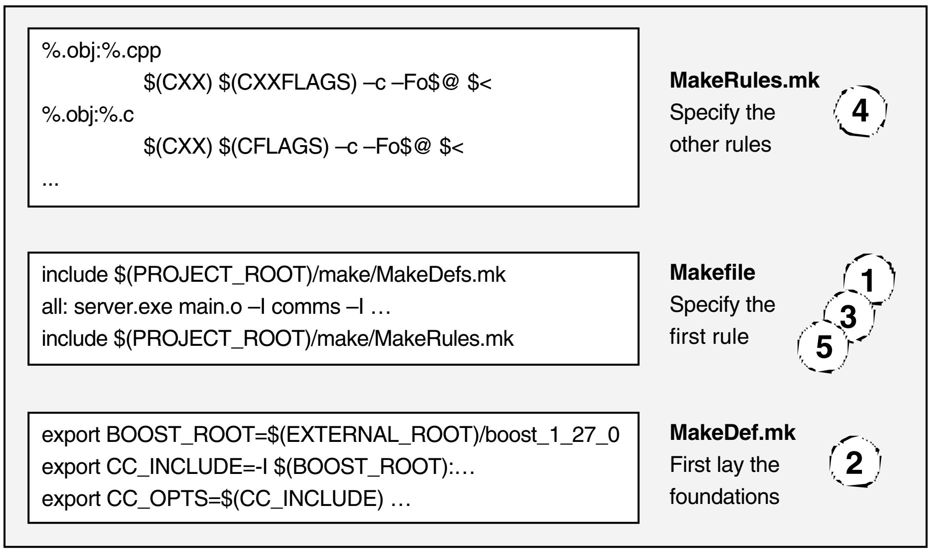
Everything I talk about here is specifically about GNU make. Your system may come with another make program, some of what I say will still be relevant but not all. Since there as almost as many make programs as there are compilers it is impossible to cover them all.

Before continuing let me point out that when I speak of build systems, I'm talking about the whole build process, source code extraction, build scripts, process and make files. When I talk of make systems, I'm talking specifically about the make scripts which run the compiler and other tools.

Make sandwiches

Make systems usually end up as a three layer sandwich. The first layer specifies the environment variables to be used, this sets the foundations of what is to come. The second layer is thin but essential: it specifies the first rule. The third layer specifies everything else, that is: the other rules needed.

We want our make system to be easy to use, ideally we should just type make, and everything will be built. Or rather, when we



```
%.obj:%.cpp
        $(CXX) $(CXXFLAGS) –c –Fo$@ $<
%.obj:%.c
        $(CXX) $(CFLAGS) –c –Fo$@ $<
...
```

```
include $(PROJECT_ROOT)/make/MakeDefs.mk
all: server.exe main.o –l comms –l …
include $(PROJECT_ROOT)/make/MakeRules.mk
```

```
export BOOST_ROOT=$(EXTERNAL_ROOT)/boost_1_27_0
export CC_INCLUDE=-I $(BOOST_ROOT):…
export CC_OPTS=$(CC_INCLUDE) …
```

Figure 1 Makefile sandwich

type make, we want everything in our current directory to be made, and if necessary, anything below the current directory.

If no file is specified on the command line make will attempt to process the file `Makefile` - there are several other names it will try before or after this, e.g. `GNUmakefile`, so consult your documentation.

Since the environment variables and rules are fixed it is this element of the sandwich that changes, the makefile in our immediate proximity is the filling in the sandwich, and it is here that execution begins, this is the interesting bit between two bits of bread. Most fillings will look very similar; they pull in the definitions, specify the first rule, and then pull in the other rules.

From a command line, with our current directory containing the makefile shown in figure 1, processing runs something like this:

1 Make is invoked and looks for `Makefile`, loading this file it starts to process
2 First include `MakeDefs.mk` - this pulls in additional variable definitions. This file is not local so we must specify where to find it. If `PROJECT_ROOT` is not set it will fail. Importantly, it cannot contain any rules as we want `all` to be the first rule make encounters.
3 Encounter rule `all` and remember it is the first rule we have encountered.
4 Include `MakeRules.mk`, store rules "as is", and only expand environment variables when the rules are encountered.
5 Execute the first rule, `all`, and any that are implied by it, when complete end.

Dependencies

One group of rules we need to specify is file dependencies, we need rules which say `Server.cpp` depends on `Server.hpp`, `Tcp.hpp`, `Utils.hpp` and so on. Given that a typical `.cpp` file may specify half a dozen or more `.hpp` file this can be a fairly large task so we don't want to write these rules ourselves. (If we don't specify these rules then the compiler won't get to recompile `Server.cpp` after a change to `Server.hpp`.)

Luckily help is at hand. GCC provides the several command line options (`-M`, `-MG`, `-MM`, etc.) to generate this information. Visual C++ provides the `/FD` switch for something similar - although I'll freely admit I've never actually used this in anger, I have looked at it well enough to say: it is not well documented and doesn't work like the GCC options.

My current make system uses makedepend, which originated at Tektronix and MIT as part of the X system. This comes as standard on some Unix versions, if you don't have it already, or are using Windows it is easy to track down a version on the net. Using the same program on Windows and Unix means there is no need for the makefiles to differ.

The default behaviour of makedepend is to append the generated rules to any makefile it finds in the current directory. This complicates matters as files will appear to change when there is no substantive change. A better way is to use makedepend to generate an additional makefile and include this. Since make will look at its own rules for any file it can't include this becomes almost trivial.

In the `MakeRule.mk` file we can add a new rule:

```
export DEP_FILE=depends.mk
$(DEP_FILE):
  @echo \# Generated dependencies file
                      >$(DEP_FILE)
  @echo \# Never check this in to source
                      control >>$(DEP_FILE)
  @echo \# Dependencies follow >>$(DEP_FILE)
  makedepend -f $(DEP_FILE) \
    -Y $(CC_OPTS) $(CC_INCLUDE) $(SRCS) \
    -s"# Dependencies follow" > /dev/null 2>&1
  grep ".o:" $(DEP_FILE)
            | sed "s/\.o/\.obj/g" >>$(DEP_FILE)
```

Where `SRC` is a list of the source files to be processed, `CC_OPTS` specifies the compiler options and `$(CC_INCLUDE)` is a list of include paths.

In our local makefile we can add a new include after we include `MakeRules.mk`:

```
include $(DEP_FILE)
```

When make can't find `depends.mk` it will now know how to generate it from the rule in `MakeRules.mk`. Be warned: running makedepend can be time consuming.

The example rule has an additional grep command run once the dependencies have been generated. This is to cope with platform differences. On Unix object files are normally `.o` files, while on Windows they are normally `.obj` files. Being a Unix program makedepend will generate the rules in terms of `.o` files, for a Windows compile these rules are pointless, there will never be an `.o` file so the rules are never used. Unfortunately this means there are no rules for `.obj` files. Hence the grep and sed to create Windows equivalent rules. On a Unix system, the reverse is true and the `.obj` rules will be ignored.

Recursive and clean

Although not shown here it is advisable to include some house keeping targets. Typically a clean target will delete everything that has been generated, `.obj`'s, `.exe`'s, `.lib`'s, etc. This can be drastic so some other targets like `cleanlib` and `cleanobj` can be included too which do lesser cleanups.

When we organise our directories as hierarchies we frequently end up with makefiles that don't run the compiler at all, instead they simply call several makefiles in sub-directories. There are two types of node in our hierarchy tree. Leaf nodes that contain source code and makefiles to compile the source, and intermediate nodes that serve to collect leaf nodes together.

Processing can be speeded up in these cases by skipping the inclusion of `MakeRules.mk`.

Care needs to be taken to ensure that these intermediate directories pass through targets like `clean`. The `clean` rule in an intermediate makefile will look quite different to one in a leaf node, although both must share the same name.

To this end I've recently separated my `clean` rules into `MakeClean.mk` and `MakeCleanRecursive.mk`. `MakeClean.mk` can be included from `MakeRules.mk` so is available in all leaf nodes. Intermediate nodes don't include `MakeRules.mk` but instead include `MakeCleanRecursive.mk` to pass the target on to sub-directories.

Sometimes when recursing it is easier to use a bit of shell script, since make is happy to run a shell we can write:

```
clean: cleanlib
for i in $(MODULES); do $(MAKE) -C $$i clean;
                                            done
```

The fact that we can recurse into our tree is thanks to our directory structure. If we had a simple flat structure with everything, applications and libraries hanging off a single root, things would be more complicated.

Get source code

One of the tasks originally envisaged by make's designers was getting source code from source code control before building; indeed, there are built-in rules for SCCS and RCS. Usually though it is better to treat the extraction of source code as one step, and the building as another even if this means having a shell script to run one command and then make.

This makes it easier for developers to work with make. When developing code you are unlikely to want make getting the latest version of files as soon as someone has checked them in. This is particularly true if using CVS where the very file you are working on may change if this happens.

Separating the get from the build means developers can choose when to update their tree. Simply running a build will not result in source code updates. Nor will we incur the time delay as the source control system checks for updates and retrieves any.

Although developers will usually update their tree at convenient internals to suit themselves batch builds should always build a fresh tree. It is a good test of a system to be able to completely delete a tree and rebuild it from scratch. Indeed, this is worth while exercise for developers to ensure they don't inadvertently forget to check-in some file.

Make Miscellany

- It is usually a good idea to disable the built-in rules. This can be done with the `.SUFFIX` directive - which itself has subtly changed its use over the years. Disabling the built-in rules assists with debugging (`make -d`) and marginally improves performance.
- I've taken to including a `help` target (`make help`) in my systems. This normally takes the form of a `help` rule in a `MakeHelp.mk` file. Normally this will simply validate any external variables.
- The list of source files used for building is normally referenced in several places. To save duplication it usually helps to lists these in one variable, e.g.

```
SRCS =  Configurator.cpp Interface.cpp main.cpp
```

- Frequently we want to create different build types for our compilations. Say a debug version and a release version. This is somewhat tricky as make expects to work in the current directory. My solution is to massage the filenames, this can be done with variable definitions given the list of source files:

```
# first change the suffix on the filenames
ifeq ($(COMPILER),msvc)
export OBJNAMES = $(SRCS:.cpp=.obj)
export OBJNAMES += $(CSRCS:.c=.obj)
endif
ifeq ($(COMPILER),g++)
export OBJNAMES = $(SRCS:.cpp=.o)
export OBJNAMES += $(CSRCS:.c=.o)
endif
# now append the build directory
export OBJS = $(addprefix $(BUILD_TYPE)/,
                                $(OBJNAMES))
```

  Where `BUILD_TYPE` would normally be set to `Debug` or `Release` in the environment.
- Remember to create directories before you try and put anything in them, this can be done with a rule for the directories themselves:

```
$(OUTPUT_DIR):
    mkdir -p $(OUTPUT_DIR)
$(BUILD_TYPE):
    mkdir -p $(BUILD_TYPE)
```

Generate what you can

Sometime we don't want to write in our chosen programming language. Some things are easier to represent in other forms, for example we use IDL to describe object interfaces, or we may want to encode the contents of a text file within source code, e.g. error messages which are defined externally, or localised message files.

In these cases it is useful to generate C++ code from another source and then compile it. Taking our error messages file this may contain an error number, sub-system code and a text message. Using a make rule we may specify a rule for converting text files to `.cpp` files, the rule could run a Python script. Once generated we would then compile the resulting `.cpp` file as normal.

The side box "Generating build numbers" provides an example of how we can use Python to generate C++ source code.

Just the beginning

Once you have your make system up and running there are a whole host of enhancements you may want to consider. Rather than go into details here are some ideas:

- Always create a log file of messages from the batch build - when it runs at night you need to know what happened.
- Make the build log public, e-mail to developers.
- Add automated tests to the build process - this is a good place to start automating your tests.
- Add a "package" target to the make system to collect all the files needed for an install. For Unix you may like to tar these up, for Windows you could automate your installer creation.

The example system I included with my article Writing Extendable Software (Overload 49) includes a simple make system along these lines and is available at `http://www.allankelly.net`. Time permitting I may upload another example.

Finally

To those who use Microsoft `.dsp` project files this may all seem excessively complex. However, `.dsp` files do not scale and do not work well on larger projects. Nor are they as rigorous and adaptable as makefiles.

Time spent developing a directory structure in depth, and a rigorous build system will result in a better defined project which can absorb additions and expansion more easily than one which is held together with sticky-tape and rubber bands. The rigors of this approach force problems to the surface.

By developing these aspects of the project fully we create strong logistics support for our development. This is all part of our strategy to create a software development process which can produce quality software, where every activity in the development process interlocks and supports the others.

Allan Kelly
Allan.Kelly@bigfoot.com

Generating build numbers

In addition to version numbers it can be useful to individually number each build. To do this we need a mechanism for storing the last build number, incrementing it, restoring the incremented number and incorporating the number within our build.

First, we want a file we can check out of source control, change and check back in with the revised version number. This may also be a useful place to keep various other pieces of information we don't want to hard code, e.g. the version number or company name. And we would like the file to be plain text so we can change it easily.

Ideally, we want our file to look something like:

```
BuildNumber = 126
VersionNum = "0.1 alpha"
Copyright = "(c) Jiffy Software 2002"
```

As luck would have it this is actually valid Python so we can save this as `Version.py`. If we had written our file as C++ we would need a complex parsing algorithm to read the file, increment the build number and re-write the file. Since it is Python we can write another Python script which includes the file and treats these variables as, well, variables.

So, we write a script called `GenVersion.py` that looks a bit like this:

```
import Version
version.BuildNumber = version.BuildNumber +1
...
# rewrite the version file
ver_output = open("version.py", "w")
ver_output.write("BuildNumber = " + \
              str(version.BuildNumber) + "\n")
...
# write a C++ file
cpp_output = open("Version.cpp", "w")
cpp_output.write("std::string \
                 Version::BuildNumber() {\n");
cpp_output.write("\treturn \"" + \
           str(version.BuildNumber) + "\";\n")
cpp_output.write("}\n\n")
```

Next you will want to add a rule to your makefile to execute the script:

```
Version.cpp: version.py
bash -c "python GenVersion.py"
```

This example creates `Version.cpp` so you will need a `Version.hpp` but as this rarely changes it doesn't need regenerating. You may like to generate other files with this information, say a Windows resource script, or write the build number to the build log.

In this example I've omitted niceties such as dealing with the other variables, closing files and such. You can see the complete scripts at my web site, `http://www.allankelly.net/writing`. The most important nicety that is missing is an option not to bump up the build number.

Hang on you say, "Aren't we talking about incrementing the build number? Why would we want not to do it?" Actually it is important to ensure the build number is only bumped when we want it bumped. If every time a developer tried to build source code the build number would run wild and be useless.

The trick is to only bump the build number when we are doing an official build. Usually this means a batch build. Developers building on their local machine don't count. So, we make the default case the current build number from `Version.py`, and add a different rule to bump the build number, so we get:

```
all: Version.cpp
bumpBuild: all
   bash -c "python GenVersion.py Bump"
Version.cpp: version.py
   bash -c "python GenVersion.py NoBump"
```

By default Make will execute the `all` rule, which will only rebuild `Version.cpp` if `Version.py` is newer, in which case the build number will be unchanged. If however, we execute the `bumpBuild` rule (which we will do for an official build) then we execute the second rule, which will always generate `Version.cpp` and in the process increment the build number.

Deciding when to increment the build number can become a more vexed question still. Suppose you bump it for every over night build on Windows, suppose you now introduce a nightly Linux build. Should they use the same number? Should they even use the same number sequence? Maybe the new build should start from one.

Generated files like this should not be checked into source code control. We can recreate them at any time and they only clutter up source control with frequent, minor changes. We also need to take care not to make then read-only at any time as this may cause a build error. Finally, if your build system has a "`make clean`" option we need to remember to delete any intermediate files we may have generated.

We can use these same principles of code generation to create other elements of our system. Examples include:

- Internationalised language resources can be built from plain text files, thus allowing translators to work with something friendlier than C++ or a proprietary file format.
- Error and other information messages can be generated from text files. Tab delimited files specifying error number, sub-system and message text can be turned into C-arrays or even C++ exception classes.
- SQL can be customised to different target databases.
- Install scripts which only install the components that are built.

Like template programming we have an option to vary what is compiled into a program.

Implementing the Observer Pattern in C++ - Part 2

by Phil Bass

In part 1 of this article I presented an Event/Callback library intended to support the Observer pattern and hinted that it had some limitations. The library was based on the following Event class template:

```
template<typename Arg>
class Event {
public:
  // Iterator type definition.
  typedef ... iterator;

  // Destroy an Event.
  ~Event();

  // Attach a simple function to an
  // Event.
  template<typename Function>
  iterator attach(Function);

  // Attach a member function to an
  // Event.
  template<class Pointer,
           typename Member>
  iterator attach(Pointer, Member);

  // Detach a function from an Event.
  void detach(iterator);

  // Notify Observers that an Event has
  // occurred.
  void notify(Arg) const;

private:
  ...
};
```

Listing 1 - The Event<> class interface

In this version of the library an Event is essentially a list of polymorphic function objects (callbacks) owned by the Event. The `attach()` functions create a callback and add it to the list; the `detach()` function removes a callback from the list and destroys it. The `notify()` function simply calls each callback in the list passing a single parameter.

Logic Gates

The company I work for supplies industrial control systems. These systems are based on general purpose hardware components and highly configurable software. For example, we build "pods" that are part of underwater pipeline repair tools. The pods are designed to withstand conditions on the sea bed, but they contain general purpose I/O boards. As far as the software is concerned a pod is little more than a collection of digital and analogue inputs/outputs. Pods can be used to monitor and control tools for lifting and lowering sections of pipeline, cutting the pipe, welding pipes together and all sorts of ancillary operations. Different tool sets are used for different jobs and the software has to be configured accordingly. The configuration information is held in a database.

The database stores information that defines the control logic. For example, the data may indicate that a particular button on a control panel turns on a lamp or controls a pump. At present this is rather inflexible, so we considered using arbitrary networks of logic gates as a more general solution. The database would store these networks in the form of tables for each of the basic types of logic gate (NOT, AND, OR), plus a table specifying connections between the inputs and outputs of these logic gates. (Analogue inputs and outputs are not considered, here, but a similar mechanism can be envisaged for them.)

On start-up the control system software would create some logic gate objects and connect their inputs and outputs as specified in the database. The natural way to implement the connections was to use our existing Event/Callback library, which is based on the Event class template sketched in Listing1. An output is an Event and an input is a function that can be attached to such an Event.

A Worry

As always, there is a down side to this design. Because AND, OR and NOT gates are very simple a large number of them may be needed for practical control systems. And, because the software only "sees" an arbitrary network, it is not possible to control the complexity of connections by using a hierarchical data structure. The logic gates almost have to be stored as simple collections. And collections require value semantics. Unfortunately, the Event classes, and hence the logic gates that contain them, don't have the required semantics - in particular, they can not be copied safely.

There are two well-known ways to resolve this problem. Either the Event classes are changed so that they can be safely stored in containers or some sort of value-like handles to Events are stored instead of the Events themselves.

Copyable Events

It is fairly straightforward to change the Event template so that Events can be stored in standard containers. Objects in standard containers are required to be Assignable and CopyConstructible, so implementing a suitable copy assignment operator and copy constructor will do the trick. Since an Event owns its callbacks, making a copy involves cloning the callbacks, which can be done using the "virtual constructor" technique described in [1].

So, yes, we can make Events copyable, but is it a good idea? Well, not really, because it creates another, less tractable, problem. Consider the following scenario:

1 An Event is added to a container.
2 A callback is attached to the Event.
3 Another Event is added to the container.
4 The callback is detached from the original Event.

The program in Listing 2 illustrates this scenario.

With the implementation of `std::vector` that I am using, step 4 fails because step 3 invalidates the iterator created in step 2. In this case, the iterator is invalidated when the container it points into (an Event) is moved from one memory location to another. C++ doesn't provide a mechanism for defining 'move' semantics, so the vector copies the original Event and destroys the original, leaving a dangling iterator.

It is the application program's responsibility to avoid this problem. Possible fixes include: reserving space in the vector for

```
#include <vector>
#include "Event.hpp"

void f(int) { return; }

int main() {
  std::vector< Event<int> > events;
  events.push_back(Event<int>());
                                 // Step 1
  Event<int>::iterator i0 =
          events[0].attach(f);  // Step 2
  events.push_back(Event<int>());
                                 // Step 3
  Event<int>::iterator i1 =
                     events[1].attach(f);
  events[1].detach(i1);
  events[0].detach(i0);          // Step 4
  return 0;
}
```

Listing 2 - Invalidating iterators

all the events before attaching their callbacks; using a container whose `push_back()` function doesn't move existing elements; and attaching the callbacks only after all events have been added to the vector. In the tiny program shown here we can simply swap steps 2 and 3, but in general there may not be an easy fix.

There is one other avenue we might explore in our attempt to store Events in standard containers without placing a burden on the client code. Suppose we change the Event classes so that they keep track of the iterators returned by `attach()`. Then, when an event is moved all iterators pointing to that event can be adjusted accordingly. But the appropriate 'move' semantics must be implemented using the copy constructor, assignment operator and destructor. These functions must adjust iterators pointing to the original Event when it is moved, but not when it is merely copied, assigned or destroyed. I imagine this can be achieved, but it certainly makes the Event classes less efficient and less cohesive. There has to be a better way.

Copyable Event Handles

If making Events copyable gets us into murky waters, perhaps we should be using non-copyable events and accessing them via copyable handles. The handles can be stored in standard containers and the Event iterators will remain valid when the handles are moved.

The Boost [2] shared pointer is a suitable candidate for a copyable Event handle. It is a smart pointer template with shared ownership semantics. All shared pointers pointing to the same target object share ownership of that object, so that the target object is destroyed when its last shared pointer is destroyed.

Using this technique the code in Listing 2 becomes the program in Listing 3.

Although this solves the copyability problem, it comes at a cost. Events are now allocated on the heap, the shared pointers have some house-keeping to do and there is an extra level of indirection involved in all Event accesses. Whether that cost is affordable depends on the application, but it has a "bad smell" (in the Extreme Programming sense). The purpose of an Event is to allow callbacks to be attached and there's nothing about this that suggests Events should be stored on the heap.

```
#include <vector>
#include <boost/shared_ptr.hpp>
#include "Event.hpp"

void f(int) { return; }

typedef boost::shared_ptr< Event<int> >
                                  Handle;

int main() {
  std::vector<Handle> events;
  events.push_back(Handle(new Event<int>()));
  Event<int>::iterator i0 =
                  events[0]->attach(f);
  events.push_back(Handle(new Event<int>()));
  Event<int>::iterator i1 =
                  events[1]->attach(f);
  events[1]->detach(i1);
  events[0]->detach(i0);
  return 0;
}
```

Listing 3 - Using copyable handles.

Asking the Wrong Question?

In my experience, if a neat and tidy solution seems elusive it's usually because we're trying to solve the wrong problem. So, let's look at the problem again and try to understand it better.

What is it about the Event classes that makes them so uncooperative? It is simply that an Event copies its callbacks when the Event itself is copied. It does so because it owns the callbacks. Does it need to own its callbacks? Well, no, it doesn't, so let's see what happens if we remove the responsibility for creating and destroying callbacks from the Event classes.

Firstly, the Event classes become a lot simpler - little more than a list of pointers to abstract functions. Listing 4 shows a reasonable implementation of the simpler Event. Note that the `std::` prefix has been omitted (and an unnecessary `typedef` introduced) to simplify the layout of the code on the printed page.

```
// Abstract Function interface class.
template<typename Arg>
struct AbstractFunction {
  virtual ~AbstractFunction() {}
  virtual void operator() (Arg) = 0;
};

// Event class template.
template<typename Arg>
struct Event :
          list<AbstractFunction<Arg>*> {
  void notify(Arg arg) {
    typedef AbstractFunction<Arg> Func;
    for_each(begin(),
             end(),
             bind2nd(mem_fun(
                &Func::operator()),arg));
  }
};
```

Listing 4 - Revised Event class template.

```
// Callback class template.
template<typename Arg, typename Function>
class Callback : public AbstractFunction<Arg> {
public:
  Callback(Function fn) : function(fn) {}
private:
  virtual void operator() (Arg arg) { function(arg); }
  Function function;
};

// Event/Callback connection.
template<typename Arg>
class Connection {
public:
  template<typename Fn>
  Connection(Event<Arg>& ev, Fn fn)
    : event(ev), callback(ev.insert(ev.end(),
      new Callback<Arg,Fn>(fn))
    {}
  ~Connection() {
    delete (*callback);
    event.erase(callback);
  }
private:
  Event<Arg>&          event;
  Event<Arg>::iterator callback;
};
```

Listing 5 - The Callback and Connection templates.

In this version of the Event classes I have omitted the `attach()` and `detach()` functions because I feel the `std::list` member functions already do an adequate job. In fact, the full splendour of the `std::list` interface has been made available to clients of the Event classes by using public inheritance.

Making Connections

Having removed the responsibility for creating and destroying callbacks from the Event classes, we can either invent something else for that purpose or pass the buck to the client code. I propose to offer the programmer a choice. The new Event/Callback library will provide a Connection class template that manages callbacks itself; alternatively, the client code can create and destroy its own callbacks. Sometimes we can have our cake and eat it!

Listing 5 shows the Connection template and the Callback template used in its implementation.

The Connection classes are designed for the "Resource Acquisition Is Initialisation" technique described in [1]. A callback is created and attached in the Connection's constructor and (predictably) the callback is detached and destroyed in the Connection's destructor. Instead of calling `attach()` and

```
// Old Event/Callback example
#include <iostream>
#include "Event.hpp"
using namespace std;
...

// A callback function.
void print(Button::State state) {
  cout << "New state = "
       << state
       << endl;
}

// A sample program
int main() {
  Button button;
  cout << "Initial state = "
       << button.state
       << endl;
  Event<Button::State>::iterator i =
    button.stateChanged.attach(print);
  button.press();
  button.release();
  button.stateChanged.detach(i);
  return 0;
}
```

```
// New Event/Callback example
#include <iostream>
#include "Event.hpp"
using namespace std;
...

// A callback function.
void print(Button::State state) {
  cout << "New state = "
       << state
       << endl;
}

// A sample program
int main() {
  Button button;
  cout << "Initial state = "
       << button.state
       << endl;
  Connection<Button::State>
  connection(button.stateChanged,
             print);
  button.press();
  button.release();
  return 0;
}
```

Listing 6 - Sample program.

detach() the client program creates and destroys a Connection object.

Copying Connections

The Connection classes as shown here suffer from the same disease as the old Event classes - they can not be copied safely. However, an event with two connections to the same callback would invoke the callback twice each time the event occurs and it's difficult to see what use that might be. It is tempting, therefore, to make the Connection classes non-copyable (by deriving them from the boost::noncopyable class, for example). On the other hand, it might very well be useful to store Connections in standard containers, and for that they must be copyable.

One way to make Connections copyable is to have them store a shared-pointer to a reference-counted callback and provide appropriate copy constructor and copy assignment functions.[1] Doing so opens up the possibility of iterators being invalidated, but this is no longer a problem for the Event/Callback library. It was a problem in the old library because it required the client code to store the iterator returned by attach() and supply it to the detach() function. There is no such requirement for Connections.

```
// Callback as part of an Observer
class Observer : public
AbstractFunction<Button::State> {
public:
  Observer(Event<Button::State>& e) : event(e) {
    callback = event.insert(event.end(), this);
  }
  ~Observer() { event.erase(callback); }

private:
  virtual void operator() (Button::State state) {
    cout << "Button state = " << state << endl;
  }
  Event<Button::State>&          event;
  Event<Button::State>::iterator callback;
};
```

Listing 7 - Callback as part of an Observer

Object Lifetimes

Typically, a callback will invoke a member function. It is important, therefore, for the callback's target object to exist when the callback executes. Similarly, an Event must continue to exist until all its Connections have been destroyed because the Connection's destructor will erase a pointer from the Event. I **think** these restrictions are best treated as requirements imposed on the client code by the library. An alternative approach is to maintain enough information within the library to handle these situations internally. The Boost.Signals library [2] and the SigC++ library[2] seem to offer this type of full lifetime management.

Sample Code

Listing 6 illustrates how the new version of the Event/Callback library is used. It shows a snippet from the sample program given in Part 1 of this article, slightly modified to show an explicit detach() call and to fit into a two-column page layout. The code in the left column uses the original version of the library in which callbacks are owned by the Event; the right column shows the new version in which the callbacks are owned by a Connection object. The difference is minimal. Where the old code had an attach() call, the new code creates a Connection object; and where the old code had an explicit detach(), the new code uses the implicit destruction of the Connection object. In this simple program the benefit of the new design is not very striking, but it should make the Logic Gates program a whole lot easier to write (if we ever find the time to implement it).

1 Writing a copyable Connection class is left as an exercise for the reader.

2 Sorry, I don't have a reference to hand for the SigC++ library.

Roll-Your-Own Connections

The Connection classes create callbacks on the heap. In cases where callbacks are better stored as local objects or as part of larger objects the programmer can use the Callback template directly. Listing 7 shows an example in which an Observer is its own callback.

Conclusion

The Event/Callback library described in part 1 of this article seemed to serve its purpose well. Experience has shown, however, that there are circumstances where it doesn't live up to its promise. This is called "learning" and I believe it illustrates once again that software development is a young technology.

More experience is needed with the new Event/Callback library before we can be confident that it, too, is not flawed, but I'm fairly confident that the new is better than the old.

Phil Bass
phil@stoneymanor.demon.co.uk

References

1 Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, ISBN 0-201-889554-4.
2 See http://www.boost.org/

From Mechanism to Method - Total Ellipse

By Kevlin Henney

Introduction

The interface to an object indicates what you can do with it, and what you can do with an object is a matter of design. C++ supports different kinds of interfaces, from the explicit compileable interface of a class to the more implicit requirements-based interfaces associated with the STL. Interfaces can be named and organized in terms of one another, again a matter of design.

The most common form of interface relationship is that of substitutability, which, in its popular form, is associated with the good use of inheritance – if class `B` is not a kind of a class `A`, `B` should probably not publicly inherit from `A`. More generally, one type is said to be substitutable for another if the second satisfies the interface of the first and can be used in the same context. Satisfaction is a matter of direct realization, where all the features of expected of an interface are provided, or specialization, where new features may be added, existing features constrained, or both.

Not all specialization is inheritance and not all substitutability is inheritance based [Henney2000a, Henney2000b, Henney2000c, Henney2001]. Substitutability acts as a more general guideline for the use of inheritance, conversions, operator overloading, templates, and `const-volatile` qualification.

Substitutability is relative to a given context or use. Normally the context and use are implied, so that statements about substitutability come out as absolutes. For example, when a developer says that instances of class `B` may be used where `A` is expected, this is normally a shorthand for saying that pointers or references to `B` may be used where pointers or references to `A` are expected. It is unlikely that copy and slice were intended, especially if `A` is an abstract class.

The idea that a `const`-qualified type is effectively a supertype of a non-`const`-qualified type was introduced in the last column [Henney2001]. Each ordinary class can be considered – with respect to `const` qualification – to have two interfaces, one of which is a specialization of the other. This interesting and different perspective can influence how you go about the design of an individual class, but does it have any other practical and tangible consequences? A single class can support multiple interfaces, but by the same token you can split a class into multiple classes according to interfaces. Consideration of mutability does not simply allow us a rationale for adorning our member functions with `const-volatile` qualifiers; it can also allow us to separate a single class concept into base and derived parts.

Ellipsing the Circle

Modeling the relationship and similarity between circles and ellipses (or squares and rectangles, or real numbers and complex numbers) is a recurring question in books, articles, and news groups [Cline+1999, Coplien1992, Coplien1999]. It pops up with a regularity you could almost set your system clock by. Some believe it to be a problem with inheritance – or indeed object orientation as a whole – whereas others believe it is one of semantics [Cline+1999]. But I'm getting ahead of myself: "A circle is a kind of ellipse, therefore `circle` inherits from `ellipse`". Let's start at the (apparent) beginning.

Redundant State

A direct transliteration of that English statement gives us the following C++:

```
class ellipse {
   ...
};
class circle : public ellipse {
   ...
};
```

No problems so far. Depending on your sensibilities, you will next focus on either the public interface or the private implementation. Let's do this back to front and focus on the representation first, just to get one of the common problems out of the way. To keep things simple, let's focus only on the ellipse axes:

```
class ellipse {
   ...
private:
   double a, b;
};
class circle : public ellipse {
   ...
};
```

An ellipse's shape can be characterized by its semi-major (`a`) and semi-minor axes (`b`), whereas a circle needs only its radius to describe it. This means that there is no need to add any additional state representation in `circle` as everything that it needs has already been defined in `ellipse`. So what's the problem? Before you think it, the `private` specifier does not need to be `protected` – *that* would be a problem. The problem is that not only does `circle` not need any additional state, it actually has too much state: `a == b` is an assertable invariant of the circle, which means that there is redundant data. It is neither possible to uninherit features (a rock) nor desirable to maintain redundant state (a hard place).

Focusing on inheritance of representation alone could you lead to the topsy-turvy view that `ellipse` should inherit from `circle` because it would add to its state. Although bizarre, this view is entertained surprisingly often (but is entertaining only if the code is not yours to work with). From bad to worse, there would not only be the absence of any concept of substitutability whatsoever, the `circle` would also have the wrong state to inherit in the first place: `ellipse` would not just inherit a `double`, it would specifically inherit a radius. Now, is the radius the semi-major or the semi-minor axis? The proper conclusion of a representation-focused approach should be that `circle` and `ellipse` should not share an inheritance relationship at all. At least this conservative view is safe and free from strange consequences. For some systems it may also prove to be the right choice in practice: Premature generalization often leads to unnecessary coding effort in creating options that are never exercised.

Interface Classes

Let's focus on the interfaces instead. How can you represent the interface for using instances of a class without actually representing the implementation? In particular, how can you do this if you have related but separate types of objects, e.g. circle versus ellipse? Abstract base classes are often thought of as a mechanism for factoring out common features – interface or implementation, but typically a bit of both – in a class hierarchy. Their most common use is as partially implemented classes, but it turns out that their most powerful use is as pure interfaces.

Interface classes are an idiom rather than a language feature, a method for using a mechanism:

- All ordinary member functions in an interface class are pure `virtual` and `public`.
- There are no data members in an interface class, except perhaps `static const` members.
- The destructor in an interface class is either `virtual` and `public` or non-`virtual` and `protected`. In the former case destruction through the interface class is permitted, and therefore must be made safe. In the latter case destruction is not one of the features offered by the interface, and restricting it also excludes the kinds of `public` problem that require `virtual`.
- An interface class may have type members, e.g. nested classes, which need not be abstract.
- These requirements apply recursively to any base classes.
- That's it.

Writing a class that actually does nothing seems counter-intuitive, if not a little uncomfortable, for many developers. However, having a pure representation of interface gives the developer two clear benefits:

- The ability to publish the interface to objects without the potential cloud and clutter of implementation, the flip side of which is the ability to focus on implementation detail given a clear interface.
- Decoupling client code from implementation detail, which reduces build times (assuming that the concrete implementing class is not in the same header) and the rebuild effect of any change to the implementation (remembering that in software development the only constant is change).

It turns out that these two benefits can be summarized more generally as one: separation of concerns.

Returning to our shapes, we can start establishing the usage for ellipses and circles without getting lost in representation:

```
class ellipse {
public:
   virtual double semi_major_axis() const = 0;
   virtual double semi_minor_axis() const = 0;
   virtual double area() const = 0;
   virtual double eccentricity() const = 0;
   ...
};
class circle : public ellipse {
public:
   virtual double radius() const = 0;
   ...
};
```

Concrete Leaves

The separation of interface from implementation class makes things much clearer. Now, when we come to provide sample implementations, we can see that there is little to be gained from using trying to share implementation detail:

```
class concrete_ellipse : public ellipse {
public:
   ellipse(double semi_major, double
semi_minor)
      : a(semi_major), b(semi_minor) {}
   virtual double semi_major_axis() const {
      return a;
   }
   virtual double semi_minor_axis() const {
      return b;
   }
   virtual double area() const {
      return pi * a * b;
   }
   virtual double eccentricity() const {
      return sqrt(1 - (b * b) / (a * a));
   }
   ...
private:
   double a, b;
};

class concrete_circle : public circle {
public:
   explicit circle(double radius)
      : r(radius) {}
   virtual double radius() const {
      return r;
   }
   virtual double semi_major_axis() const {
      return r;
   }
   virtual double semi_minor_axis() const {
      return r;
   }
   virtual double area() const {
      return pi * r * r;
   }
   virtual double eccentricity() const {
      return 0;
   }
   ...
private:
   double r;
};
```

Should you wish to share implementation you can turn to other techniques to do so, e.g. the BRIDGE pattern [Gamma+1995], but the choice of representation is kept clear from the presentation of interface.

The hierarchy shown here has two very distinct parts to it: The classes that represent usage and the classes that represent implementation. Put another way, the classes exist either for interface, e.g.

```
void draw(point, const ellipse &);
```

or for creation, e.g.

```
concrete_circle unit(1);
draw(origin, unit);
```

In the class hierarchy only the leaves are concrete, and the rest of the hierarchy is abstract – in this case, fully abstract. This design recommendation to "make non-leaf classes abstract" [Meyers1996] can also be stated as “never inherit from concrete classes”. It often helps to simplify subtle class hierarchies by teasing apart the classes according to the distinct roles that they play.

Implementation-Only Classes

If you wish to take decoupling a step further, the separation of concerns can be further reinforced by introducing the polar opposite of an interface class: an implementation-only class. This idiom allows developers to define classes that can only be used for object creation; subsequent object manipulation must be through interface classes [Barton+1994]:

- An implementation-only class inherits, using `public` derivation, from one or more interface classes.
- All ordinary functions in an implementation-only class are `private`.
- Constructors are `public` to allow creation of instances.
- All manipulation of instances is done via pointers or references to one of the base interface classes.
- The destructor is normally `private` if it is `public` in the base interface classes. This means that instances of implementation-only classes are normally on the heap, i.e. the result of the `new` expression is passed immediately to an interface class pointer.
- If the destructor must be `public` in the implementation-only class, this means that instances are normally value based and their lifetime is bound by the enclosing scope, e.g. local variables or data members. This is sometimes a little awkward because the object as held cannot have its members called without an explicit upcast! Hence, implementation-only classes make practical sense only for heap-based objects, which is not appropriate for all designs.

This idiom takes advantage of a feature of C++ that is often seen as a quirk: access specification and `virtual` function mechanisms are orthogonal. Here are alternative definitions for concrete circles and ellipses:

```
class creation_only_ellipse : public ellipse {
public:
   creation_only_ellipse(double semi_major,
                         double semi_minor);
   ...
private:
   virtual double semi_major_axis() const;
   virtual double semi_minor_axis() const;
   virtual double area() const;
   virtual double eccentricity() const;
   ...
   double a, b;
};

class creation_only_circle : public circle {
public:
   explicit creation_only_circle(double radius);
   ...
private:
   virtual double radius() const;
   virtual double semi_major_axis() const;
   virtual double semi_minor_axis() const;
   virtual double area() const;
   virtual double eccentricity() const;
   ...
   double r;
};
```

In use, we could see a circle being created and used as follows:

```
circle *unit = new creation_only_circle(1);
```

Change Happens

Management summary of the story so far: circles substitutable for ellipses, some useful techniques, and no real problems. However, good as these techniques are in many designs, they have not resolved the challenge that really makes the circle–ellipse problem as infamous as it is. If you look back, or if you are already familiar with the problem, you will notice that there has been a little sleight of hand in the interfaces presented: The only ordinary member functions are `const` qualified. In other words, there are no modifiers. You cannot resize circles or stretch ellipses.

Let’s introduce *setters* corresponding to the axis *getters*:

```
class ellipse {
public:
   virtual void semi_major_axis(double) = 0;
   virtual void semi_minor_axis(double) = 0;
   ... // as before
};

class circle : public ellipse {
public:
   virtual void radius(double) = 0;
   ... // as before
};
```

It is fairly obvious what an actual circle does for `radius` and an actual ellipse does for `semi_minor_axis` and `semi_major_axis`, but what does a circle do for the `semi_minor_axis` and `semi_major_axis` setters that it has inherited?

```
class concrete_ellipse : public ellipse {
public:
 virtual void semi_major_axis(double new_a) {
    a = new_a;
 }
 virtual double semi_minor_axis(double new_b) {
    b = new_b;
 }
   ...
private:
  double a, b;
};

class concrete_circle : public circle {
public:
 virtual void radius(double new_r) {
    r = new_r;
 }
 virtual void semi_major_axis(double new_a) {
    ... // what goes here?
 }
 virtual double semi_minor_axis(double new_b) {
    ... // what goes here?
 }
 ...
private:
   double r;
};
```

This is the issue that gets people excited, producing all kinds of fabulous suggestions, all of which violate substitutability in some way:

- Set the radius to the new semi-major or semi-minor axis, regardless. This means that `semi_major_axis` and `semi_minor_axis` break the contract expected of them, i.e. that they resize either one of the semi-major or semi-minor axis, but not both.
- Throw an exception, or crash the program with a diagnostic message, when one of the awkward functions is called.
- Pretend that there isn't really a problem, and that users of the code should not expect anything specific to happen anyway. Besides, all it takes a little semantic elastic to reinterpret the implied meaning of the `semi_major_axis` *setter* not as "reset the semi-major axis" but as "reset the semi-major axis, or do something magical and unspecified, or do nothing".

None of these are reasonable. Distinct unrelated classes are preferable to any of them.

No Change

When you find yourself at the bottom of a hole, stop digging. The design we developed moved forward in stable, well-defined steps, and only became unstuck with a particular requirement.... Wait a minute, resizing circles and ellipses was never a requirement. In fact, we have no requirements or real statement of the original problem domain. So, we're in the middle of a design that does not work, and we have no requirements; do we blame the design, the paradigm, or what? "A circle is a kind of ellipse" was about as much analysis as we – or indeed most people looking at the problem – set out with. We have not stated the purpose or context of our system: Design is context dependent, so if you remove the context you cannot have meaningful design.

Circles and ellipses also seem such a simple, familiar, and intuitive domain that we all feel like domain experts. However, intuition has misled us: When you were taught conic sections at school, were you ever taught that you could that you could resize them? If you transform an ellipse by stretching it, you have a different ellipse, if you resize a circle you have a different circle. In other words, if we chose to model our classes after the strict mathematical model, we would not evolve the design beyond the `const`–member–function-only version, although we might wish to add factory functions to accommodate the transformations. That's it. What violated substitutability was the introduction of change: Eliminate it and you have a model that is both accurate and works [Henney1995].

Now that we understand the real root of the problem – substitutability with respect to change – we can consider alternative approaches that accommodate both change and inheritance-based substitutability. If we do not need both of those features, we can choose to disallow modifiers or not to relate the classes by inheritance, as necessary

Change is the Concern

Therefore, separate according to change. Here is a more useful interpretation of the original statement: "a circle is a kind of ellipse, so long as you don't change it as an ellipse". This is perhaps more constrained than is strictly required, i.e. geometrically resizing an ellipse on both of its axes is also safe for circles, but the constraint makes the issues and their solution clearer.

Given that statement, what we want is something like the following:

```
class ellipse {
   ...
};

class circle : public const ellipse {
                              // not legal C++
   ...
};
```

Other than that one minor nit – that the code isn't legal C++ – we have a solution that allows circles to be substituted for ellipses where they cannot be changed. Where a function expects to change an `ellipse` object via reference or pointer a `circle` would not work, but references and pointers to `const ellipse` would allow `circle` objects to be passed.

However, the good news is that the wished-for version does give us a hint as to how we might organize our classes and functions. Factoring the `const` member functions into a separate interface class gives the following:

```
class const_ellipse {
public:
   virtual double semi_major_axis() const = 0;
   virtual double semi_minor_axis() const = 0;
   virtual double area() const = 0;
   virtual double eccentricity() const = 0;
   ...
};

class ellipse : public const_ellipse {
public:
   using const_ellipse::semi_major_axis;
   using const_ellipse::semi_minor_axis;
   virtual void semi_major_axis(double) = 0;
   virtual void semi_minor_axis(double) = 0;
   ...
};

class const_circle : public const_ellipse {
public:
   virtual double radius() const = 0;
   ...
};

class circle : public const_circle {
public:
   using const_circle::radius;
   virtual void radius(double) = 0;
   ...
};
```

For consistency with the standard library's naming conventions for iterators, `const_ellipse` and `ellipse` are named rather than `ellipse` and `mutable_ellipse`, but your mileage may vary. For consistency within the same design, `circle` is also split into a `const_circle` and a `circle` class.

[concluded at foot of page 18]

Indexing STL lists with maps

Silas S. Brown

A list will keep its elements in the same order that they were added, but searching for elements takes linear time (i.e. time proportional to the length of the list). Removing an element from a list is a constant-time operation if you have an iterator that refers to the element in question, but if you don't have such an iterator then the element must first be found and this will also take linear time.

In the case of a set, searching and removing is faster (logarithmic, and in the case of the SGI extension `hash_set`, amortized linear). However, sets don't remember the order that the elements were added in.

If you want to have an ordered list of elements, but you also want to be able to find given elements quickly, then you can combine a list with a map to index it. This article describes a brief template class called `hashed_list`, which uses the SGI extension `hash_map` and the STL class `list`; it could also use the STL class `map` with little modification. The template class described here does not support all STL semantics (for example, you can't get a non-const iterator out of it); this article is meant to demonstrate the concept.

Note that `hash_map` will assume that all the elements are unique. Generalising this class to support non-unique elements is left as an exercise to the reader (besides using `multimap`, you need to think about the `erase()` method that is defined below).

The general idea

The objects are stored in a list, and also in a map. The map will map objects to list iterators. Whenever an object is added to the list, an iterator that refers to that object is taken and stored in the map under that object. Then when objects need to be counted or deleted, the map is used to quickly retrieve the list iterator, so that the list does not need to be searched linearly.

This relies on the fact that iterators in a list will remain valid even if other parts of the list are modified. The only thing that invalidates a list iterator is deleting the object that it points to. Hence it is acceptable to store list iterators for later reference.

The code

First, we include the two container classes that we will be using:

```
#include <list>
#include <hash_map>
using namespace std;
```

Now for the `hashed_list` class itself. Since `hash_map` is a template class with three arguments (the object type, the hash function, and the equality function), we need to support the same three arguments if we're making a general template:

```
template<class object,
         class hashFunc,class equal>
class hashed_list {
```

I like to start with some typedefs to save typing later. We will need iterators and const iterators to the list, and also a type for the map:

[continued from page 17]

Conclusion

Substitutability is not always about "is a kind of", and not all forms of specialization fit an inheritance hierarchy out-of-the-box. Natural language is often not a very good guide at this level so we often need more phrasing options to reveal the useful relationships in our system, such as "implements" and "can be used where a … is expected".

There are many criteria for partitioning classes into base and derived parts. Taxonomic classification is the most familiar to OO developers, but we have also seen that interface–implementation separation is compelling and that separation with respect to mutability also has a place.

Kevlin Henney
kevlin@curbralan.com

References

[Barton+1994] John J Barton and Lee R Nackman, *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, Addison-Wesley, 1994.

[Cline+1999] Marshall Cline, Greg Lomow, and Mike Girou, *C++ FAQs*, 2nd edition, Addison-Wesley, 1999.

[Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

[Coplien1999] James O Coplien, *Multi-Paradigm Design for C++*, Addison-Wesley, 1999.

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

[Henney1995] Kevlin Henney, "Vicious Circles", *Overload 8*, June 1995.

[Henney2000a] Kevlin Henney, "From Mechanism to Method: Substitutability", *C++ Report 12(5)*, May 2000, also available from `http://www.curbralan.com`.

[Henney2000b] Kevlin Henney, "From Mechanism to Method: Valued Conversions", *C++ Report 12(7)*, May 2000, also available from `http://www.curbralan.com`.

[Henney2000c] Kevlin Henney, "From Mechanism to Method: Function Follows Form", *C/C++ Users Journal C++ Experts Forum*, November 2000, `http://www.cuj.com/experts/1811/henney.html`.

[Henney2001] Kevlin Henney, "From Mechanism to Method: Good Qualifications", *C/C++ Users Journal C++ Experts Forum*, January 2001, `http://www.cuj.com/experts/1901/henney.html`.

[Meyers1996] Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.

This article was originally published on the C/C++ Users Journal C++ Experts Forum in March 2001 at
`http://www.cuj.com/experts/1903/henney.htm`
Thanks to Kevlin for allowing us to reprint it.

```
    typedef list<object>::iterator iterator;
    typedef list<object>::const_iterator
                            const_iterator;
    typedef hash_map<object,iterator,
                 hashFunc,equal> mapType;
```

And the data itself. For brevity I'll be storing it directly and I won't write an explicit constructor, so we don't have to worry about the allocation.

```
    list<object> theList;
    mapType theMap;
```

Now for some methods. Obtaining iterators is simply a matter of getting them from the list; we only allow const iterators because otherwise we'd have to write lots more code to deal with what happens when the user changes things via the iterators. Similarly, obtaining a count of objects just involves getting it from the map (since `hash_map` does not allow duplicate keys, the result will be `0` or `1`).

```
  public:
    const_iterator begin() const {
      return theList.begin();
    }
    const_iterator end() {
      return theList.end();
    }
    mapType::size_type count(
                 const object& k) const {
      return theMap.count(k);
    }
```

Now if we want to add an object to the list, we must also add it to the map. Since the list's `insert()` method returns an iterator that refers to the object we just added, we can give its return value to the map, so our `add()` method is one statement:

```
    void add(const object &o) {
      theMap[o] =
           theList.insert(theList.end(),o);
    }
```

To check that the "objects must be unique" constraint is not being violated, you might wish to add

```
      assert(theMap.count(o)==0);
```

to the beginning of the above method (and `#include <cassert>`).

The following method will erase a given object. First, a map iterator is obtained; this is dereferenced to get the list iterator and erase it from the list, and then the map iterator is used to erase it from the map. That way, only one lookup operation is needed in the map (when the iterator is found); it is not necessary to look up the object twice (once to reference it, again to erase it).

```
    void erase(const object &o) {
      mapType::iterator i=theMap.find(o);
      theList.erase((*i).second);
      theMap.erase(i);
    }
```

For readability you could also write it like this, but it will be less efficient if the lookup takes longer (even in a hashtable it can take a while in the worst case):

```
    void slower_erase(const object &o) {
      theList.erase(theMap[o]);
      theMap.erase(o);
    }
```

Finally, finish the class:

```
  };
```

To test it, you might want to write something like this:

```
  #include <string.h>

  struct strEqual {
    bool operator()(const char* s1,
                    const char* s2) const {
      return (s1==s2 || !strcmp(s1,s2));
      // (although strcmp() might already
      // have the s1==s2 check)
    }
  };

  typedef hashed_list<const char*,
                   hash<const char*>,
                   strEqual> TestType;

  int main() {
    TestType t;
    t.add("one");
    t.add("two");
    t.add("three");
    cout << t.count("two") << endl;
                          // should be 1
    t.erase("two");
    cout << t.count("two") << endl;
                          // should be 0
    copy(t.begin(),t.end(),
         ostream_iterator<const char*>(
                          cout," "));
    cout << endl;
  }
```

Conclusion

The above code will increase the speed of finding items in lists, particularly when they are long. This is at the expense of consuming more memory (because of the map) and making the maintenance of the list slightly slower (because the map needs to be maintained with it). If `hash_map` is being used then the overhead of maintaining the map is (in the amortized case) a constant for each maintenance operation, which may or may not be acceptable depending on the application and on what proportion of its operations are maintenance.

Silas S Brown
<ssb22@cam.ac.uk>

A Return Type That Doesn’t Like Being Ignored.

by Jon Jagger

list<type>::insert

A while ago I finished a contract to write a small subset of STL targeted at embedded C++. This proved an interesting challenge. For example, in full STL, you insert a value into a list using, you’ll never guess, `list::insert`.

```
template<typename type>
list<type>::iterator
list<type>::insert(iterator position,
                   const type& value) {
  ...
}
```

This returns an iterator to the inserted copy of the value or it throws an exception (allocating the internal list node could throw `std::bad_alloc` for example). However, you can’t use this `insert` in embedded C++ for two reasons.

1 Embedded C++ does not support templates. I regard this as a minor problem. In this case templates are being used to get the compiler to write code. You can fake this use of templates quite easily if you put your mind to it.

2 Embedded C++ does not support exceptions. This is definitely not a minor problem. What to do?

The role of failure

My first choice was to emulate exceptions. However, for various understandable reasons, my client decided not to adopt an exception-like mechanism. I’ll explain what I did shortly, but first I’d like to make a short diversion and explain why emulating exceptions is my strong preference. It’s all to do with failure [1].

The role of failure is fundamental to writing good software. When you use STL `list<type>::insert` you don’t have to worry about its return value if it fails because if it fails it doesn’t return. This allows you to write sequences of expressions and statements without having to constantly check if anything went wrong. This is really important.

```
#ifndef LOUD_BOOL_INCLUDED
#define LOUD_BOOL_INCLUDED

class loud_bool {
public:  // construction/destruction

  loud_bool(bool decorated);
  loud_bool(const loud_bool&);
  ~loud_bool();
public:  // conversion
  operator bool() const;
private: // inappropriate
  loud_bool& operator=(const loud_bool&);
private: // state
  bool result;
  mutable bool ignored;
};
#endif
```

Listing 1: loud_bool.hpp

To use an analogy, consider taking a short walk to the local shop for a pint of milk. Do you:

a) check whether you haven’t died after every step, or
b) just walk to the shop and hope you don’t die.

Answer; you just walk to the shop of course. No checks if you’ve died yet. If something serious happens (eg you’re hit by a bus) then you won’t get to the shop but frankly you won’t care much about the milk by then anyway. If you’re still alive you’ll be much more concerned about getting to the nearest hospital.

The point is that constantly checking if you’ve died is silly not because it slows your journey so massively but because it’s exactly the wrong thing to be concerned about in the first place.

The right thing to be concerned about is what happens if you **are** hit by a bus. The reality is simple. If you are hit by a bus you won’t be in much of a state to do anything so the only sensible and practical approach is to make arrangements **before** you set off.

My conclusion is this: The software model of use that exceptions bring with them is so valuable that if you’re working in an environment (or language subset) that doesn’t support exceptions you should consider ways in which you can emulate them. The alternative is constantly checking if you’ve died.

Back to the story

But as I said, my client decided not to adopt an exception-like mechanism. So what did I do? I considered making `list::insert` return an iterator equal to `end()` if the insertion failed but decided this was not a good idea. When things are different they should be clearly different.

```
#include "loud_bool.hpp"
#include <ios>
#include <iostream>

loud_bool::loud_bool(bool decorated)
    : result(decorated),
      ignored(true) {
  // all done
}

loud_bool::loud_bool(const loud_bool & other)
    : result(other.result),
      ignored(other.ignored) {
  other.ignored = false;
}

loud_bool::~loud_bool() {
  if (ignored) {
    std::cerr.setf(std::ios_base::boolalpha);
    std::cerr << "WARNING: return "
              << result
              << "; is being ignored"
              << std::endl;
  }
}

loud_bool::operator bool() const {
  ignored = false;
  return result;
}
```

Listing 2: loud_bool.cpp

One option is to return a `bool` and a `list::iterator` inside a `pair`-like structure. However, as it turned out, my client's particular list requirement didn't require the returned iterator, so the version of `list<type>::insert` I wrote looked like this:

```
bool list::insert(iterator position,
                  const type& value);
```

Lacking exceptions, this `list::insert` returns a `bool` to signify success or failure. The problem now of course is that the `bool` return is just too easy to ignore. In contrast, exceptions, by design, are impossible to ignore. In full STL if you're not interested in the iterator return value, ignoring it is precisely what you do. I thought about this for a while before realizing there was a solution. The problem is that if you ignore a `bool` nothing happens. So the answer is not to use a `bool`! Use something that looks like a `bool`, smells like a `bool`, feels like a `bool`, and shouts loudly when ignored. (In practice, this version for embedded C++ has to be modified slightly because embedded C++ does not support the `mutable` keyword either.) See Listings 1-3.

The `loud_bool` class generalizes to the template class shown in Listings 4-6 for those of you working in full C++ (I've not put it in a namespace to save horizontal space).

```
#include "loud_bool.hpp"

loud_bool example1() {
  //...
  return true;
}

int main() {
  bool result = example1(); //no warning
  example1();               //generates warning
  return 0;
}
```

Listing 3: Example use of loud_bool

```
#ifndef WARN_IF_IGNORED_INCLUDED
#define WARN_IF_IGNORED_INCLUDED

template<typename result_type>
class warn_if_ignored {
public:  // construction/destruction
  warn_if_ignored(const result_type&
                                     decorated);
  warn_if_ignored(const warn_if_ignored&);
  ~warn_if_ignored();
public:  // conversion
  operator result_type() const;
private: // inappropriate
  warn_if_ignored& operator=(const
                             warn_if_ignored&);
private: // state
  result_type result;
  mutable bool ignored;
};

#include "warn_if_ignored-template.hpp"
#endif
```

Listing 4: warn_if_ignored.hpp

The final step (left as exercise for the reader) is to parameterize the behaviour if the result is ignored.

Jon Jagger
jon@jaggersoft.com

References

[1] *To Engineer is Human. The Role of Failure in Successful Design.* Henry Petroski. Vintage. 0-679-73416-3

```
#include <ios>
#include <iostream>

template<typename result_type>
warn_if_ignored<result_type>::warn_if_ignored(
    const result_type & decorated)
    : result(decorated),
      ignored(true) {
  // all done
}

template<typename result_type>
warn_if_ignored<result_type>::warn_if_ignored(
    const warn_if_ignored & other)
    : result(other.result),
      ignored(other.ignored) {
  other.ignored = false;
}

template<typename result_type>
warn_if_ignored<result_type>::~warn_if_ignored(){
  if (ignored) {
    std::cerr << "WARNING: return "
              << result
              << "; is being ignored"
              << std::endl;
  }
}

template<typename result_type>
warn_if_ignored<result_type>::operator
                          result_type() const {
  ignored = false;
  return result;
}
```

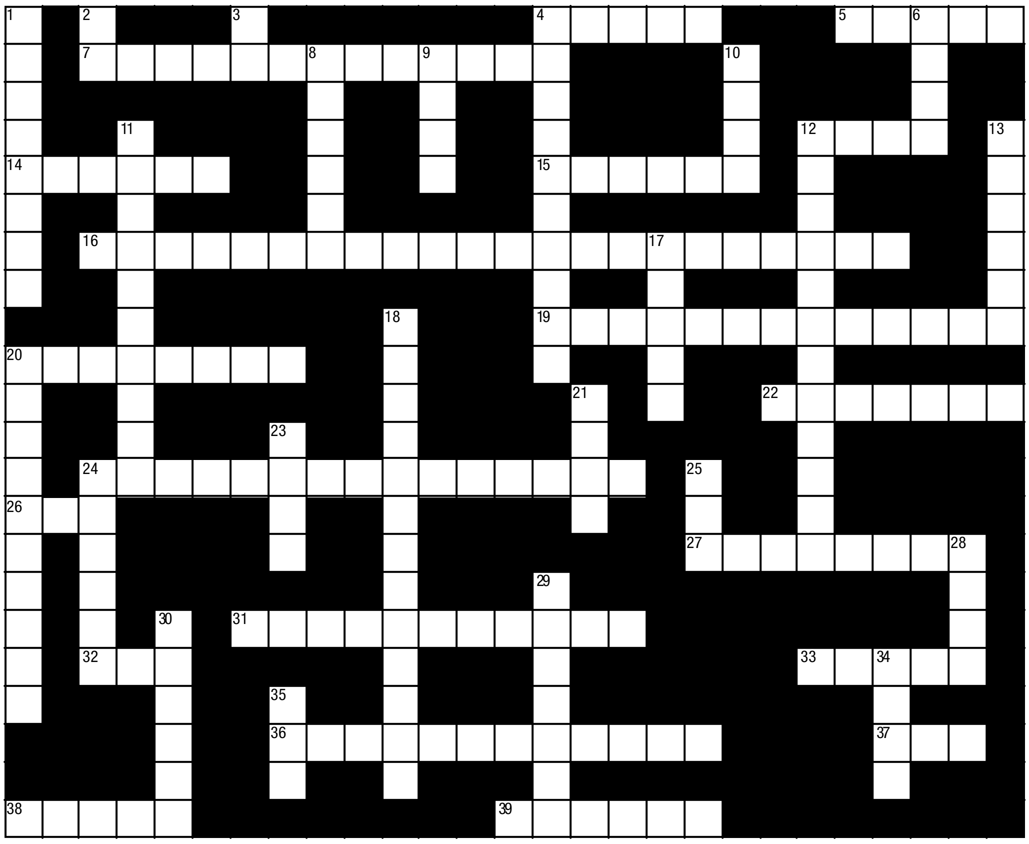
Listing 5: warn_if_ignored-template.hpp

```
#include "warn_if_ignored.hpp"

warn_if_ignored<bool> example2() {
  // ...
  return false;
}

int main() {
  bool result = example2(); //no warning
  example2();               //generates warning
  return 0;
}
```

Listing 6: Example use of warn_if_ignored template

C++rossword

by Lois Goldthwaite



ACROSS

4. Deer and antelope are equally at home here, unless I am in error (5) [25.3.3.3, 19.1p3]
5. Whether on its own or as part of a class, does intensive study risk brain damage? (5) [21, 27]
7. Sign of honest dealing (6,7) [25.2.11]
12. Optional member of a sequence, but not of a set (4) [23.1.1p12]
14. Another word for lower and upper bounds (6) [18.2]
15. Bug - the malarial kind, that is (6) [23.2.4]
16. Useful algorithm when consulting a dictionary (15,7) [25.3.8]
19. Most likely to catch the bus (8,5) [23.2.3.2]
20. No means to light cigarette? Oh, dear (8) [25.1.7]
22. Cash consequence of a fault (7) [23.2.4.2]
24. Interior separation for horses' home (6,9) [25.2.12]
26. Whole, but abbreviated (3) [3.9.1p2]
27. If car won't start, do this (4,4) [25.3.6.1]
31. Time too short to put everything in order? Maybe this is good enough (7,4) [25.3.1.3]
32. Every tourist needs at least one (3) [23.3.1]
33. A disadvantage or drawback that lessens the value of something (5) [20.3.2]
36. Serendipitous discovery, perhaps? (8,4) [25.1.5]
37. Number 8's successful attempt to score (3) [15]
38. Make two into one (5) [25.3.4]
39. Volunteer sailors endeavour to keep bark in proper condition (1,5) [3.9.1]

Solution on page 26

DOWN

1. Careless specialisation might do this (8) [14.7.3p7]
2. Logical alternative (2) [20.3.4]
3. Song of dwarves, or indeed anyone with need to communicate (2) [27]
4. Curator finds fake artifact in museum, and corrects the situation (6,4) [25.2.7]
6. Debugger's motto: to strive, to ____, to find, and not to yield (4) [27]
8. Marionette muscle (6) [21]
9. Action performed on petrol tanks and other containers (4) [25.2.5]
10. To perform odd jobs around the house, like scorching a steak (4) [3.9.1]
11. Hydrogen (3,7) [25.3.7]
12. Treasure hunt for specified quantity of something (6,6) [25.3.3]
13. What a good investment should do, often (6) [3.9.1]
17. Useful when dozy programmers need a wake-up call (5) [18.7, 20.5]
18. Goes now this way, now that (13) [24.1.4]
20. What a single rabbit doesn't do, no matter how many times it tries (10) [20.3.2]
21. Wooden shoe obstructs drainage to narrow stream (4) [27.3.1]
23. To tear something apart violently, with terminal conclusion (4) [23.1p9]
24. Venue for Pooh sticks (6) [27]
25. Informal reference to father(3) [23.2.3.3]
28. A good point, a positive advantage, in fact (4) [20.3.2]
29. Might be noble, under certain conditions (5,2) [25.1.6]
30. A sailor does this to mainbraces and docking lines (6) [23.2.2.4]
34. After Pied Piper left Hamelin, how many rats were still there? (3,1) [20.3.5]
35. Australian madman (3) [25.3.7]

C++ Standards Library Report

by Reg Charney

I have just returned from Santa Cruz where we held the semi-annual Standards meeting on C++. I spent most of my time in the Library Working Group discussing new features for the next Library Extensions Technical Report. I have reported on some of the main topics below.

Move semantics

Move semantics promises to significantly increase run-time efficiency of many library elements — in some cases by a factor of 10+. The proposal has generated a lot of discussion for years and is finally bearing fruit. It originates in the Core Working Group but mainly affects the Library. Changes are 100% compatible with existing code, while introducing a fundamental new concept into C++.

This proposal augments copy semantics. A user might define a class as copyable and movable, either or neither. The difference between a copy and a move is that a copy leaves the source unchanged. A move on the other hand leaves the source in a state defined differently for each type. The state of the source may be unchanged, or it may be radically different. The only requirement is that the object remain in a self consistent state (all internal invariants are still intact). From a client code point of view, choosing move instead of copy means that you don't care what happens to the state of the source. For plain old data structures (PODs), move and copy are identical operations (right down to the machine instruction level).

This paper proposes the introduction of a new type of reference that will bind to an rvalue:

```
struct A {/*...*/};
void foo(A&& i); // new syntax
```

The `&&` is the token which identifies the reference as an "rvalue reference" (bindable to an rvalue) and thus distinguishes it from our current reference syntax (using a single `&`).

The rvalue reference is a new type, distinct from the current (lvalue) reference. Functions can be overloaded on `A&` and `A&&`, requiring such functions each to have distinct signatures.

The most common overload set anticipated is:

```
void foo(const A& t); // #1
void foo(A&& t);      // #2
```

The rules for overload resolution are (in addition to the current rules):

- rvalues prefer rvalue references.
- lvalues prefer lvalue references.

CV qualification conversions are considered secondary relative to r-/l-value conversions. rvalues can still bind to a const lvalue reference (`const A&`), but only if there is not a more attractive rvalue reference in the overload set. lvalues can bind to an rvalue reference, but will prefer an lvalue reference if it exists in the overload set. The rule that a more cv-qualified object can not bind to a less cv-qualified reference stands - both for lvalue and rvalue references.

Examples:

```
struct A {};
A source();
const A const_source();
// ...
A a;
const A ca;
foo(a);               // binds to #1
foo(ca);              // binds to #1
foo(source());        // binds to #2
foo(const_source()); // binds to #1
```

The first `foo()` call prefers the lvalue reference as (lvalue)a is a better match for `const A&` than for `A&&` (lvalue -> rvalue conversion is a poorer match than `A& -> const A&` conversion). The second `foo()` call is an exact match for #1. The third `foo()` call is an exact match for #2. The fourth `foo()` call can not bind to #2 because of the disallowed `const A&& -> A&&` conversion. But it will bind to #1 via an rvalue->lvalue conversion.

Note that without the second `foo()` overload, the example code works with all calls going to #1. As move semantics are introduced, the author of `foo` knows that he will be attracting non-const rvalues by introducing the `A&&` overload and can act accordingly. Indeed, the only reason to introduce the overload is so that special action can be taken for non-const rvalues.

Tuples

Tuples are fixed-size heterogeneous containers containing any number of elements. They are a generalized form of `std::pair`. The proposal originates in the Core Working Group from a Boost Library [1] implementation, but it will mainly affect in the Library.

The proposed tuple types:

- Support a wider range of element types (e.g. reference types).
- Support input from and output to streams, customizable with specific manipulators.
- Provide a mechanism for 'unpacking' tuple elements into separate variables.

The tuple template can be instantiated with any number of arguments from 0 to some predefined upper limit. In the Boost Tuple library, this limit is 10. The argument types can be any valid C++ types. For example:

```
typedef
  tuple< A,
         const B,
         volatile C,
         const volatile D
       > t1;
```

An n-element tuple has a default constructor, a constructor with n parameters, a copy constructor and a converting copy constructor. By converting copy constructor we refer to a constructor that can construct a tuple from another tuple, as long as the type of each element of the source tuple is convertible to the type of the corresponding element of the target tuple. The types of the elements restrict which constructors can be used:

- If an n-element tuple is constructed with a constructor taking 0 elements, all elements must be default constructible.

For example:

```
   class no_default_ctor {no_default_ctor();};
   tuple<no_default_ctor, float> b;
                // error - need default ctor
   tuple<int&> c; // err no ref default ctor
   tuple<int, float> a;  // ok
```

If an n-element tuple is constructed with a constructor taking n elements, all elements must be copy constructible and convertible (default initializable) from the corresponding argument. For example:

```
   tuple<int, const int, std::string>(1,
                                 'a', "Hi") // ok
   tuple<int, std::string>(1, 2); // error
```

- If an n-element tuple is constructed with the converting copy constructor, each element type of the constructed tuple type must be convertible from the corresponding element type of the argument.

```
   tuple<char, int, const char(&)[3]>
                              t1('a', 1, "Hi");
   tuple<int,float,std::string> t2 = t1; // ok
```

The argument to this constructor does not actually have to be of the standard tuple type, but can be any tuple-like type that acts like the standard tuple type, in the sense of providing the same element access interface. For example, `std::pair` is such a tuple-like type. For example:

```
   tuple<int,int> t3 = make_pair('a',1); // ok
```

The proposal includes tuple constructors, utility functions, and ways of testing for tuples. The I/O streams library will be updated to support input and output of tuples. For example,

```
   cout << make_tuple(1,"C++");
```

outputs

```
   (1 C++)
```

Hash Tables

Hash tables almost made it into the first issue of the Standard, but our deadline precluded doing due diligence on the proposal at the time.

The current proposal is what you would expect and is compatible with the three main library implementations for hash tables: SGI/STLport, Dinkumware, and Metrowerks. What is new is that things like `max_factor`, `load_factor`, and other constants are now treated as hints. Each implementation is free to deal with them as it sees fit. Also, most `double` values and parameters are going to be `float`. It was felt that only one or two significant digits were meaningful, so the extra space needed by `double` was wasted.

There was one major outstanding issue. Since there are already widespread hash library implementations in use, how do we avoid breaking existing code. Proposals included making the new hash library names slightly different, or placing them in different namespaces, or usurping the current names because only the committee has the authority to place things in namespace `std` (any private implementation that placed their hash library into `std` was, by definition, in error.)

Polymorphic Function Object Wrapper

This proposal is based on the Boost Function Template Library [1]. It was accepted in Santa Cruz for inclusion in the next TR of the standard. It introduces a class template that generalizes the notion of a function pointer to subsume function pointers, member function pointers, and arbitrary function objects while maintaining similar syntax and semantics to function pointers.

This proposal defines a pure library extension, requiring no changes to the core C++ language. A new class template function is proposed to be added into the standard header `<functional>` along with supporting function overloads. A new class template `reference_wrapper` is proposed to be added to the standard header `<utility>` with two supporting functions. For example,

```
   int add(int x, int y) {
     return x+y;
   }
   bool adjacent(int x, int y) {
     return x == y-1 || x == y+1;
   }
   struct compare_and_record {
     std::vector<std::pair<int,int> > values;
     bool operator()(int x, int y) {
       values.push_back(std::make_pair(x,y));
       return x == y;
     }
   };
   function< int (int, int) > f;
   f = &add;
   cout << f(2, 3) << endl; // outputs 5
   f = minus<int>();
   cout << f(2, 3) << endl; // outputs -1
```

The proposed function object wrapper supports only a subset of the operations supported by function pointers with slightly different syntax and semantics. The major differences are detailed below, but can be summarized as follows:

- Relaxation of target requirements to allow conversions in arguments and result type.
- Lack of comparison operators. However, checking `if (f)` and `if (!f)` is allowed.
- Lack of extraneous null-checking syntax.

The function class template is always a function object.

Conclusions

This short report did not touch on other exciting new features that will likely appear in the language, including the Regular Expression proposal, the use of static assertions to allow more extensive compile-time checking based on type information, and the auto expressions proposal to simplify template function definitions.

As you can see from the wide range of topics we discussed in Santa Cruz, C++ is still an exciting language that is continuing to adapt to users needs.

Reg Charney

References

The Boost Library: `www.boost.org`

This article was originally published on the ACCU USA website in November 2002 at
`http://www.accu-usa.org/2002-11.html`
Thanks to Reg for allowing us to reprint it.

C++ Templates

by Reg Charney

If you wanted to know just about everything about C++ templates then the book *C++ Templates—The Complete Guide* by David Vandevoorde and Nicolai Josuttis, (ISBN 0-201-73484-2) is a readable reference book you can use. Normally, discussing a book would appear in a book review. However, since the authors have done such a good job of describing C++ templates, I though the topic and the book deserved more complete coverage. You may recall Nicolai as the author of the excellent text, *The C++ Standard Library*, ISBN 0-201-37926-0. David is the author of *C++ Solutions: Companion to the C++ Programming Language , Third Edition*, ISBN 0-201-30965-3. Both authors are also longtime members of the ANSI/ISO C++ X3J16 Standardization Committee.

Ordering Convention

Most of us write the following `const` or `volatile` declaration thus:

```
volatile int vInt;
const char *pcChar;
```

The authors suggest that a better way is:

```
int volatile vInt;  // 1
char const *pcChar; // 2
```

Here, `const` and `volatile` appear before what they qualify. Since I read declarations from right to left, in `//1`, I get `vInt` is a `volatile int` and in `//2`, I get `pcChar` is a pointer to a `const char`. The real power comes when we use this ordering in `typedef` statements. For example,

```
typedef int   *PINT;
typedef PINT  const CPINT;
typedef const PINT  PCINT;
```

Here we can see that the first `typedef` (pointer to `int`) can be used in the second `typedef` (`const` pointer to `int`). The third `typedef` completely changes the meaning of the type (pointer to `const int`).

typename Keyword

Historically, we have used the keyword, `class`, in template parameter lists:

```
template< class T > . . .
```

However, `T` could be replaced by things other than a class name, so the use of the new keyword `typename` is preferred.

Reducing Ambiguities

Often, instantiating a template can result in an ambiguity error that is difficult to understand. Here is a simple example:

```
template< typename T > plot2D(T& x, T& y);
plot2D(3,4);     // ok
plot2D(3.14,1); // error - types of
                // arguments differ
```

The ambiguity occurs because all parameters are supposed to be of same type, but in the second case, it is unclear whether that type should be an `int` or a `double`.

In this simple example (and many others like it), there are 3 ways of disambiguating the statement:

```
plot2D<int>(3.14,1);                 // 3
plot2D(static_cast<int>(3.14), 1); // 4
```

In `//3`, the template type is forced to be `int`. In `//4`, casting forces all the argument types to be the same. The third way to eliminate this problem uses more sophisticated templates.

Overloading function template

It is possible to have both template and non-template versions of the same function. Often, non-template versions are called specializations. In such situations, function overloading and its rules are used to resolve any ambiguities. In addition to normal function overloading rules, a few extra rules are needed for templates.

- In instantiating a template function, no type conversions are done.
- All things being equal, specialized template functions are preferred over template ones.
- If instantiating a template function results in a better match, the better match is used. By better match, we mean that things like conversions are not need to make a match.
- If the empty angle brackets notation is used, non-template functions are ignored in matching argument.

Here are some examples:

```
int cmp(int const& a, int const& b);
template<typename T> T cmp(T const& a,
                           T const &b);
cmp(1,2);         // 5
cmp(4.3, -1.2);   // 6
cmp('x','s');     // 7
cmp<>(-3,2);      // 8
cmp<int>(4.3,4); // 9
```

In `//5`, the non-template function is the best match (Rule 2). In `//6`, template argument deduction instantiates the `cmp<double>` function (Rule 3). In `//7`, the instantiated function is `cmp<char>` (Rule 3). In `//8`, the notation forces use of the template function and results in a `cmp<int>` function being instantiated and used instead of the non-template version of the `cmp` function (Rule 4).

Partial Specialization

When a template class or function has more than one template parameter, one or more may be specified creating a partially specialized template. However, if one or more partial specializations match the same template, an ambiguity occurs. For example,

```
#include <typeinfo>
#include <stdio.h>
typedef char const CC;
// general template function
template<typename T1,typename T2>
void f(T1 t1, T2 t2) {
  CC* s1=typeid(T1).name();
  CC* s2 = typeid(T2).name();
  printf("f(%s,%s)\n",s1,s2);
}
// partial specialization 1, both types the same
template<typename T>
void f(T t1, T t2) {
  CC* s1 = typeid(T).name();
  CC* s2 = typeid(T).name();
  printf("f1(%s,%s)\n",s1,s2);
}
// partial specialization 2
// 2nd parameter is non-type
template< typename T>
void f(T t1, int t2) {
  CC* s1 = typeid(T).name();
  CC* s2 = typeid(int).name();
  printf("f(%s,%s)\n",s1,s2);
}
```

```
// partial specialization 3
// parameters are all pointers
template< typename T1, typename T2>
void f(T1* t1, T2* t2) {
  CC* s1 = typeid(T1*).name();
  CC* s2 = typeid(T2*).name();
  printf("f(%s,%s)\n",s1,s2);
}
int main(int argc,char* argv[]) {
  char const* s1=”one”;
  f(1,21.3);
  f(1.2,-3);
  f('a','z');
  f(s1,2);
  f(&s1,"two");
  return 0;
}
```

The output from this program is:

```
f(int,double)
f(double,int)
f(char,char)
f(CC *,int)
f(CC **,char *)
```

Note that calling `f(3,5)` is ambiguous because both `f(T,T)` and `f(T,int)` match this call.

Non-type Template Parameters

Both classes and functions can use non-type template parameters. When used, non-type parameters become part of the classes type and functions signature. Thus,

```
template<typename T, int n>
class C {
  T a[n];
  // . . .
};
C<char, 10> c10A;
C<char, 5>  c5A;
C<int, 10>  i10A;
```

Each of `c10A`, `c5A`, and `i10A` are all distinct types.

An example of a template function using non-type parameter follows:

```
template<typename T, int n>
T g(T t) {
  return t+n;
}
int main() {
  printf("g<double,5>(1.3)=%g\n",
         g<double,5>(1.3));
  printf("g<char,4>(‘a’)=%c\n",
         g<char,4>(‘a’));
}
```

The output of this short program is:

```
g<double,5>(1.3)=6.3
g<char,4>('a')=e
```

Non-type template parameters have restrictions: they must be integral values, enumerations, or instance pointers with external linkage. They can’t be string literals nor global pointers since both have internal linkage.

Keyword typename

You may have wondered why we needed the new keyword `typename`. Besides being a better choice for template parameters, it is needed to disambiguate certain declarations. E.g.,

```
template<typename T, int n>
class C {
  typename T::X *p;
  // . . .
};
```

Without the keyword `typename`, the declaration for `p` becomes an expression: the value of `T::X` is multiplied by the value of `p`.

Generally, prefix all declarations using template parameters with `typename`.

this pointer

Normally derived and base classes share the value of `this`. However, lookup rules for template classes are different. Template base class members are not lookup when searching derived template class member functions.

```
template<typename T>
class B {
  void foo();
}
template<typename T>
class D: public B<T> {
  void bar() { foo(); }
}
```

In `bar()`, `foo()` would not be found. To get `B`’s `foo`, you need to say `this->foo()`. Again, the rule given in the book states that any base class member used in a derived class should be qualified by `this->` or `B<T>::`.

Conclusion

I have not begun to cover many of the interesting aspects of templates in this short article. And I have barely covered the other fun parts of this book. Get it.

Reg Charney

This article was originally published on the ACCU USA website in December 2002 at
http://www.accu-usa.org/2002-12.html
Thanks to Reg for allowing us to reprint it.

C++rossword answers

Across: 4 RANGE, 5 BASIC, 7 RANDOM SHUFFLE, 12 BACK, 14 LIMITS, 15 VECTOR, 16 LEXICOGRAPHICAL COMPARE, 19 PRIORITY QUEUE, 20 MISMATCH, 22 RESERVE, 24 STABLE PARTITION, 26 INT, 27 PUSH HEAP, 31 PARTIAL SORT, 32 MAP, 33 MINUS, 36 ADJACENT FIND, 37 TRY, 38 MERGE, 39 A FLOAT

Down: 1 IMMOLATE, 2 OR, 3 IO, 4 REMOVE COPY, 6 SEEK, 8 STRING, 9 FILL, 10 CHAR, 11 MIN ELEMENT, 12 BINARY SEARCH, 13 DOUBLE, 17 CLOCK, 18 BIDIRECTIONAL, 20 MULTIPLIES, 21 CLOG, 23 REND, 24 STREAM, 25 POP, 28 PLUS, 29 COUNT IF, 30 SPLICE, 34 NOT 1, 35 MAX