The magazine of the ACCU

www.accu.org

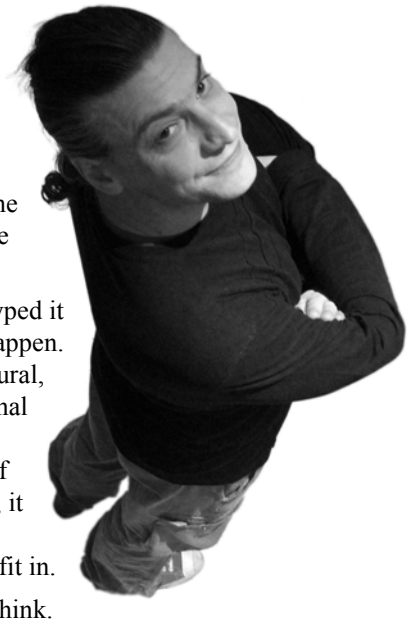
{cvu}

Volume 21 Issue 5 November 2009

Features

{cvu}
Volume 21 Issue 5
November 2009
ISSN 1354-3164
www.accu.org

Reflections on Learning



Like many other people, I first learned programming in BASIC, on a Sinclair ZX81. In fact, it started out a bit like pair programming. The ZX81 belonged to my friend and we would type in listings from some of the popular personal computing magazines of the time. One would dictate the code, the other would type it in. The 'dictation' part doesn't ***necessarily*** fit the pair programming practice, but part of the process was that the typos made by the person at the keyboard might be more easily spotted by the one reading from the printed page.

Back then we paid no heed to 'good practice', we just typed it in and occasionally changed things to see what would happen. It wouldn't be for some time until I encountered procedural, then modular, then object oriented, generic, and functional programming. These things all have their own good practices, even if you use one or two languages for all of them. There are subtleties which, as a lone programmer, it would have taken forever just to find out about. That's where pair programming, and communities like ACCU fit in.

The ways we learn from each other shape the ways we think. Having variety here helps us to distinguish good practice from common, but questionable, practice. There is no substitute for experience; we learn from our mistakes, and sometimes it brings us insight to question what we had previously taken for good practice.

I hope I haven't fulfilled Edsger Dijkstra's assertion that it's impossible to teach good programming practice to students with prior exposure to BASIC! ('How do we tell truths that might hurt?' (EWD498), 1975). That said, I think I am ***still*** learning good programming practice, and at least I was never taught COBOL...

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

The articles in this magazine have all been written by programmers, for programmers – and have been contributed free of charge.

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

REGULARS

FEATURES

SUBMISSION DATES

C Vu 21.5: 1st October 2009
C Vu 21.6: 1st December 2009

Overload 94: 1st November 2009
Overload 95: 1st January 2010

WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

ADVERTISE WITH US

COPYRIGHTS AND TRADE MARKS

Respect the Software Release Process

Pete Goodliffe implores us to take the 'last step' carefully.

Creating a *software release* is an incredibly important step in your software development process, and not one that should only be thought about at the last minute. On more than one recent occasion I have run into silly, perfectly avoidable problems that were caused by other developers' lackadaisical approach to the construction of software releases.

Many of these were caused by the sloppy habit of creating a 'release' of a local working directory, rather than from a clean checkout. (Hint: this is not a real software release, it's a 'build' of your code, you need a *lot* more process to create a proper release.)

For example:

- An external software release was made from a developer's local working directory. The developer hadn't checked the state of his checked-out code thoroughly. The directory contained uncommitted source file changes. The 'release' was made anyway, and when there were problems reported with it we had no record of exactly *what* went into that build. Once we knew it had been built like this, no-one had faith in the quality of the release at all.
- An external software release was made from a local directory that wasn't up-to-date. The developer hadn't updated to the HEAD of the Subversion repository. So it was missing one feature, and some bug fixes. But the developer tagged the HEAD of the repository as the 'release point', and then claimed he'd built that version. The built code begged to differ. When people noticed that the code didn't contain fixes that were marked in the bug tracker, the developer tried to deny responsibility. (Bonus question: how did it get released at all, without thorough testing?)

I mean, *come on*! It's not *that* hard, is it?

Well, actually: yes it is. Creating a serious high-quality software release is actually a lot more work than just hitting 'build' in your IDE and shipping whatever comes out. If you are not prepared to put in this extra work then you should not be creating releases.

Harsh. But fair.

Part of the process

Most people write software for the benefit of others as well as themselves. So it has to get into the hands of your 'users' somehow. Whether you end up rolling a software installer shipped on a CD, a downloadable installer bundle, a zipfile of source code, or deploying the software on a live web server, this is the important process of creating a software release.

The software release process is a critical part of your software development regimen, just as important as design, coding, debugging, and testing. To be effective your release process must be:

- simple
- repeatable
- reliable

Get it wrong, and you will be storing up some potentially nasty problems for your future self. When you construct a release you must:

- Ensure that you can get the exact same code that built it back again from your source control system. (You do use source control, don't you?) This is the only concrete way to prove which bugs were and were not fixed in that release. Then when you have to fix a critical bug in version 1.02 of a product that's five years old, you can do so.
- Record exactly how it was built (including the compiler optimisation settings, target CPU configuration, etc.). These features may subtly affect how well your code runs, and whether certain bugs manifest themselves.
- Capture the build log for future reference.

A cog in the machine

The bare outline of a good release process is:

- Agree that it's time to spin a new release. A formal release is treated differently to a developer's test build, and should *never* come from an existing working directory.
- Agree what the 'name' of the release is (e.g. '5.06 Beta1' or '1.2 Release Candidate').
- Determine exactly what code will constitute this release. In most formal release processes, you will already be working on a *release branch* in your source control system, so it's the state of that branch right now. You should rarely release code directly from your source control system's mainline (i.e. *trunk* or *master*) of code development. A release branch is a stable snapshot of the code that allows you to continue development on the trunk. You can merge in the good, stable, known fixes from the mainline into the release branch once they are proven. This maintains the integrity of the release codebase whilst allowing other new work to continue on the mainline.
- Tag the code in source control to record what is going into the release. The tag name must reflect the release name.
- Check out a virgin copy of the entire codebase at that tag. ***Never*** use an existing checkout. You may have uncommitted local changes that change the build. Always tag *then* checkout the tag. This will avoid many potential problems.
- Build the software. This step ***must not*** involve hand-editing any files at all, otherwise you do not have a versioned record of exactly the code you built.
- Ideally, the build should be automated: a single button press, or a single script invocation. Checking the mechanics of the build into source control with the code records unambiguously how the code was constructed. Automation reduces the potential for human error in the release process.
- Package the code (create an installer image, CD ISO images, etc.). This step should also be automated for the same reason.
- Always test the newly constructed release. Yes, you tested the code already to ensure it was time to release, but now you should test this 'release' version to ensure it is of suitable release quality.
- Construct a set of 'Release notes' describing how the release differs from the previous release: the new features and the bugs that have been fixed.
- Store the generated artefacts and the build log for future reference.
- *Test the release!* There should be an initial smoke-test to ensure that the installers work OK (on all supported deployment platforms) and

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes. Pete can be contacted at pete@goodliffe.net

Java Dependency Management with Ivy

Paul Grenyer looks into Ivy to help with Java dependencies.

In Part I [1] of 'Java Dependency Management with Ivy', I looked at basic Ivy [2] usage and configuration using Ant [3] and the Ivy Eclipse [4] plugin, IvyDE [5]. I demonstrated how Ivy can be used to download modules (dependencies) from a repository, such as the Maven Repository [6] and cache them locally, negating the need to check them into a source control system.

However there are some scenarios where the Maven repository is not suitable:

1. Your development team may not have direct access to the Maven repository or you want to prevent each module from being downloaded more than once.
2. You may want to restrict or specify the modules your development team has access to.
3. You want to use libraries such as Microsoft's SQL Server JDBC driver [7] or your own proprietary JARs that are not hosted in the Maven repository.

Ivy and IvyDE can be easily configured to look at custom repositories. In this article I will discuss a way of setting up a local public repository and a shared repository, and how to reference them from Ivy and IvyDE.

In my previous article I also explained the difference between a repository and a cache. As I am going to look at repositories in a little more detail it is worth repeating the distinction. A cache is usually local. When you do a build, Ivy checks the cache to see if you already have the required modules. If you do, it uses them, otherwise it looks in one or more repositories for them and downloads them. Repositories can be local, but tend to be remote on the internet or on a central server in an organisation. Maven is a repository and stores a large number of modules.

Ivy repositories

As described by the Ivy 'Adjusting Default Settings' documentation[8], Ivy uses three kinds of repositories:

- Public – a public repository in which most modules, and especially third party modules, can be found
- Shared – a repository which is shared between all the members of a team and hosts proprietary modules
- Local – a repository which is private to the user and hosts modules specific to them.

The documentation goes into more depth, but basically by default the Public repository is the Maven Repository, the Shared repository is a directory called `shared` in the user's home directory and the Local repository is in a folder called `local` in the user's home directory. The Shared repository is intended to be shared by the development team and would usually, but not by default, be hosted on an internal server. The local repository is private to each user and hosted on their local machine.

By default, Ivy first checks the Local repository, then the Shared repository and finally the Public repository. This can be changed as described in the documentation. If for some reason you are unable or do not want to access the Maven repository directly, a shared repository can be set up to proxy the Maven repository and be used as the public repository. How to set this up is documented on the Ivy website under 'Building a Repository' [9], but the documentation lacks detail, so I'll describe it in more detail here.

Glossary of terms

Some of the terms used in the Ivy documentation and the various XML files are not as intuitive as they could be. Particularly the difference between a module and an artifact. To try and make this article a little easier to understand I'm including a brief glossary of terms below. More detailed explanations can be found on the Ivy 'Terminology' page [11].

PAUL GRENYER

An active ACCU member since 2000, Paul is the founder of the Mentored Developers. Having worked in industries as diverse as direct mail, mobile phones and finance, Paul now works for a small company in Norwich writing Java. He can be contacted at paul.grenyer@gmail.com

Respect the Software Release Process (continued)

that the software appears to run and function correctly. Then perform whatever testing is appropriate for this type of release. Internal test releases may be run through test scripts by in-house testers. Beta releases may be released to selected external testers. Release candidates should pass suitable final regression checks. A software release should not be distributed until it has been checked.

- Deploy the release. Perhaps this involves putting the installer on your website, sending out memos or press releases to people who need to know. Update release servers as appropriate.

Release early and often

One of the worst release process sins is to think about this stuff only as you reach the end of a project, when you finally need to perform a public software release.

We've seen that the ideal release process is entirely automated. The automated build and release plumbing should be established early on in the development process. It should then be used often (daily, if not more frequently) to prove that it works, and that it is robust. This will also help to highlight and eliminate any nefarious presumptions in the code about the installation environment (hard coded paths, presumptions about the computer's installed libraries, capabilities, etc.).

Starting the software release process early in the development effort gives you plenty of time to iron out wrinkles and flaws so that when you are in the run-up to a real public release you can focus on the important task of making your software work, rather than the tedious mechanics of how to ship it.

And there's more...

I'll admit I've passed over these points fairly quickly. This is a large topic tied intimately with configuration management, source control, testing procedures, software product management, and the like. If you have any part in releasing a software product you really must understand and respect the sanctity of the software release process. ■

Term	Description
Organisation	An organisation is either a company, an individual, or simply any group of people that produces software. The Ivy 'organisation' is very similar to the Maven POM 'groupId'.
Module	A module is a self-contained, reusable unit of software that, as a whole unit, follows a revision control scheme. Ivy is only concerned about the module deliverables known as artifacts, and the module descriptor that declares them.
Module Descriptor	A module descriptor is a generic way of identifying what describes a module: the identifier (organisation, module name, branch and revision), the published artifacts, possible configurations and their dependencies. The most common module descriptors in Ivy are Ivy Files, xml files with an Ivy specific syntax, and usually called `ivy.xml`.
Artifact	An artifact is a single file ready for delivery with the publication of a module revision, as a product of development. Compressed package formats are often preferred because they are easier to manage, transfer and store. For the same reasons, only one or a few artifacts per module are commonly used. However, artifacts can be of any file type and any number of them can be declared in a single module.
Revision	A unique revision number or version name is assigned to each delivered unique state of a module. Ivy can help in generating revision numbers for module delivery and publishing revisions to repositories, but other aspects of revision control, especially source versioning, must be managed with a separate version control system. Therefore, to Ivy, a revision always corresponds to a delivered version of a module.

Hosting a repository

A repository can be a few things, including a shared drive or a webserver. Setting up a webserver to host repositories negates the need for sharing drives, which has got to be a good thing. Any webserver that can serve files can be used. My personal preference is for Apache [11]. After installing Apache create a directory called `Ivy` in the `htdocs` directory. This is the location for your repositories and can be accessed as http://myserver/ivy.

Creating a public repository

A locally hosted public repository holds the required subset of modules from external repositories such as the Maven repository. The **`ivy:install`** task is used to download the required modules and install them in the local public repository. It requires source and destination resolvers which are usually specified in a settings file (Listing 1).

The **`ibiblio`** resolver is for the Maven repository. The destination resolver, **`public-repo-resolver`**, describes the location (**`${dest.dir}`**), path (**`[organisation]/[module]/[type]s`**) and file format (**`[artifact]-[revision].[ext]`**) the modules will be written to for the local public repository.

`commons-lang` [12] is a good example of a module in the Maven repository that can be easily installed in a local public repository (Listing 2).

`dest.dir` specifies the location of the public repository and the **`ivy:settings`** target specifies the location of the settings file. Creating

Listing 1

```
<!--  public-repo-settings.xml -->
<ivysettings>
  <resolvers>
    <ibiblio name="ibiblio" m2compatible="true" />
    <filesystem name="public-repo-resolver">
      <artifact pattern="${dest.dir}/
          [organisation]/[module]/[type]s/
          [artifact]-[revision].[ext]"/>
    </filesystem>
  </resolvers>
</ivysettings>
```

Listing 2

```
<!--  build.xml -->
<project default="create-public-repo"
   xmlns:ivy="antlib:org.apache.ivy.ant">
  <target name = "create-public-repo" >
    <property name="dest.dir"
       value="C:/.../htdocs/ivy/public" />
    <ivy:settings id="repo.settings"
       file="public-repo-settings.xml"/>
    <ivy:cleancache/>

    <ivy:install  settingsRef="repo.settings"
                  organisation="commons-lang"
                  module="commons-lang"
                  revision="2.0"
                  from="ibiblio"
                  to="dest-resolver"
                  overwrite = "true"
                  haltonfailure = "yes"
                  transitive="true"/>

    <ivy:cleancache/>
  </target>
</project>
```

a repository using **`ivy:install`** resolves all modules to the local cache. This can cause problems when using the Ivy Ant client or IvyDE later on, so it's best to clean it out before and after the install using the **`ivy:cleancache`** task. However, if you are trying to resolve a number of dependencies for a module, as you'll see with Hibernate shortly, it is worth commenting **`ivy:cleancache`** out so that modules are only downloaded once.

The **`ivy:install`** target is given the settings id so it can find the settings file, the organisation, name and revision of the module and the source and destination resolvers. **`overwrite`** instructs the task to overwrite any previous installations of the module, **`haltonwrite`** instructs the task to overwrite any previous installations of the module, **`haltonfailure`** stops the install process if there is an error and **`transitive`** tells **`ivy:install`** to download and install all of the modules dependencies.

Running **`build.xml`** downloads and caches **`commons-lang`** locally and then installs it in the public repository. If you look in the `Ivy` directory in Apache's `htdocs` directory you will see a new directory called `public`. Inside `public` you will find the structure shown in Listing 3.

The `ivy-2.0.xml` file describes the module and its artifacts. `commons-lang-2.0.jar` is obviously the **`commons-lang`** JAR, the `.md5` files contain a checksum that can be used to verify the modules once they are downloaded and the `.sha1` files contain a hash function that can be used for secure download.

At this point **`commons-lang`** is installed in the local public repository and ready to use, but before I describe how to configure the Ivy client to use it, I am going to look at a more complex example.

`commons-lang` is a very simple module without any dependencies. More complex modules, such as Hibernate [13] are dependant on a number of other modules and these should also be installed in the local public repository. Fortunately this is what the **`transitive`** attribute is for:

Listing 3

```
commons-lang
  commons-lang
    ivys
      ivy-2.0.xml
      ivy-2.0.xml.md5
      ivy-2.0.xml.sha1
    jars
      commons-lang-2.0.jar
      commons-lang-2.0.jar.md5
      commons-lang-2.0.jar.sha1
```

```
<ivy:install settingsRef="repo.settings"
             organisation="org.hibernate"
             module="hibernate"
             revision="3.2.5.ga"
             from="ibiblio"
             to="dest-resolver"
             overwrite = "true"
             haltonfailure = "yes"
             transitive="true"/>
```

Unfortunately, due to the badly configured Maven repository some of the modules fail to install. This is where the Ivy documentation really falls down as it suggests that you should '...download those artifacts manually, and copy them to your destination repository to complete the installation'. You could do that, but there are a couple of other things to try first. The failing modules are:

- **commons-attributes.commons-attributes-compiler-2.1**
- **javax.security.jacc-1.0**
- **javax.transaction.jta-1.0.1B**

commons-attributes.commons-attributes-compiler-2.1 can be installed, simply by specifying an **ivy:install** task for it:

```
<ivy:install  settingsRef="repo.settings"
              organisation="commons-attributes"
              module="commons-attributes-compiler"
              revision="2.1"
              from="ibiblio"
              to="dest-resolver"
              overwrite = "true"
              haltonfailure = "yes"/>
```

Unfortunately, both **javax.security.jacc-1.0** and **javax.transaction.1.0.1B** are too badly configured in Maven to be resolved like this, so they have to be downloaded manually, but they can be installed using **ivy:install**, complete with Ivy XML files, rather than just being copied into the public repository.

First the **javax.security.jacc-1.0** and **javax.transaction.jta-1.0.1B** JARs need to be downloaded, renamed as `javax.security.jacc-1.0.jar` and `javax.transaction.jta-1.0.1B.jar` respectively and put into a temporary location (e.g. `C:\Temp\repo-src`). Then a new resolver needs to be added to the Ivy settings file to enable **ivy:install** to locate the JARs (see Listing 4).

Then a **src.dir** property needs to be created in `build.xml` and set to `C:/Temp/repo-src` and then **ivy:install** tasks can be created for **javax.security.jacc-1.0** and **javax.transaction.jta-1.0.1B** (Listing 5).

Listing 4

```
<!--  public-repo-settings.xml -->
<ivysettings>
  <resolvers>
    <ibiblio name="ibiblio" m2compatible="true" />
    <filesystem name="dest-resolver">
      <artifact pattern="${dest.dir}/
         [organisation]/[module]/[type]s/
         [artifact]-[revision].[ext]"/>
    </filesystem>
    <filesystem name="local-resolver">
      <artifact pattern="${src.dir}/
         [organisation].[artifact]-[revision].
         [ext]"/>
    </filesystem>
  </resolvers>
</ivysettings>
```

Listing 5

```
<ivy:install  settingsRef="repo.settings"
              organisation="javax.security"
              module="jacc"
              revision="1.0"
              from="local-resolver"
              to="dest-resolver"
              overwrite = "true"
              haltonfailure = "yes"/>

<ivy:install  settingsRef="repo.settings"
              organisation="javax.transaction"
              module="jta"
              revision="1.0.1B"
              from="local-resolver"
              to="dest-resolver"
              overwrite = "true"
              haltonfailure = "yes"/>
```

With the **ivy:install** tasks for **commons-attributes.commons-attributes-compiler-2.1**, **javax.security.jacc-1.0** and **javax.transaction.jta-1.0.1B** added along with the task for **org.hibernate.hibernate-3.2.5.ga**, Hibernate can be successfully installed in the public repository by running `build.xml`.

Configuring Ivy to use a local public repository

To demonstrate the use of the local public repository I am going to use the example from 'Java Dependency Management with Ivy – Part I':

```
import org.apache.commons.lang.WordUtils;
public class IvyAnt
{
  public static void main(String[] args)
  {
    final String msg = "hello, world!";
    System.out.println(
       WordUtils.capitalizeFully(msg));
  }
}
```

As you can see this code uses the **WordUtils** class from **commons-lang** to capitalise a string. The Ivy file is shown in Listing 6 and will download **commons-lang** from the Maven repository by default.

To get Ivy to request **commons-lang** from your local public repository you need to override its default `ivysettings.xml` file and provide your own. The default `ivysettings.xml` file is included in Ivy's JAR file and looks like Listing 7.

Briefly, here the public, shared and local repository resolver configuration files are specified as well as files that describe the order in which they should be used. A more detailed description is provided in the 'Adjusting Default Settings' documentation.

Listing 6

```
<!-- ivy.xml -->
<?xml version="1.0" encoding="ISO-8859-1"?>
<ivy-module version="2.0"  xmlns:xsi=
   "http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="http://ant.
   apache.org/ivy/schemas/ivy.xsd">
  <info organisation="Purple Tube"
     module="IvyAnt" status="integration"/>
  <dependencies>
    <dependency org="commons-lang"
                name="commons-lang"
                rev="2.0"
                conf="default"/>
  </dependencies>
</ivy-module>
```

To override the `ivysettings.xml` from the Ivy JAR, create a file called `ivysettings.xml` in the same place as `ivy.xml` and copy the xml in Listing 7 into it. The settings file will be automatically picked up by the Ivy Ant task, but you'll need to tell IvyDE about it by:

1. Right clicking on '`ivy.xml [*]`' and selecting properties.
2. Going to the Main tab and ticking 'Enable project specific settings'.
3. Entering the path to `project:///ivy-settings.xml` into the Ivy settings path.
4. Clicking Ok.

Listing 7

```
<ivysettings>
  <settings defaultResolver="default"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-public.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-shared.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-local.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-main-chain.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-default-chain.xml"/>
</ivysettings>
```

Listing 8

```
<!-- ivysettings-purple-public.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ivysettings>
<ivysettings>
  <resolvers>
    <url name="public">
    <artifact pattern="http://myserver/ivy/public/
      [organisation]/[module]/[type]s/[artifact]-
      [revision].[ext]" />
    <ivy pattern="http://myserver/ivy/public/
      [organisation]/[module]/ivys/ivy-
      [revision].xml"/>
    </url>
  </resolvers>
</ivysettings>
```

Listing 9

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ivysettings>
<ivysettings>
  <settings defaultResolver="default"/>
  <include url="ivysettings-purple-public.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-shared.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-local.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-main-chain.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-default-chain.xml"/>
</ivysettings>
```

Listing 10

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ivysettings>
<ivysettings>
  <settings defaultResolver="default"/>
  <include url="http://localhost/ivy/ivysettings-
     purple-public.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-shared.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-local.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-main-chain.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-default-chain.xml"/>
</ivysettings>
```

Next you need a file telling Ivy how to find dependencies in the local public repository (Listing 8).

The XML in Listing 8 specifies a new url resolver that can be added to Ivy's settings. The artifact tag tells Ivy where to look for the dependency (`http://myserver/ivy/public`) and what format the name of the JAR and the Ivy file will be in. In this case **`[organisation]`** relates to the **`org`** attribute of the Ivy dependency XML tag, **`[artifact]`** and **`[module]`** relate to the **`name`** attribute, **`[revision]`** relates to the **`rev`** attribute and **`[ext]`** defaults to `jar`.

The XML in Listing 8 needs to go into another file called something like `ivy-purple-settings.xml` and can then either go together with `ivy.xml` and `ivysettings.xml` or, more sensibly I think, into the `ivy` directory in the Apache `htdocs` directory of the repository. Either way you need to modify `ivysettings.xml` so Ivy can find it (Listing 9 or Listing 10).

With these files in place Ivy and IvyDE will now be able to find **`commons-lang`** in the local public repository and download it to the cache.

Setting up a shared repository

There is no need to set up a shared repository unless you have something to put in it. You could, if you wanted to, use your public repository as the shared repository. However, I think it is good to keep third party and proprietary modules separate. You may even want to keep them on separate servers, or create a shared repository and continue to use the Maven repository as your public repository. Ivy gives you lots of options!

The Microsoft SQL Server JDBC driver is a good example of a library that is not hosted in the Maven Repository and should therefore be put in a shared repository. At the time of writing Hibernate 3.3.2 is another good example. However, writing client code to demonstrate the use of either is quite verbose, so I am going to use a custom JAR instead.

My custom JAR, `net.purpletube.goodmusic-0.1.jar` contains a single class (see Listing 11) that is also dependent on the **`WordUtils`** class from the Apache **`commons-lang`** library. Therefore any client that uses the **`FavoriteAlbum`** class will have dependencies on both `net.purpletube.goodmusic-0.1.jar` and `commons-lang.jar`. `net.purpletube.goodmusic-0.1.jar` is not, of course, in the Maven Repository or the local public repository, but `commons-lang.jar` is. Therefore `net.purpletube.goodmusic-0.1.jar` should be published to a shared repository, with a dependency on `commons-lang.jar`, but `commons-lang.jar` should not.

Before `net.purpletube.goodmusic-0.1.jar` can be published, it needs an Ivy file to describe it and its dependent modules, as shown in Listing 12. This XML specifies the organisation, name and version of the module, the artifacts (JARs in this case) to be published and the dependency on **`commons-lang`**.

Listing 11

```
package net.purpletube.goodmusic;
import org.apache.commons.lang.WordUtils;
public class FavoriteAlbum
{
  public String getTitle()
  {
    return "ROMULOUS";
  }
  public String getArtist()
  {
    return WordUtils.capitalizeFully("EX DEO");
  }
}
```

Listing 12

```
<!-- net.purpletube.goodmusic-0.1.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="2.0">
  <info organisation="net.purpletube"
        module="goodmusic" revision="0.1"/>
  <publications>
    <artifact name="goodmusic" type="jar"
              ext="jar" conf="default"/>
  </publications>
  <dependencies>
    <dependency org="commons-lang"
                name="commons-lang"
                rev="2.0"
                conf="default"/>
  </dependencies>
</ivy-module>
```

Listing 13

```
<!--  ivysettings-repo.xml -->
<ivysettings>
  <resolvers>
    <filesystem name="local-resolver">
      <artifact pattern="${src.dir}/
         [organisation].[artifact]-[revision].
         [ext]"/>
      <ivy pattern="${src.dir}/[organisation].
         [module]-[revision].xml"/>
    </filesystem>
    <filesystem name="dest-resolver">
      <artifact pattern="${dest.dir}/
         [organisation]/[module]/[type]s/
         [artifact]-[revision].[ext]"/>
    </filesystem>
  </resolvers>
</ivysettings>
```

Listing 14

```
<target name = "create-shared-repo" >
  <property name="dest.dir"
            value="C:/.../htdocs/ivy/shared" />
  <property name="src.dir"
            value="C:/Temp/repo-src" />
  <ivy:settings id="repo.settings"
                file="shared-repo-settings.xml"/>
  <ivy:cleancache/>
    <ivy:install  settingsRef="repo.settings"
                  organisation="net.purpletube"
                  module="goodmusic"
                  revision="0.1"
                  from="local-resolver"
                  to="dest-resolver"
                  overwrite = "true"
                  haltonfailure = "yes"/>
  <ivy:cleancache/>
</target>
```

ivy:install and **ivy:publish** can both be used to create a shared repository. **ivy:install** is easier to use, so I'll describe it here. As with installing the local public repository, you need a source resolver and a destination resolver in another settings file (Listing 13).

As with the local public repository, **local-resolver** describes the file name format (**[organisation].[artifact]-[revision].[ext]**) and location (**${src.dir}**) of modules to be installed. The **dest-resolver** describes the location (**${dest.dir}**), path (**[organisation]/[module]/[type]**s) and file format (**[artifact]-[revision].[ext]**) the modules will be written to.

The install task is configured as shown in Listing 14.

As with the public repository **ivy:install** task, **src.dir** specifies the location of the dependency to install to the shared repository, **dest.dir**

Listing 15

```
import net.purpletube.goodmusic.FavoriteAlbum;
public class GoodMusicClient
{
  public static void main(String[] args)
  {
    final FavoriteAlbum
       fav = new FavoriteAlbum();
    final StringBuilder
       buf = new StringBuilder(fav.getTitle());
    buf.append(" by ");
    buf.append(fav.getArtist());
    System.out.println(buf);
  }
}
```

Listing 16

```
<ivy-module version="2.0"
   xmlns:xsi=
     "http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation=
     "http://ant.apache.org/ivy/schemas/ivy.xsd">
  <info organisation="purpletube.net"
        module="IvyAnt" status="integration"/>
  <dependencies>
    <dependency org="net.purpletube"
                name="goodmusic"
                rev="0.1"
                conf="default"/>
  </dependencies>
</ivy-module>
```

specifies the location of the shared repository and the **ivy:settings** target specifies the location of the settings file. The **ivy:install** target is given the settings id so it can find the settings file, the organisation, module and revision of the dependency and the source and destination resolvers. **overwrite** instructs the task to overwrite any previous installations of the dependency and **haltonfailure** stops the install process if there is an error. You'll notice that **transitive** is missing, this is because we don't want **common-lang** installed to the shared repository.

To install `net.purpletube.goodmusic-0.1.jar`, simply copy it and `net.purpletube.goodmusic-0.1.xml` to `C:\Temp\repo-src` and run the Ant script. If you look in the `Ivy` directory in Apache's `htdocs` directory you will see a new directory called `shared`. Inside there you will find the following structure, just like in the public repository:

```
net.purpletube
  goodmusic
    ivys
      ivy-0.1.xml
      ivy-0.1.xml.md5
      ivy-0.1.xml.sha1
    jars
      goodmusic-0.1.jar
      goodmusic-0.1.jar.md5
      goodmusic-0.1.jar.sha1
```

The `ivy-0.1.xml` file describes the dependency. `goodmusic-0.1.jar` is obviously the **net.purpletube.goodmusic** JAR, the `.md5` files contain a checksum that can be used to verify the module once it is downloaded and the `.sha1` files contain a hash function that can be used for secure download.

The shared repository is now set up, installed and ready to use.

Configuring Ivy to use a shared repository

To demonstrate the use of the shared repository we need a piece of client code that uses the **FavoriteAlbum** class (Listing 15) and the appropriate Ivy file (Listing 16).

Note that only `net.purpletube.goodmusic-0.1.jar` is specified. To get `commons-lang-2.0.jar`, we're relying on the fact that the

Writing the article

The first part of Java Dependency Management with Ivy was easy to write as using the Ivy clients is easy. The only complication I had was a new version of Ivy and IvyDE coming out while I was writing it! Creating a repository and getting to grips with the terminology and Ivy's poor documentation is quite demanding. It's not that Ivy doesn't have documentation, in fact it has quite a lot, it's just that it's so badly written with lots of assumption and very little detail.

This is the third version of this article. The first version described how to set up a very simple shared repository, without using `ivy:install` and then I intended to adjust it for dependencies of the modules installed in it. That turned out to be confusing, so I adjusted it to use `ivy:install` and handled the dependencies of the modules from the start.

Then I sent it over to Mr Jez Higgins for a read and he pointed out that he would like to know how to set up an alternative public repository. Due to bugs in the Maven Repository, configuring this is not straight forward. The Ivy documentation touches on this, but doesn't really address it properly or give a proper solution. So in this third version I look at creating an alternative public repository that doubles as a shared repository for your own modules.

shared repository knows that `net.purpletube.goodmusic.jar` depends on it.

As with the local public repository you need a file telling Ivy how to find dependencies in the shared repository (Listing 17).

Put this alongside `public-repo-settings.xml` in the `ivy` directory in the Apache `htdocs` directory. Then modify the local `ivysettings.xml` file to use the `public-repo-settings.xml` rather than the default shared repository (Listing 18).

With these files in place Ivy and IvyDE will now be able to find `net.purpletube.goodmusic-0.1.jar` in the shared repository and download it to the cache. It will also see that `net.purpletube.goodmusic-0.1.jar` depends on commons-lang and that it is not in the shared repository, so will download it from the local pubic repository.

Running **`GoodMusicClient`** will write **`ROMULOUS by Ex Deo`** [14] to the console, proving that, although not straightforward, creating Ivy repositories is quite easy to do. ■

Listing 17

```
<!-- shared-repo-settings.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ivysettings>
<ivysettings>
  <resolvers>
    <url name="shared">
      <artifact pattern="http://myserver/ivy/
          shared/[organisation]/[module]/[type]s/
          [artifact]-[revision].[ext]" />
      <ivy pattern="http://myserver/ivy/shared/
          [organisation]/[module]/ivys/ivy-
          [revision].xml"/>
    </url>
  </resolvers>
</ivysettings>
```

Listing 18

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ivysettings>
<ivysettings>
  <settings defaultResolver="default"/>
  <include url="http://myserver/ivy/ivysettings-
     purple-public.xml"/>
  <include url="http://myserver/ivy/ivysettings-
     purple-shared.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-local.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-main-chain.xml"/>
  <include url="${ivy.default.settings.dir}/
     ivysettings-default-chain.xml"/>
</ivysettings>
```


Acknowledgements

Thank you to Shawn Castrianni, Geoff Clitheroe, Joshua Tharp and Tom Widmer of the Ivy Users list for helping me when I could not see the wood for the trees, and Jez Higgins and Steve Love for review and direction.

References

[1] Grenyer, Paul 'Java Dependency Management with Ivy – Part1', *CVu 21-4*, September 2009.
[2] 'Ivy, The Agile Dependency Manager': http://ant.apache.org/ivy/
[3] Ant: http://ant.apache.org/
[4] Eclipse IDE: http://www.eclipse.org/
[5] Ivy Eclipse Plugin: http://ant.apache.org/ivy/ivyde/
[6] Maven Repository: http://mvnrepository.com/
[7] Microsoft SQL Server JDBC Driver: http://msdn.microsoft.com/en-us/data/aa937724.aspx
[8] 'Adjusting Default Settings': http://ant.apache.org/ivy/history/2.1.0-rc1/tutorial/defaultconf.html
[9] 'Building a Repository': http://ant.apache.org/ivy/history/latest-milestone/tutorial/build-repository.html
[10] 'Ivy Terminology': http://ant.apache.org/ivy/history/latest-milestone/terminology.html
[11] Apache Webserver: http://httpd.apache.org/
[12] Apache commons-dbcp Library: http://commons.apache.org/lang/
[13] Hibernate: https://www.hibernate.org/
[14] Ex Deo: http://www.myspace.com/exdeo

Charming the Snake

Steve Love makes his Python programmes more modular.

After quite a while of just 'playing around' with Python, and getting small programs working, I decided it was time to do the thing properly and understand how more significant projects fit together. This article is about how Python modules and packages work, how the various import statements behave, and how all of this relates to structuring significantly sized projects. An important part of any project is the part marked 'unit tests', and I want to explore how to fit those tests neatly into your structure without compromising it, while still making the most of being able to fake or mock features appropriately.

I'm certainly no *Pythonista*, merely an interested hobbyist really, and I'm sure my background in C++ and C# development projects shows, so much of what I describe here is about my journey towards getting it working. I'd certainly be very interested to learn how I've done it wrong! Correct me at cvu@accu.org – of course.

I won't be assuming much Python knowledge, although being basically familiar with the syntax of Python programs will be helpful. On the other hand, this isn't a Python tutorial, either, although I'll try and explain most Python terms as I go.

I used Python 2.6 [1] for the examples here; I don't think there is very much version-specific code, but some changes may be needed for older installations (in particular, `string_thing.format()` is new in 2.6).

The declaration of a class or function is considered executable code, so *every* Python script is executable to the processor

Bits of Python

The basic unit of a Python program is the python file, which contains a script. This program may be broken down as lines of executable code, functions and classes. The declaration of a class or function is considered executable code, so *every* Python script is executable to the processor. Of course, it may not be especially useful if the contents of the function or class are never invoked! Listing 1 shows a very simple Python program.

Listing 1

```
import sys

name = "world"
if len( sys.argv ) > 1:
  name = sys.argv[ 1 ]
greeting = "Hello, {0}".format( name )
print( greeting )
```

If you run this from a command line, you can supply arguments, and `name` takes up the first one, or the program provides a default for the variable `name`.

Even in such a simple example, we see the need and use for Python modules external to our program. In this case the `sys` module contains `argv` (amongst other things) which is used to get basic access to arguments passed to the program. By contrast, `len` is a standard function (the standard library is imported automatically), and `print` is a built-in keyword [2] like `if` and `import`, so neither require further imports or qualification.

As I mentioned, function declarations are executable code too. In fact the code *inside* a function is only checked for basic syntax – that it parses. It is only when the function is invoked that its correctness is verified:

```
def nonsense():
  this.compiles( but, wont, run )
```

Attempting to call this function will result in various errors at runtime (`this` isn't defined, neither are `but`, `wont` or `run`), but syntactically it's valid. The interpreter simply adds the name of the function to its collection of callable objects in the current namespace when it sees the `def nonsense():` line.

Class declarations are handled slightly differently; generally, classes contain data and methods, but *instance* data is not declared as part of the class – it's declared in the special `__init__` member function. A class's member function (or method) declarations behave like other function declarations. Code that *isn't* a method declaration gets invoked as part of the class declaration, and creates class-scoped objects:

```
class rubbish( object ):
  def putOut( self ):
    so.what()
  i = 10
```

The `i` object above is a class object (unbound to a specific instance, so a bit like `static` in C++ and C#), and `putOut` is a method. So, this class is fine until `putOut` is called on an instance of `rubbish`, which will result in similar errors to the example above.

Modules and packages

As we've already seen with the `sys` module, you can save your scripts away and re-use them in other programs. This is just a matter of saving the file so that you can refer to it later in an `import` statement; the name of the file (minus the extension) is the name of the module, which forms its own namespace. Declarations in that module, therefore, need to be qualified with the namespace name. Hence, `sys.argv`. A different form of the import statement allows use of the names *without* qualification:

```
from sys import argv
```

allows `argv` to be used without the `sys` prefix. This form also allows multiple names to be imported from the module in one go:

```
from sys import argv, exit, path
```

Python looks for modules in specific locations: the current directory (as indicated by the main program module), and the Python search path [3], *in that order*. Therefore, naming a module that clashes with a system module is a Bad Idea™ because your module will be loaded and used in preference to the system one.

One way of avoiding this particular problem is to group modules into packages. A package is a container of modules in the same way that a file-system directory is a container of files. A package *is* a directory, with a special file, `__init__.py`, which indicates that the modules within that directory are to be treated collectively as a package. This file can be empty, or may contain initialisation code if you wish.

The import statement 'runs' the module if it's the first time it's been imported into the program. This means executing all of the top-level

STEVE LOVE

Steve Love is an independent developer constantly searching for new ways to be more productive without endangering his inherent laziness. He can be contacted at steve@arventech.com

statements in the file, so, for example, importing the example program above would cause "Hello, world" to be printed to the output console. The contents of `__init__.py` for a package behave in the same way when importing the package. More commonly, a module will contain class and function declarations, so executing them brings the names into the current scope.

It's possible to check if a module is being run as a program or imported as a module by checking its name, made available through the **`__name__`** global identifier. If this identifier is set to **`__main__`**, then the module is being run, as shown below.

```
if __name__ == '__main__':
  do( stuff )
```

If it's the module name, then the module is being imported.

Camouflage

So what does the incantation **`import name`** mean? The **`name`** part in this case always identifies a module or a package. The identifier **`name`** is brought into scope, and its contents (either modules, functions, classes or objects, depending on whether it's a module or a package) can be referenced by preceding their names with the module or package name.

What about **`from module import name`**? In this case, **`name`** could be a function, object, class, module or package. It is, in fact, just a name of something. How it gets interpreted will determine whether it is valid or not. For example, if **`name`** is a module and you attempt to use it like this:

```
thing = name()
```

you'll get an error, because **`name`** is not callable. Importantly, the difference between these two forms means that the former brings the module or package name into scope, the latter only the final name:

```
import sys
from os.path import walk
```

Here, the contents of **`sys`** must be prefixed thus: **`sys.argv`**. The name **`walk`** has been imported into scope, but neither **`os`** nor **`path`** have been brought into scope as names. **`walk`** must therefore be used *unqualified*.

We'll skip over the **`from module import *`** version, since it's frowned upon, and doesn't always do what you expect [4]. In fact, we're in enough trouble already because the other two versions might not, either...

For example, given a package called **`store`**, and a module called **`db`**, these two imports are almost equivalent:

```
import store.db
# ...
from store import db
```

In either case, the name **`db`** is in scope. In the former, it must be referred to as **`store.db`**, in the latter, just **`db`** is fine.

Suppose however, you have a module called **`store`** in the current directory, defining a class called **`db`**. In this instance, the first call will fail (no such module **`db`**), and the second will import the *class* name into the current scope. Now suppose you also have a package called **`store`** which contains a module called **`db`**:

```
src/
    store/
        __init__.py
        db.py
    myprog.py
    store.py
```

What do the above import statements mean now? Well, the rules are quite specific; packages are always preferred to modules, so the module **`db`** is imported, *not* the class **`db`**.

The nearest bite

One last thing then. Watch closely because this is important. Consider the following directory tree:

```
src/
    store/
        __init__.py
        db.py
    test/
        __init__.py
        runtest.py
    prog.py
```

Suppose that `prog.py` (the main python program) contains **`from test import runtest`** and that `runtest.py` contains **`from store import db`**.

This works fine if you run `prog.py`, because the Python interpreter's path looks in the current directory *as determined by the running module*. If you execute `runtest.py` directly, the **`store`** package cannot be found because it appears in none of the current namespaces.

Now suppose that you have two packages with the same name at different levels in the directory structure:

```
src/
    store/
        __init__.py
        db.py
    test/
        store/
            __init__.py
            db.py
        runtest.py
    prog.py
```

Running `runtest.py` directly now is fine, because it finds a suitable **`db`** module in a suitable **`store`** package within its namespace. What about running `prog.py`, which imports **`runtest`**?

I find it easier to locate my source code when it's all in one place

The imported **`runtest`** module *no longer* finds the top-level **`store`** package, because there's a nearer one, right there. So **`test.store.db`** gets imported instead. Recall what I said earlier about hiding system modules? Well the same rule applies to your own modules, too.

One way of getting around this issue is by putting your modules and packages in the Python path or one of the standard module locations. I'm not suggesting you don't do this, but I find it easier to locate my source code when it's all in one place; modifying the path so that modules can be found is arguably *harder* than structuring your projects so that they just work. We'll look at this in a bit more detail in the later section about mock and fake objects. Of course, if you're authoring modules to be shared across many applications, then putting them in one of the standard places makes sense.

Be careful how you name your packages and modules, and where you put them, and be sure you're always getting what you expect.

Listing 2

```
def Worker( callable, x ):
  callable( x )

def Print( x ):
  print(x)

def Ignore( x ):
  pass

a = Print
b = Print
Worker( a, 10 )
Worker( b, "foo" )
Worker( Ignore, "foo" )
```

Listing 3

```
from application import myapp
import fakes.db
import unittest

def countItems( db ):
    #presume db.items is iterable only
    return len( [ i for i in db.items ] )

class applicationTest( unittest.TestCase ):
  def setUp( self ):
      self.db = fakes.db.store()
  def tearDown( self ):
      self.db.close()

  def testOpen( self ):
      app = myapp.app( self.db )
      self.assertTrue( self.db.isOpen )
      self.assertEqual(1, app.countConnections())
      self.assertEqual(10, countItems( self.db ))

if __name__ == '__main__':
  unittest.main()
```

The difference between a duck

Python is a dynamically typed language. In essence this means you don't have to name the *type* of your variables, they are deduced by the interpreter. Using such types is straightforward – if you try to do something that is not supported by the type (e.g. call a method) you get an error. Python programmers sometimes call this Duck Typing – if it walks, quacks and swims like a duck, it's *probably* a duck. The 'probably' is important when it comes to substitutability and fake objects used in testing. It's not just about types, either; Python has the notion of a 'callable', which can be called like a function [5]. A variable can be assigned to a function and the function called through that variable, so in Listing 2, **a** and **b** are *callable* in the same way that each of the functions defined are.

One point about the expected interface is that it needs only to be *sufficient*. In statically typed languages where a common interface definition is used, the derived tpes need to implement all of the interface. In a dynamically typed environment, only those parts of the interface that get used in context are required. C++ template aficionados will be familiar with this technique, since template mechanism provides precisely the same kind of duck-typing present in Python.

Listing 4

```
src/
    application/
        __init __ .py
        myapp . py
    comms/
        __init __ .py
        3ghandler . py
        loopbackhandler . py
    fakes/
        __init __ .py
        comms. py
        store . py
    store/
        __init __ .py
        db. py
    ui/
        __init __ .py
        mainwin . py
        commselectdlg . py
    program . py
    testapp . py
    testcomms . py
    teststore .py
    testutilities .py
```

Dynamic typing goes further than this. If you don't need to create an instance of a class, or refer directly to a function, i.e. you've had an instance or a callable respectively *given* to you, then you do not need to import the module it came from.

Consider this simple example:

```
def search( root, walker, output ):
  for path, dirs, files in walker.walk( root ):
    output.send( files )
```

Here, our function is using what looks very much like the system function **os.walk**, and some other class with a **send** method that accepts a list of strings. For the sake of the argument, let's say it's a class called **comms** in a module called **network**. If we'd named these two things directly in the code, we would also have to import their modules, tying this code to those specific implementations. As it is, the calling code can pass in **walker** and **output**, and the code will work provided they exhibit the correct interface.

Making a mockery of it

Python's unit testing facilities are pretty easy to use, and I won't go into a great deal of detail here. The standard tools that come with Python are in the **unittest** module, and follow a similar pattern to xUnit tools in other languages. Listing 3 has a simple example.

This test program can be run directly from the command line [6]. Note that the test methods are all prefixed with **test** which allows the automatic test discovery mechanism to work. Recall the section on picking the most local module or package name now, because this determines where the modules **application** and **fakes.db** are found. There is a (simple) helper function **countItems()**, whcih is used by the test(s).

In a dynamically typed environment, only those parts of the interface that get used in context are required

If the above example is in a project root level file, then **fakes** (possibly a module, possibly a package) also needs to be at the same location, or in the Python path, to be found. Similarly, **myapp.app** looks like a class found in a module called **myapp** in a package called **application**.

```
src/
    application/
        __init__.py
        myapp.py
    fakes/
        __init__.py
        db.py
    program.py
    test.py
```

Having all tests in one file might be OK if they are relatively few. Similarly, having a package of fake objects might be OK from the root if the main program only consists of a single program file. However, larger projects rarely look like this. Listing 4 is perhaps more conceivable.

Now it starts to look cluttered and difficult to separate tests from application code. Apart from naming conventions, that is.

The **testutilities** module might contain helper functions only used by the tests, such as the **countItems()** function used above. Where such functions are shared across more than one test module, putting those helper functions in one place can be useful. Of course, helper functions that are only used by a single **TestCase** object should probably reside in the same module as that object.

Some restructuring – and careful attention to naming – can help a great deal. (Listing 5.)

A test module – `tests.py` – is still required at the top-level. In order for the tests to be able to import modules and packages from the main

Listing 5

```
src/
    application/
        __init __ .py
        myapp . py
    comms/
        __init __ .py
        3ghandler . py
        loopbackhandler . py
    store/
        __init __ .py
        db. py
    ui/
        __init __ .py
        mainwin . py
        commselectdlg . py
    tests/
        __init __ .py
        fakes/
            __init __ .py
            comms. py
            store .py
        shared/
            __init __ .py
            testutilities .py
        testapp . py
        testcomms . py
        teststore .py
    program . py
    tests .py
```

application level, a separate test harness is needed, which just imports the test modules. Despite this (small) nit, I think this structure is much easier to understand. Most noticeably, the **fakes** package is now located with the tests, and all tests are together. The top-level `tests.py` program simply imports the necessary test modules, and runs them. E.g.

```
import unittest
from tests.testapp import applicationTest
from tests.testcomms import commsTest
from tests.teststore import storetest

if __name__ == '__main__':
  unittest.main()
```

For a straightforward test-harness this isn't bad. We still have to name the test classes so that the mechanics of **unittest.main()** will work without extra information, but that can be made more sophisticated and 'discover' tests for you [7].

The `testapp.py` module might contain similar code to the example in Listing 3, but without the line containing **unittest.main()** – I hope it's clear now that this will not work with our new project layout. Of course, the `shared.testutilities` module would now need to be imported.

The contents of the `myapp.py` module might conceivably be like this:

```
class app( object ):
  def __init__( store ):
    self.__store = store
    self.__store.open()
    self.__countConnections = 1

  def countConnections( self ):
    return self.__countConnections
```

Notably, it does *not* import the **store** package, or the **db** module within it. Clients of the **app** class (in this example, the **testapp** module, and the Python program itself in `program.py`) will need to import a class which conforms to the interface expected by **app**, and pass it in to its initialiser.

Listing 6

```
class store( object ):
  def open( self ):
      self.open = True
      return True
  def close( self ):
      self.open = False
      return True

  @property
  def isOpen( self ):
    return self.open

  @property
  def items( self ):
    return range( 10 )
```

The system of modules and packages employed by Python makes it easy to break your application into components

Which brings us finally to the fake store class. It's nothing very sophisticated (Listing 6).

This implements *just enough* of the **db** interface to allow the tests to run. Presumably, the complete production interface is much richer.

Round up

An important – even vital – part of any development environment is its ability to manage significant projects' components and the dependencies between them. Python has excellent facilities for breaking large projects up into manageable pieces and gives you, the programmer, great flexibility in how you choose to structure your code.

The system of modules and packages employed by Python makes it easy to break your application into components. Dynamic 'duck' typing makes dependency management straightforward, too, since you often do not need to hard-wire the packages and modules required by different parts of the code; provided the component can be passed as a parameter, then as long as its interface is as expected by the code that uses it, all will be well.

This flexibility comes with responsibility too; responsibility to your fellow programmers, and people who read your code (with the usual caveat that it'll probably be you some time hence!), to structure your projects so they are easy to follow, and don't do unexpected things. Clear code is about much more than the actual syntax and internal structure. Project structure is key to making your code clear, maintainable, and flexible. ■

Acknowledgements

Many thanks to Frances Buontempo, Pete Goodliffe, Roger Orr and Ric Parkin for their suggestions and corrections.

References

[1] www.python.org
[2] The **print** statement becomes a built-in function in Python 3.0
[3] The PYTHONPATH system variable defines what this is
[4] http://www.python.org/doc/essays/pages.html See 'Importing * From a Package'
[5] Classes can define a **__call__** method which allows an instance to be called like a function
[6] The **main** function of **unittest** has a lot of mechanics to discover properly-named test functions.
[7] See the Python documentation on TestLoader objects

Deciding Between IF and SWITCH When Writing Code

Derek Jones analyses some programmers' habits.

When writing software a common requirement is for the execution of some sequence of statements to depend on a variable having a particular value. Programming languages provide various constructs to support this requirement, e.g., the if-statement (which often supports checking against a single value) and the switch-statement (which supports the checking against a set of values). Measurements show that approximately every fifth statement is a selection statement.

This article investigates the possible factors that influence developers in their choice of selection statement, i.e., when deciding whether to use an if-statement or a switch-statement to implement some desired functionality. Two sources of data are analysed: measurements of existing code and the results of an experiment carried out at the 2009 ACCU conference.

This article has two parts, the first (this one) discusses measurements of **`if`** and **`switch`** statement usage in C source code, looking for differences in usage patterns; while a second one analyses the results of an experiment that asked subjects to write code whose behaviour depended on a variable that could take multiple values.

Language support for a switch-statement is unnecessary in the sense that it is always possible to write a sequence of if-statements that achieves the same effect. Reasons for supporting a switch-statement include ease of compiler optimizations [1] (i.e., for two equivalent sets of selection statements significantly less compiler implementation effort is needed to generate good quality code) and the belief that use of the **`switch`** form requires less developer effort and in some circumstances is less error prone.

As an example the following code (referred to as an *if-else-if* sequence, the expression appearing between parenthesis is the *controlling expression* and **`var`** is the *tested-expression*) where **`C_1`** and **`C_2`** are compile time constants:

```
if (var == C_1)
  stmt_seq_1;
else if (var == C_2)
  stmt_seq_2;
```

could be written as (provided none of the blocks in the above sequence contained a **`break`** statement; an **`else`** appearing in the second if-statement would be mapped to a **`default`**):

```
switch (var)
{
  case C_1: stmt_seq_1;
            break;
  case C_2: stmt_seq_2;
            break;
}
```

Common constraints on the use of the switch-statements include: the value must be known at translation time and that the value be representable as an integer type. Languages that do not have one or more of these constraints include PERL (using the **`Switch`** module) where the value need not be constant and can have any type for which equality is defined, and C# supports the use of string literals in case-labels.

DEREK JONES

Derek used to write compilers that translated what people wrote. These days he analyses code to try and work out what they intended to write. Derek can be contacted at derek@knosof.co.uk

Here we will limit ourselves to the situation where case-label values must be constant and representable in an integer type. Languages that have these constraints include Java, C and C++.

Application, algorithmic and evolutionary factors

The runtime execution decision represented by an **`if`** and **`switch`** statement is a consequence of an application or algorithmic requirement. An example of an application requirement is the handling of user input generated by the selection of an item from a list of options displayed by a program. An example of an algorithmic requirement is checking whether a value is within the bounds supported by the algorithm.

Almost no information is available on the break down between application and algorithmic requirements. Work on the Model C Implementation provides one data point for the break-down of applications vs. algorithmic if-statement usage. Every if-statement was tagged as being either as an application requirement (i.e., a requirement specified in the C Standard) or an algorithmic requirement [2]. Of the 4,329 if-statements (excluding the contents of the support library directories) 54.3% were tagged as applications requirements. This implementation differed from most compilers in that it performed no optimizations and so is likely to underestimate the percentage of if-statements attributable to algorithmic requirements. No break down by controlling expressions involving equality tests against constants was reported.

At a particular point in the code factors such as the number of conditionally executed statement sequences, the number and kind of values tested and the amount of code that depends on the decision made are likely to be outside the immediate control of the developer writing the code. The developer simply gets to decide how to write the code.

As it is being written code evolves and code that is part of an application that is *used* often evolves over a longer period than it took to originally write. The factors driving a developer's decision making process may be different between initial development and maintenance. During initial development, a switch-statement might be chosen if the author anticipates that the code will soon be updated to include additional sequences of conditionally executed statements. During maintenance, statements will be added and removed and the remaining code may be left untouched or affect the kind of code that is added. For instance, when extra if-statements are added to an existing if-else-if sequence, is the combined code refactored as a switch-statement; or when case-labeled statements are removed from a switch-statement, is the code rewritten as an if-else-if sequence?

Some coding guidelines recommend that **`default`** and **`else`** always be used, the intent being to catch unanticipated out-of-bounds conditions.

Factors influencing developer choice

The following list are some of the factors that might influence a developer's decision on whether to use an **`if`** or **`switch`** statement:

- Cognitive biases. Does a developer carry out enough analysis before writing code to have a reasonable idea of what is involved, then use this information to decide which selection statement is most appropriate, or does a developer use fast and frugal heuristics [3], or is the choice more stream-of-consciousness driven?

  Specific possibilities include:

- laziness, such as not investing the effort needed to accurately estimate the likely structure of the code being written or the perceived physical effort needed to write the code, e.g., amount of typing.
- unwillingness to change a course of action that has been embarked upon (i.e., a developer starts using an if-statement, because that is the common case, and continues to use it after discovering that multiple tests are involved).
- the current frame of mind. This might result in one kind of statement being used because it is written immediately after having written the same kind of statement, i.e., an if-statement will be preferred after another if-statement and a switch-statement preferred after or within a switch-statement.
- existing practices on the use of selection statements, i.e., what is commonly seen in source. For instance, while it is not incorrect to use a switch-statement where a single if-statement would suffice, based on existing practice such usage might be considered *unnatural* by developers. The opposite situation where a series of if-else-if-statements are used where a switch-statement could have been used is perhaps not viewed with the same degree of surprise.

- Developer expectations on the number of conditional arms expected to occur in the final code. Perhaps the probability of switch-statement being preferred increases as the number of arms expected increases.
- Developers are often driven by a desire to write efficient code and they have beliefs about whether a compiler is likely to generate higher quality code for one construct compared to another. The perceived runtime overhead of the expression being compared against may cause an increase in the probability of a switch-statement being preferred as this perceived overhead increases.
- Developer expectations on the number of different values that are likely to be compared against within the controlling expression of each conditional arm (e.g., three comparisons are needed for an arm that is executed if its tested-expression equals 4, 5 or 6).

  Perhaps the probability of switch-statement being preferred increases as the number of comparisons that are expected to be made increases.
- Developer expectations on the number of statements likely to be present in the conditionally executed arms.

Analysis of existing source

Measurements of **`if`** and **`switch`** statement usage in existing source can provide evidence about whether certain factors are likely to influence developer choice. This subsection discusses measurements of various kinds of **`if`** and **`switch`** statement usage in the visible form of a number of large C programs (e.g., gcc, idsoftware, linux, netscape, openafs, openMotif and postgresql).

The primary tool used to make these measurements was Coccinelle[4]. This tool converts the visible form of C source to an abstract syntax tree and provides a mechanism to specify patterns that match against this representation. The following is an extract from a pattern that matches a sequence of if-else-if statements, storing information on the position (line and column) of various constructs; other parts of this pattern write out the matched expressions **`E_1`** and **`E_2`** and their locations.

```
expression E_1, E_2;
statement S_1, S_2, S_3;
position p_1, p_2, p_3, p_4;
@@
if@p_1 (E_1)
  S_1
else@p_3 if@p_2 (E_2)
  S_2@p_4
else
  S_3
```

Conditional preprocessing directives, e.g., **`#if`**/**`#endif`**, are a significant source of problems for tools attempting to parse the visible source. Coccinelle is able to parse source containing these directives provided the conditional arms contain complete statements, declarations or expressions (measurements have shown this to be the majority of instances [5]). The version of Coccinelle used (0.1.10) internally handles conditional compilation directives in a way that causes some patterns to ignore selection statements containing such directives. The patterns used to measure statement counts (see Figure 5) are the only ones known to be affected by this behaviour.

if-statement characteristics

This subsection describes the process used to extract information about if-statement sequences that could be mapped to an equivalent switch-statement.

Constants

Most of the usage patterns being searched for require the controlling expression to contain one or more equality tests against a constant value. Symbolic names are often used to denote constant values in the visible source, these symbolic names might be defined as macros or enumeration constants. Macro names do not usually contain lower-case letters[5] and Coccinelle supports the matching of identifiers that don't contain any lower-case letters as a constant.

To validate the accuracy of constant detection, two sets of Coccinelle patterns were written: one requiring that at least one operand be a constant and the other having no such requirement. Comparing the output of the constant and non-constant pattern (7,727 non-constant occurrences for if-else-if and 2,742 occurrences for if-if, if-if is defined below), the constant pattern matched 77% (if-else-if) and 82% (if-if) of all possible matches. A manual check of the non-constant cases showed that while some were constant, but not treated as such because their name included lower-case letters, the number was sufficiently small that they could be ignored.

`NULL` usually denotes the null pointer constant, which is not a valid value in a case-label. Any if-statement sequence whose tested-expression was compared against this symbolic value was excluded from these measurements. In practice if-statement sequences containing this constant generally only occurred for sequences that involved a single tested-expression, see Figure 3.

Partial sequences

An if-else-if sequence may contain within it a subsequence that has the characteristics required for it to be mappable to a switch-statement. For instance in the following:

```
if (var == C_1)
   stmt_seq_1;
else if (var == C_2)
   stmt_seq_2;
else if (expr_1 != non_constant)
   stmt_seq_3;
```

the first two controlling expressions have the desired characteristics, but the third does not. Could this sequence be reordered so that the two expressions with the desired characteristics appeared last and so could be mapped to a switch-statement? Answering this question for most if-else-if sequences requires resources not available on this project and such subsequences were not included in the measurements reported here.

Only those if-else-if sequences whose controlling expressions all have the desired characteristics or those where the appropriate controlling expressions occurred last were included in the measurements reported here.

An if-if sequence differs from an if-else-if sequence in that replacing one of its subsequences by a switch-statement will not effect the control flow of any if-statements that appear immediately before or after it. The right plot of Figure 2 includes any mappable if-if subsequence, while the left

plot does not count an if-if sequence if it is immediately preceded or followed by a non-mappable if-statement.

if-else-if sequences

The process used to extract if-else-if sequences that are mappable to a switch-statement was as follows:

1. The controlling expressions in all if-else-if sequences were extracted. For instance, the three expressions **expr_1**, **expr_2** and **expr_3** would be extracted from the following code:

```
if (expr_1)
  stmt_seq_1;
else
  if (expr_2)
    stmt_seq_2;
else
  if (expr_3)
    stmt_seq_3;
```

2. Those controlling expressions in each sequence that all had one of the forms: equality test against a constant, a series of such equality tests combined using the logical-OR operator or a *between* operation implemented using relational operators and the logical-AND operator (combinations such as the constant appearing on the left-hand side and other ways of expressing *between* were checked for) were extracted. These expressions map to case-labels as shown in Figure 1, where **expr**'s token sequence is identical in every expression within the sequence (e.g., the expressions **x+y** and **y+x** were considered to be different).

Testing whether an expression having an unsigned type is less than some small constant is effectively a between operation. There was not sufficient time to measure this usage.

if-if sequences

A sequence of if-statements of the form (referred to as an *if-if* sequence here):

```
if (x == 1)
  stmt_1;
if (x == 2)
  stmt_2;
```

is equivalent to:

```
if (x == 1)
  stmt_1;
else if (x == 2)
 stmt_2;
```

if the value of **x** is not changed by the execution of **stmt_1**, or the last statement of **stmt_1** is a **return** statement.

This equivalence also holds when any statements appearing between the two if-statements can be moved to before or after the sequence without changing the external behavior of the program. An optimistic search (the simplifying assumptions made are likely to overestimate the number of occurrences) for if-statements separated by short sequences of unrelated statements found a small number of possible instances. The number found is sufficiently small that it can be ignored without significantly effecting the results.

It was not practical to fully analyse the consequences of executing **stmt_1** to deduce whether it resulted in the value of **x** being modified. The number of occurrences of if-if sequences based on the following three levels of analysis measured, see Figure 2.

1. assuming that **stmt_1** does not modify the value of **x**.
2. detecting some of those operations that could result in the value of **x** being modified (e.g., assignment, having its address taken, pre/post increment); any called functions were not analysed to deduce their effect on **x**.
3. treating any occurrence of **x** in **stmt_1** as a modification of its value.

The compound statement associated with the first if-statement of an otherwise mappable sequence ended with a **break** statement in 1% of occurrences. The percentage of **return** statements appearing in this context was 5% for if-else-if sequences and 36% for if-if.

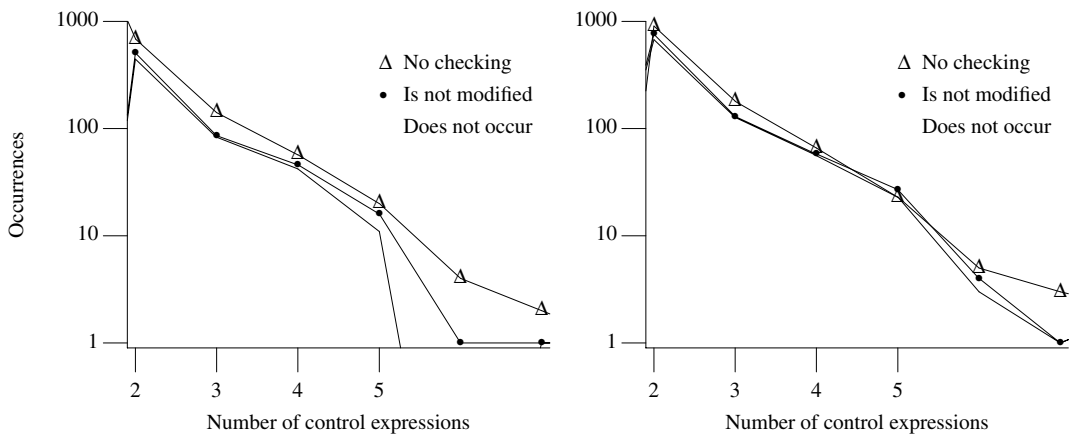
Figure 1

```
expr == constant                                ⇒ case constant:

expr == constant_1 || expr == constant_2 ⇒ case constant_1:
                                                  case_constant_2:

expr >= min_const && expr <= max_const    ⇒ case min_const:
                                                  case ...
                                                  case max_const:
```

Figure 2



Number of if-if sequences of a given length at three levels of analysis. The left plot only includes those if-if sequences where all of the if-statements are mappable to a switch-statement. The right plot includes any if-if sequences that is a subsequence of a longer, mappable, if-if sequence. In the case of a two if-statement if-if sequence, taking the *no checking* measurements as representing 100%, the percentage of uses where the block associated with the first if-statement does not contain any of the (checked) operations that modify the tested-expression is 75%/84% (left plot/right plot); the percentage of such sequences where the first block does not contain an instance of the tested-expression is 65%/75%.

switch-statement characteristics

The following is an example of the structure of Coccinelle patterns used to extract information on switch-statement usage (a separate pattern matched code where the last case-labels in a switch-statement were not followed in a jump-statement):

```
expression E_1, E_2;
position p_1, p_2, p_3;
@@
switch (E_1)@p_1
   {
   case ...:@p_2
            ...
(
            break;@p_3
|
            continue;@p_3
|
            return;@p_3
|
            return E_2;@p_3
)
   }
```

This pattern matches any sequence of code, within a switch-statement, that starts with a case-label and ends with a **break**, **continue** or **return** statement.

Any case-labeled statements that *fall through* to the following case-labeled statement are treated as-if they consisted of the statements associated with the fallen-into case-labels. Falling through to another sequence of statements is sufficiently rare that its effect on these measurements can be ignored.

default and final else

Use of **default** in a switch-statement is equivalent to a final **else** in an if-else-if sequence and the two constructs should be counted in the same way.

A default-label is sometimes prefixed to the same statement sequence as one or more case-labels; in the source benchmarks measured for this paper 5.7% of all **default**s were so prefixed [5]. This usage can occur through code evolution or a desire to explicitly map every named requirement in a specification to source code (this is sometimes handled via a comment). An example of this usage is:

```
case MEM_FAIL:
case DISC_FAIL:
default:        return ERR_CODE;
```

The nearest if-else-if sequence equivalent to the above code would be for the final controlling expression to perform equality tests on the case-label values and for the statement sequence in both of the conditional arms it controls to be the same. There are 25 instances where both arms of the final if-statement in a if-else contain the same statement sequence; a small number that has no significant affect on the measurement results.

Threats to validity

It is inevitable that the way in which selection statements are used will vary and some method of calculating the likely variation is needed if measurements of different constructs are to be compared in a meaningful way. For some of the measurements it was not possible to derive any statistical model and so no statistically meaningful comparisons can be made about the results obtained for those constructs.

Developers may make different decisions when writing new code compared to when modifying existing code. The C source measured for this study has been actively worked on for many years and during this time its selection statements are likely to have evolved. It is not possible to separate out any differences in this decision process for the measurements reported here (the experiment described in part 2 asked developers to write new code).

The only source code measured was written in C. To what extent is it possible to claim that the findings apply to code written in other languages? While there are no obvious reasons why the usage patterns found in C should not also occur in other languages, there is no reason why they should. Measurements of source written in other languages would help put this issue to rest.

The following are characteristics of the tool used, Coccinelle, and the patterns used that might be affected by these characteristics, i.e., the measurements might not be as accurate as expected

- the patterns used to measure selection statement arm length only counted statements, i.e., they did not include support for the presence of declarations, consequently any selection statement containing declarations in its conditional arm was not counted. It is known that approximately 10% of locally defined objects are defined in nested scopes [5], but the distribution of these definitions (i.e., whether the probability of them occurring increases as the number of statements in a block increases) is not known.
- the release of Coccinelle used (0.1.10) does not always match constructs containing conditional preprocessor directives. This affects the measurement of the number of statements contained within selection statements, resulting in those containing such directives being ignored.
- parsing C using incomplete information on identifiers (perhaps because the appropriate headers have not be processed) does not always succeed. Work has been done to reduce parse failures in Coccinelle when processing the Linux source. Parse failures will result in the associated construct being excluded from the matching process. The extent to which parse failures will skew the measurements, as opposed to uniformly reducing the number of matched constructs is not known.

Results

The measured source contains 29 times as many if-statements as switch-statements. Of the 384,749 if-statements measured 9.6% are contained within an if-else-if sequence and 15.9% of these sequences are mappable to a switch-statement. The corresponding values for the if-if sequence are 29.4% and 2.5% respectively.

There were 13,152 switch-statements measured and these contained a total of 63,395 **case** labels. A **default** label appeared in 49% of switch statements.

While the percentage of all if-statements that include an **else** arm is 22%, the percentage of if-else-if sequences having a final **else** arm averages out at around 50% and the final if-statement in an if-if sequence contains an **else** arm in around 20% of occurrences (this percentage rapidly drops as the sequence length increases).

Cognitive issues

Typing effort What is the difference in the amount of typing that needs to be done to create equivalent **if** and **switch** statements? In the following example underscores are used to denote characters that would appear in both forms of selection statement.

Comparing:

```
if (____ == __)
  _______
else if (expr == __)
  _______
```

with the equivalent switch-statement:

```
switch (____)
{
  case __: _______
           break;
  case __: _______
           break;
}
```

approximately twice as many non-whitespace characters occur in the switch-statement and depending on layout conventions there are also likely to be more than twice as many whitespace characters. The non-whitespace character ratio only comes down on the side of the switch-statement when a `break` is not needed, i.e., a `return` or `continue` statement terminates most of the labeled statement sequences.

Effects of local context

When writing code to what extent does the last choice of selection statement made by a developer affect the probability that the same choice will be made next time a selection statement is written?

There was insufficient time available to perform the local context source measurements that could help provide an answer to this question.

Number of conditional arms in construct

How does the number of conditional arms in an if-else-if or if-if sequence compare with the number of case-labeled statement sequences that can be jumped to in a switch-statement?

The differences between the left and right plots in Figure 3 are caused by contributions from `else` and `default`. In the case of an if-if sequence the `else` can only appear on the final if-statement and a default-label be mixed with case-labels on the same statement sequence.

In Figure 3 the solid lines are a least-squares fit of the data to an exponential function; [6] `switch` fitted over the range 3 to 11, if-else-if fitted over 1 to 7 and if-if fitted over 2 to 7 give the following respective equations (with x being the number of conditional arms; rounded to two decimal digits):

$$switch \propto e^{3.92-0.16x} \quad (1)$$

$$if\ else\ if \propto e^{4.96-0.58x} \quad (2)$$

$$if\ if \propto e^{3.97-0.52x} \quad (3)$$

These measurements suggest that the use of if-statement sequences decreases at approximately a fixed rate as the length of the sequence increases, with switch-statements taking up most of the slack for sequences of three or more.

The number of conditional arms that need to be written is likely to be controlled by factors that a developer has no influence over. The dotted line in Figure 3 is a fit of the sum of all selection statements containing a given number of conditional arms, the equation is given by:

$$all \propto e^{4.35-0.22x}$$

Expression runtime overhead

Function calls are generally perceived as having a high runtime overhead. Minimizing the number of function calls that need to be evaluated by the controlling expressions of a sequence of if-statements can be achieved either by assigning the result to a temporary variable that is then compared in each controlling expression or by using a switch-statement.

The percentage of all if-statement controlling expression containing a function call is 14% [5]. As Table 1 shows, the percentage of function calls in the selection statement sequences of interest in this paper is much lower. The percentage of function calls in switch-statement controlling expressions is higher than mappable if-else-if and if-if sequences. You author was not able to find a reasonable model that enabled the statistical significance of this difference to be estimated.

Table 1

Function calls appearing in the tested-expression of a controlling expression, as a percentage of the total number of occurrences of the associated selection statement.

Selection statement	Percentage containing function calls
switch	2.30%
if-else-if	0.47%
if-if	0.88%

For those expressions that do not include a function call the number of operators that need to be evaluated at runtime is one possible measure of perceived runtime overhead. <tableref href="control_comp"> shows that approximately 90% of tested-expressions do not contain any unary or binary operators; this figure drops to 60-70% if the `->` operator is counted.

The extent to which developers regard `->` as an operator that can have a non-trivial runtime overhead, compared to other object access expressions is not known.

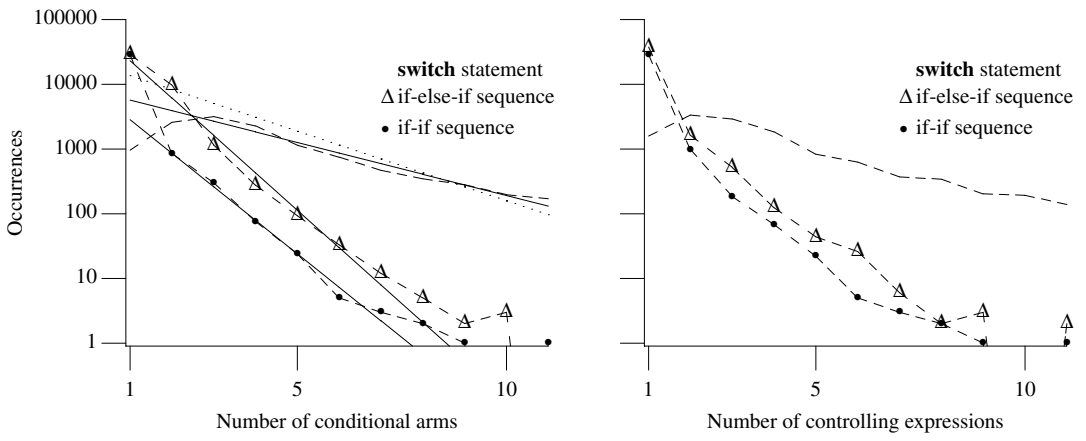
Table 2

The percentage of tested-expressions containing the given number of operators listed selection statement (all binary operators and unary excluding sizeof and casts). The second number in each column includes the -> operator in the total.

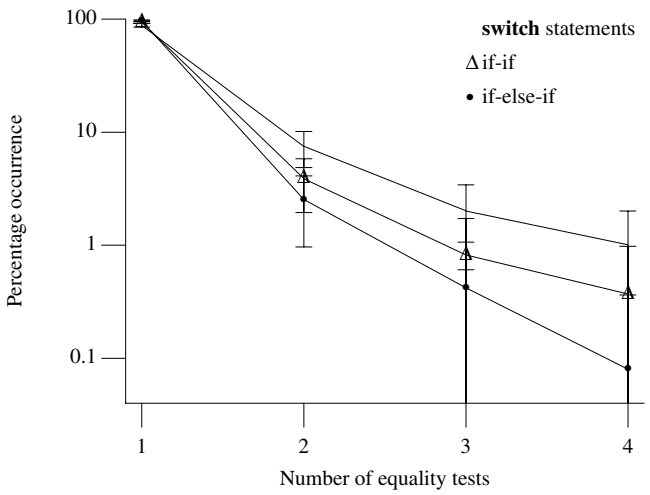
Selection statement	0	1	2
switch	89.3/59.7	8.7/35.1	1.9/4.5
if-else-if	89.8/67.3	9.3/29.9	0.9/2.7
if-if	85.9/70.8	13.5/26.2	0.5/3.0

It is tempting to observe that the number of operators in switch-statement and if-else-if sequence controlling expressions is very similar when the `->`

Figure 3



The left plot is of occurrences of if-else-if, if-if (uncertainty about which sequence a single if-statement, without an else-arm, belonged to was resolved by including it in both sequences) and switch-statements having the given number of conditional arms. The right plot is the number of controlling expressions in a if-else-if and if-if sequence and the number of case-labeled statement sequences (i.e., it excludes the effect of any default) in a switch-statement. The solid lines are a least-squares fit of the data to an exponential function. The dotted line is a fit to the sum of all sequences having a given number of arms.



Percentage of equality tests performed in the control expressions of if-else-if (97% had one) and if-if sequences (95% had one), and the number of case labels appearing together on the same statement (88% had one; any default label was not counted). Error bars are for a binomial distribution.

operator is excluded, but that when this operator is counted the most similar pairing is if-else-if and if-if. However, without a model of behaviour it is not possible to say anything statistically significant.

Number of equality tests controlling conditional arms

A controlling expression may map to multiple case-labels. For instance, each equality test in an expression containing one or more logical-OR operators is mapped to a separate case-label and a *between operation* implemented using a logical-AND operator and relational operators to compare against values within a range is mapped to the corresponding range of case-labels.

Take as an example the controlling expressions present in an if-if sequence. If each controlling expression is independent of the others, then the probability of two equality tests, for instance, occurring in any of these expressions is constant and thus given a large sample the distribution of two equality tests has a binomial distribution. The same argument can be applied to other numbers of equality tests and other kinds of sequence.

For each measurement point in Figure 4 the associated error bars span the square-root of the variance of that point (assuming a binomial distribution, for a normal distribution the length of this span is known as the standard deviation). The error bars overlap suggesting that the apparent difference in percentage of equality tests in each kind of sequence is not statistically significant.

Number of statements in conditional arm

How much code appears in the conditional arms of if-else-if, if-if sequences and case-labeled statement sequences?

Most of the Coccinelle patterns output line number information. This information can be used to calculate the difference in visible source lines between adjacent if-statement controlling expressions and between the last `case` labeling a given statement sequence and the final statement that caused the flow of control to leave the switch-statement. Those line number difference measurements appears in the left plot of Figure 5.

An example of line number difference is provided by the first if-else-if code fragment given in the introduction, where the difference between controlling expressions is 2; the distances in the first switch-statement example are both 1.

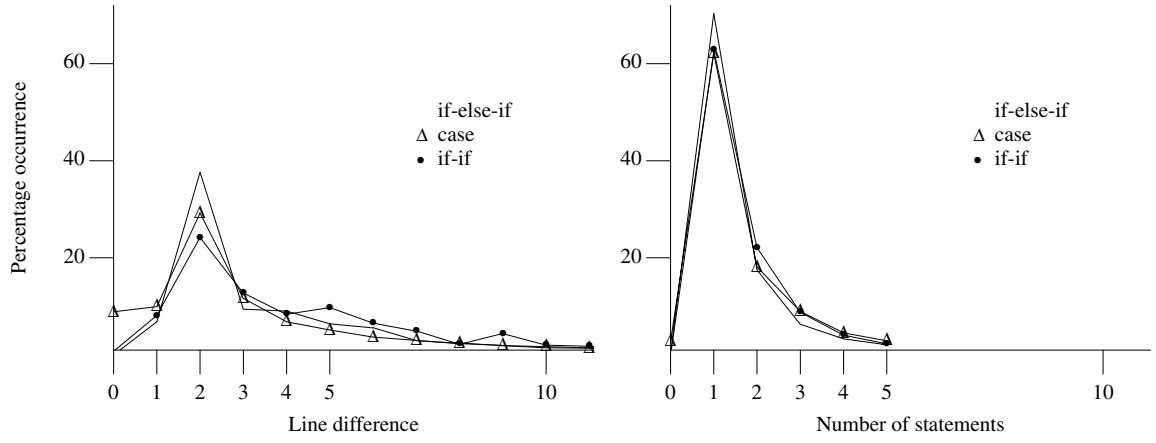
Because of the variety of different ways in which if-else-if, if-if and case-labeled statements can be visually laid out, line number differences may not be a consistent measure of the quantity of code present in the various language constructs.

The right plot of Figure 5 is based on counting statements. The Coccinelle patterns used did not count statements within nested blocks. This means, for instance, that a for-loop was counted as a single statement and any statements nested within its associated block did not contribute towards the total statement count.

The patterns used counted code containing a maximum of five statements and consumed over a day of cpu time; it was felt that measuring more statements was unlikely to change the pattern of usage seen in Figure 5. It is estimated that had longer sequences been counted the actual percentage of shorter sequences would have been around 20% lower..

Within a case-labeled statement sequence any terminating `break` statement is not included in the total statement count for that sequence. However, if the sequence is terminated by a `return` statement this statement is counted towards the total statement count for the sequence.

The two plots in Figure 5 suggest that if-else-if, if-if and case-labeled arms are very similar both in the visual number of lines and the number of top-level statements they contain.



Two methods of measuring the length of a conditionally executed arm are plotted.

The left plot is based on the difference in visible source line number between consecutive if tokens in an if-else-if and if-if sequence, and between a case token and the break, return or continue that terminates that arm.

The right plot is based on the number of top-level statements, that is statements contained within any nested block are not counted. Sequences containing more than five statements were not counted, resulting an overestimation of the actual percentage of shorter sequences.

Beyond Pipelining Programs in Linux

Unearthed Arcana (Part 1): Ian Bruntlett uncovers the back-tick.

If you're reading this, you're probably used to the command line. You can write individual programs and join them together using the `|` symbol.. For instance if you want to look at a directory's contents a page at a time, you would use a command like `ls | less`

Before I get into detail, I will introduce three commands.

1. `which` – follow the `PATH` environment variable and show which directory a command belongs to – e.g. which `ls` on my system results in `/usr/bin/less`
2. `strings` – looks at a file (typically a binary file) and outputs anything that looks like a text string.
3. `file` – looks at a file and report what kind of file it is. Example:

```
ian@Rutherford:~/c$ file /usr/bin/perl
/usr/bin/perl: ELF 32-bit LSB executable, Intel
80386, version 1 (SYSV), for GNU/Linux 2.6.8,
dynamically linked (uses shared libs), stripped
```

However, there is another way to add files together. Using the backtick operator (`), you can run a particular program (e.g. `which`) and put its output into the command line of the program you're interested in.

So, taking the above commands, we can use them together for some interesting things.

- `strings` `` `which ls` `` lists the strings embedded in the `ls` program.
- `file` `` `which perl` `` describes what kind of file perl is without the typist having to know where on the path. Example:

```
ian@Rutherford:~/c$ file `which perl`
/usr/bin/perl: ELF 32-bit LSB executable, Intel
80386, version 1 (SYSV), for GNU/Linux 2.6.8,
dynamically linked (uses shared libs), stripped
```

■

IAN BRUNTLETT

On and off, Ian has been programming for some years. He is a volunteer system administrator for a mental health charity called Contact (www.contactmorpeth.org.uk). As part of his work, Ian has compiled a free Software Toolkit (http://contactmorpeth/wikispaces.com/SoftwareToolkit).

Deciding Between IF and SWITCH When Writing Code (continued)

Discussion

Some of the measured characteristics where a notable difference was seen between `if` and `switch` statement usage include:

- Figure 5 suggests the number of conditional arms in the construct and/or the number of controlling expressions/case-labeled statement sequences have a large effect on the likelihood of a particular kind of selection statement being used. It is not possible to separate out the relative contributions of the number of controlling expressions and number of conditionally executed arms with the data available.
- the use of function calls in the tested-expression, see Table 1.

The characteristics that appear to have the largest effect on selection statement usage are the number of conditional arms in the construct and/or the number of controlling expressions/case-labeled statement sequences. The second part of this paper describes an experiment that asked subjects to write code based on specifications that involved different numbers of control expressions. ■

Further reading

A good introduction, at an undergraduate level, to the various algorithms people are thought to use when making decisions is provided by: *The Adaptive Decision Maker* by John W. Payne, James R. Bettman and Eric J. Bettman, published by Cambridge University Press; ISBN 0-521-42526-3.

Acknowledgements

The author wishes to thank everybody who volunteered their time to take part in the experiments and ACCU for making a slot available in which to run the experiment.

Thanks to Julia Lawall for suggestions on improving the Coccinelle patterns used for these measurements and responding very promptly to bug reports; also thanks to Yoann Padioleau for support using Coccinelle.

Notes and references

[1] G. Gigerenzer, P. M. Todd, and The ABC Research Group. *Simple Heuristics That Make Us Smart*. Oxford University Press, 1999.

[2] D. M. Jones. 'Who guards the guardians?' www.knosof.co.uk/whoguard.html, 1992.

[3] D. M. Jones. *The new C Standard: An economic and cultural commentary*. Knowledge Software, Ltd, 2005.

[4] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. 'Documenting and automating collateral evolutions in linux device drivers'. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems* 2008, pages 247–260, Mar. 2008.

[5] G.-R. Uh and D. B. Whalley. 'Effectively exploiting indirect jumps'. *Software–Practice and Experience*, 29(12):1061–1101, Oct. 1999.

[6] This data could have been fitted just as well with a power-law. Without a model describing developer behaviour it is not possible to distinguish a power-law from an exponential function and the latter was used to keep out the baggage that usually accompanies the former.

What is C++0x?

Bjarne Stroustrup concludes his tour of C++0x.

The first part of this series looked at changes to the core language of C++. This installment examines some of the library changes that build on those improvements.

This paper illustrates the power of C++ through some simple examples of C++0x code presented in the context of their role in C++. My aim is to give an idea of the breath of the facilities and an understanding of the aims of C++, rather than offering an in-depth understanding of any individual feature. The list of language features and standard library facilities described is too long mention here, but a major theme is the role of features as building blocks for elegant and efficient software, especially software infrastructure components. The emphasis is on C++'s facilities for building lightweight abstractions.

Concurrency and memory model

Concurrency has been the next big thing for about 50 years, but concurrency is no longer just for people with multi-million dollar equipment budgets. For example, my cell phone (programmed in C++ of course) is a multi-core. We don't just have to be able to do concurrent programming in C++ (as we have 'forever'), we need a standard for doing so and help to get concurrent code general and portable. Unfortunately, there is not just one model for concurrency and just one way of writing concurrent code, so standardization implies serious design choices.

Concurrency occurs at several levels in a system, the lowest level visible to software is the level of individual memory accesses. With multiple processors ('cores') sharing a memory hierarchy of caches, this can get quite 'interesting'. This is the level addressed by the memory model. The next level up is the systems level where computations are represented by tasks. Above that can be general or application-specific models of concurrency and parallel computation.

The general approach of C++0x is to specify the memory model, to provide primitive operations for dealing with concurrency, and to provide language guarantees so that concurrency mechanisms, such as threads, can be provided as libraries. The aim is to enable support for a variety of models of concurrency, rather than building one particular one into the language.

The memory model

The memory model is a treaty between the machine architects and the compiler writers to ensure that most programmers do not have to think about the details of modern computer hardware. Without a memory model, few things related to threading, locking, and lock-free programming would make sense.

The key guarantee is: Two threads of execution can update and access separate memory locations without interfering with each other. To see why that guarantee is non-trivial, consider:

```
// thread 1:
char c;
c = 1;
int x = c;
// thread 2:
char b;
b = 1;
int y = b;
```

For greater realism, I could have used separate compilation (within each thread) to ensure that the compiler/optimizer won't simply ignore `c` and `b` and directly initialize `x` and `y` with 1. What are the possible values of `x` and `y`? According to C++0x, the only correct answer is the obvious one: 1 and 1. The reason that's interesting is that if you take a conventional good pre-concurrency C or C++ compiler, the possible answers are 0 and 0, 1 and 0, 0 and 1, and 1 and 1. This has been observed 'in the wild'. How? A linker might allocate `c` and `b` in the same word – nothing in the C or C++ 1990s standards says otherwise. In that, C and C++ resemble all languages not designed with real concurrent hardware in mind. However, most modern processors cannot read or write a single character, it must read or write a whole word, so the assignment to `c` really is 'read the word containing `c`, replace the `c` part, and write the word back again'. Since the assignment to `b` is similarly implemented, there are plenty of opportunities for the two threads to clobber each other even though the threads do not (according to their source text) share data!

So naturally, C++0x guarantees that such problems do not occur for 'separate memory locations'. In this example, `b` and `c` will (if necessary on a given machine) be allocated in different words. Note that different bitfields within a single word are not considered separate memory locations, so don't share structs with bitfields among threads without some form of locking. Apart from that caveat, the C++ memory model is simply 'as everyone would expect'.

Fortunately, we have already adapted to modern times and every current C++ compiler (that I know of) gives the one right answer and has done so for years. After all, C++ has been used for serious systems programming of concurrent systems 'forever'.

Threads, locks, and atomics

In my opinion, letting a bunch of threads loose in a shared address space and adding a few locks to try to ensure that the threads don't stump on each other is just about the worst possible way of managing concurrency. Unfortunately, it is also by far the most common model and deeply embedded in modern systems. To remain a systems programming language, C++ must support that style of programming, and support it well, so C++0x does. If you know Posix threads or boost threads, you have a first-order approximation of what C++0x offers at the most basic level. To simplify the use of this fundamental (and flawed) model of concurrency, C++0x also offers

- thread local storage (identified by the keyword `thread_local`)
- mutexes
- locks
- conditions variables
- a set of atomic types for lock-free programming and the implementation of other concurrency facilities
- a notion of fine grain time duration

In my opinion lock-free programming is a necessity, but should be reserved for people who find juggling naked sharp swords too tame [1]. Importantly, the whole language and standard library has been re-specified (down to the last memory read or write) so that the effects of concurrency are well-specified – though of course not well defined: the result of a data race is not and should not be well defined (it should be prevented by the library or applications programmer).

As luck would have it, Anthony Williams has a paper 'Multi-threading in C++0x' in the current issue of *Overload* [2], so I don't have

BJARNE STROUSTRUP

Bjarne Stroustrup designed and implemented the C++ programming language. He can be contacted at www.research.att.com/~bs

Listing 1

```
template<class T, class V> struct Accum  {
// function object type for computing sums
  T* b;
  T* e;
  V val;
  Accum(T* bb, T* ee, const V& v) : b{bb}, e{ee},
     val{vv} {}
  V operator() () {
     return std::accumulate(b,e,val); }
};

void comp(vector<double>& v)
// spawn many tasks if v is large enough
{
  if (v.size()<10000) return std::accumulate(
     v.begin(),v.end(),0.0);

  auto f0 {
    async(Accum{v.data(),
    v.data()+v.size()/4,
    0.0})};
  auto f1 {
    async(Accum{v.data()+v.size()/4,
    v.data()+v.size()/2, 0.0})};
  auto f2 {
     async(Accum{v.data()+v.size()/2,
     v.data()+v.size()*3/4], 0.0})};
  auto f3 {
     async(Accum{v.data()+v.size()*3/4,
     v.data()+v.size(), 0.0})};

  return f0.get()+f1.get()+f2.get()+f3.get();
}
```

to go into details. Instead, I will give an example of a way for the programmer to rise above the messy threads-plus-lock level of concurrent programming (Listing 1).

This is a very simple-minded use of concurrency (note the 'magic number'), but note the absence of explicit threads, locks, buffers, etc. The type of the **f**-variables are determined by the return type of the standard-library function **async()** which called a **future**. If necessary, **get()** on a **future** waits for a **std::thread** to finish. Here, it is **async()**'s job to spawn **thread**s as needed and the **future**'s job to **join()** the appropriate **thread**s (i.e., wait for the completion of threads). 'Simple' is the most important aspect of the **async()**/**future** design; **future**s can also be used with threads in general, but don't even *think* of using **async()** to launch tasks that do I/O, manipulate mutexes, or in other ways interact with other tasks. The idea behind **async()** is the same as the idea behind the range-**for** statement: Provide a simple way to handle the simplest, rather common, case and leave the more complex examples to the fully general mechanism.

Please note that **future** and **async()** is just one example of how to write concurrent programs above the messy threads-plus-lock level. I hope to see many libraries supporting a variety of models, some of which might become candidates for C++1x. Unlike every other feature presented here, **async()** has not yet been voted into C++0x . That's expected to happen in October, but no man's life, liberty, or programming language is safe while the committee is in session (apologies to Mark Twain).

Standard library improvements

At the outset of the work on C++0x, I stated my ideal as 'being careful, deliberate, conservative and skeptic' about language extensions, but 'opportunistic and ambitious' about new standard libraries [3]. At first glance, the opposite happened, but when you count pages in the standard you find that the language sections grew by about 27% and the library sections by about 100%, so maybe it would be wrong to complain too loudly about lack of new standard library components. The most obvious improvements to the standard library are the added library components:

- Concurrency ABI:
  - **thread**
  - mutexes, locks, atomic types,
  - simple asynchronous value exchange: **unique_future**, **shared_future**, and **promise**
  - simple asynchronous launcher: **async()**
- Containers:
  - Hashed containers: **unordered_map**, **unordered_multimap**, **unordered_set**, **unordered_multiset**
  - Fixed sized array: **array**
  - Singly-linked list: **forward_list**
- Regular expressions: **regex**
- Random numbers
- Time utilities: **duration** and **time_point**
- Compile-time rational arithmetic: **ratio**
- Resource management pointers: **unique_ptr**, **shared_ptr**, and **weak_ptr**
- Utility components: **bind()**, **function**, **tuple**
- Metaprogramming and type traits
- Garbage collection ABI

Whatever project you do, one or more of these libraries should help. As usual for C++, the standard libraries tend to be utility components rather than complete solutions to end-user problems. That makes them more widely useful.

Another set of library improvements are 'below the surface' in that they are improvements to existing library components rather than new components. For example, the C++0x vector is more flexible and more efficient than the C++98 vector. As usual, the standard library is the first test of new language features: If a language feature can't be used to improve the standard library, what is it good for?

More containers

So, C++0x gets hash tables (**unordered_map**, etc.), a singly-linked list (**forward_list**), and a fixed-sized container (**array**). What's the big deal and why the funny names? The 'big deal' is simply that we have standard versions, available in every C++0x implementation (and in major C++ implementations today), rather than having to build, borrow, or buy our own. That's what standardization is supposed to do for us. Also, since 'everybody' has written their own version (proving that the new components are widely useful), the standard could not use the 'obvious' names: there were simply too many incompatible **hash_map**s and **slist**s 'out there', so new names had to be found: 'unordered' indicates that you can't iterate over an **unordered_map** in a predictable order defined by **<**; an **unordered_map** uses a hash function rather than a comparison to organize elements. Similarly, it is a defining characteristic of a **forward_list** (a singly linked list) that you can iterate through it forwards (using a forward iterator), but not (in any realistic way) backwards.

The most important point about **forward_list** is that it is more compact (and has slightly more efficient operations) than **list** (a doubly-linked list): An empty **forward_list** is one word and a link has only a one-word overhead. There is no **size()** operation, so the implementation doesn't have to keep track of the size in an extra word or (absurdly) count the number of elements each time you innocently ask.

The point of **unordered_map** and its cousins is runtime performance. With a good hash function, lookup is amortized O(1) as compared to map's O(log(N)), which isn't bad for smaller containers. Yes, the committee cares about performance.

Built-in arrays have two major problems: They implicitly 'decay' to pointers at the slightest provocation and once that has happened their size

Listing 2

```
template<class C, class V> typename
   C::const_iterator find(const C& a, V val)
{
  return find(a.begin(),a.end(),val);
}

array<int,10> a10;
array<double,1000> a1000;
vector<int> v;
// ...
auto answer = find(a10,42);
auto cold = find(a1000,-274.15);
if (find(v,666)==v.end()) cout << "See no evil";
```

is 'lost' and must be 'managed' by the programmer. A huge fraction of C and C++ bugs have this as their root cause. The standard-library array is most of what the built-in **array** is without those two problems. Consider:

```
array<int,6> a { 1, 2, 3 };
a[3] { 4 };
int x { a[5] }; // x becomes 0 because default
                // elements are zero initialized
int* p1 { a };  // error: std::array doesn't
                // implicitly convert to a pointer
int* p2 { a.data() };// ok: get pointer to first
                     // element
```

Unfortunately you cannot deduce the length of an **array** from an initializer list:

```
array<int> a3 { 1, 2, 3 };
// error: size unknown/missing
```

That's about the only real advantage left for built-in arrays over **std::array**.

The standard **array**'s features make it attractive for embedded systems programming (and similar constrained, performance-critical, or safety-critical tasks). It is a sequence container so it provides the usual member types and functions (just like **vector**). In particular, **std::array** knows its **end()** and **size()** so that 'buffer overflow' and other out-of-range access problems are easily avoided. Consider Listing 2.

Incidentally, have you ever been annoyed by having to write things like **typename C::const_iterator**? In C++0x, the compiler can deduce the return type of a simple function from its **return**-statement, so you can simplify:

```
template<class C, class V> [] sum(const C& a,
   V val)
{
  return find(a.begin(),a.end(),val);
}
```

You can read **[]** as 'function'; **[]** is a new notation to explicitly state that a function is being declared.

Better containers

I suspect that the new containers will attract the most attention, but the 'minor improvements' to the existing containers (and other standard library components) are likely to be the more important.

Initializer lists

The most visible improvement is the use of initializer-list constructors to allow a container to take an initializer list as its argument:

```
vector vs = { "Hello", ", ", "World!", "\n" };
for (auto s : vs ) cout << s;
```

This is shorter, clearer, and potentially more efficient than building the vector up element by element. It gets particularly interesting when used for nested structures:

```
vector<pair<string,Phone_number>> phone_book= {
{ "Donald Duck", 2015551234 },
{ "Mike Doonesbury", 9794566089 },
{ "Kell Dewclaw", 1123581321 }
};
```

As an added benefit, we get the protection from narrowing from the **{ }**-notation, so that if any of those integer values do not fit into **Phone_number**'s representation of them (say, a 32-bit **int**), the example won't compile.

Move operators

Containers now have move constructors and move assignments (in addition to the traditional copy operations). The most important implication of this is that we can efficiently return a container from a function:

```
vector<int> make_random(int n)
{
  vector<int> ref(n);
  for(auto x& : ref) x = rand_int();
  // some random number generator
  return ref;
}

vector<int> v = make_random(10000);
for (auto x : make_random(1000000))
   cout << x << '\n';
```

The point here is that – despite appearances – no **vector** is copied. In C++98, this **make_random()** is a performance problem waiting to happen; in C++0x it is an elegant direct solution to a classic problem, Consider the usual workarounds: Try to rewrite **make_random()** to return a free-store-allocated **vector** and you have to deal with memory management. Rewrite **make_random()** to pass the **vector** to be filled as an argument and you have far less obvious code (plus an added opportunity for making an error).

Improved push operations

My favourite container operation is **push_back()** that allows a container to grow gracefully:

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.push_back({s,i});
```

This will construct a **pair<string,int>** out of **s** and **i** and move it into **vp**. Note: 'move' not 'copy'. There is a **push_back** version that takes an rvalue reference argument so that we can take advantage of **string**'s move constructor. Note also the use of the unified initializer syntax to avoid verbosity.

Emplace operations

The **push_back()** using a move constructor is far more efficient than the traditional copy-based one in important cases, but in extreme cases we can go further. Why copy/move anything? Why not make space in the vector and then construct the desired value in that space? Operations that do that are called 'emplace' (meaning 'putting in place'). For example **emplace_back()**:

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.emplace_back(s,i);
```

An emplace function takes a variadic template (see the C++0xFAQ, [6]) argument and uses that to construct an object of the desired type in place.

Whether the `emplace_back()` really is more efficient than the `push_back()` depends on types involved and the implementation (of the library and of variadic templates). In this case, there doesn't seem to be a performance difference. As ever, if you think it might matter, measure. Otherwise, choose based on aesthetics: `vp.push_back({s,i})` or `vp.emplace_back(s,i)`. For now, I prefer the `push_back()` version because I can see that an object is being composed, but that might change over time. For new facilities, it is not immediately obvious which styles and which combinations will be the more effective and more maintainable.

Scoped allocators

Containers can now hold 'real allocation objects' (with state) and use those to control nested/scoped allocation (e.g. allocation of elements in a container). A rather sneaky problem can occur when using containers and user-defined allocators: Should an element's free-store allocated sub-objects be in the same allocation area as its container? For example, if you use `Your_allocator` for `Your_string` to allocate its elements and I use `My_allocator` to allocate elements of `My_vector` then which allocator should be used for string elements in `My_vector<Your_string>`? The solution is to tell a container when to pass an allocator to an element. For example, assuming that I have an allocator `My_alloc` and I want a `vector` that uses `My_alloc` for both the `vector` element and `string` element allocations. First, I must make a version of `string` that accepts `My_alloc` objects:

```
using xstring = basic_string<
// a string with my allocator
                              char,
                              char_traits<char>,
                              My_alloc<char>
                              >;
```

This use of `using` is new in C++0x. It is basically a variant of `typedef` that allows us to define an alias with the name being defined coming up front where we can see it.

Next, I must make a version of `vector` that accepts those `xstring`s, accepts a `My_alloc` object, and passes that object on to the `xstring`:

```
using svec = vector<
// a string with a scoped allocator
      xstring,
      scoped_allocator_adaptor<My_alloc<xstring>>
      >;
```

The standard library 'adaptor' ('wrapper type') `scoped_allocator_adaptor` is used to indicate that `xstring` also should use `My_alloc`. Note that the adaptor can (trivially) convert `My_alloc<xstring>` to the `My_alloc<char>` that `xstring` needs.

Finally, we can make a `vector` of `xstring`s that uses an allocator of type `My_alloc<xstring>`:

```
svec v {
   scoped_allocator_adaptor(
   My_alloc<xstring>{my_arena1})
};
```

Now `v` is a `vector` of `string`s using `My_alloc` to allocate memory for both `string`s and characters in `string`s.

Why would anyone go to all that bother with allocation? Well, if you have millions of objects and hard real-time requirements on your performance, you can get rather keen on the cost of allocation and deallocation and concerned about the locality of objects. In such cases, having objects and their subobjects in one place can be essential – for example, you may dramatically improve performance by simply 'blowing away' a whole allocation arena by a single cheap operation rather than deleting the objects one by one.

Resource management pointers

Resource management is an important part of every non-trivial program. Destructors and the techniques relying on them (notably RAII) are key to most resource management strategies. C++0x adds a garbage collection ABI to our toolset, but that must not be seen as a panacea: The question is how to combine destructor based management of general resources (such as locks, file handles, etc.) with the simple collection of unreferenced objects. This involves non-trivial challenges: for example, destructor-based reasoning is essentially local, scoped, and typed whereas garbage collection is based on non-local reasoning with little use of types (beyond knowing where the pointers are).

Since the mid 1980s, C++ programmers have used counted pointers of various forms to bridge that gap. Indeed, reference-counted objects were the original version of garbage collection (in Lisp) and are still the standard in several languages. They are supported by the standard library `shared_ptr` which provides shared ownership and `weak_ptr` which can be used to address the nasty problems that circular references causes for reference-counted pointers. All `shared_ptr`s for an object share responsibility for the object and the object is deleted when its last `shared_ptr` is destroyed. The simplest example simply provides exception safety:

```
void f()
{
  X* p = new X;
  shared_ptr<X> sp(new X);
  if (i<99) throw Z(); // maybe throw an exception
  delete p;
  // sp's destructor implicitly deletes sp's object
}
```

Here the object pointed to by `p` is leaked if an exception is thrown, but the object pointed to by `sp` is not. However, I am in general suspicious about 'shared ownership' which is far too often simply a sign of weak design and made necessary by a lack of understanding of a system. Consider a common use of a `shared_ptr`:

```
shared_ptr<X> make_X(int i)
{
  // check i, etc.
  return shared_ptr<X>(new X(i));
}

void f(int i)
{
  vector<shared_ptr<X>> v;
  v.push_back(make_X(i));
  v.push_back(make_X(j));
  // …
}
```

Here we use `shared_ptr` for three things:

- Getting a large object out of a function without copying
- Passing an object from place to place by passing a pointer to it without worrying who eventually needs to destroy it
- Having an owner of the object at all times so that the code is exception-safe (using RAII).

However, we didn't actually share that object in any real sense, we just passed it along in a reasonably efficient and exception-safe manner. The `shared_ptr` implementation keeps a use count to keep track of which is the last `shared_ptr` to an object. If we looked carefully, we'd see that the use count bob up and down between 1 and 2 as the object is passed along before the count finally goes to 0. If we *moved* the pointer around instead of *making copies*, the count would always be 1 until its object finally needed to be destroyed. That is, we didn't need that count! What we saw has been called 'false sharing'.

C++0x provides a better alternative to `shared_ptr` for the many examples where no true sharing is needed, `unique_ptr`:

- The `unique_ptr` (defined in `<memory>`) provides the semantics of strict ownership.
  - owns the object it holds a pointer to
  - can be moved but not copied

- stores a pointer to an object and deletes that object when it is itself destroyed (such as when leaving block scope).

- The uses of **unique_ptr** include
  - providing exception safety for dynamically allocated memory,
  - passing ownership of dynamically allocated memory to a function,
  - returning dynamically allocated memory from a function.
  - storing pointers in containers
- 'what **auto_ptr** should have been' (but that we couldn't write in C++98)

Obviously, **unique_ptr** relies critically on rvalue references and move semantics. We can rewrite the **shared_ptr** examples above using **unique_ptr**. For example:

```
unique_ptr<X> make_X(int i)
{
  // check i, etc.
  return unique_ptr<X>(new X(i));
}

void f(int i)
{
  vector<unique_ptr<X>> v;
  v.push_back(make_X(i));
  v.push_back(make_X(j));
  // …
}
```

The logic is inherently simpler and a **unique_ptr** is represented by a simple built-in pointer and the overhead of using one compared to a built-in pointer are miniscule. In particular, **unique_ptr** does not offer any form of dynamic checking and requires no auxiliary data structures. That can be important in a concurrent system where updating the count for a shared pointer can be relatively costly.

Regular expressions

The absence of a standard regular expression library for C++ has led many to believe that they have to use a 'scripting language' to get effective text manipulation. This impression is further enhanced because that a lack of standard also confounds teaching. Since C++0x finally does provide a regular expression library (a derivative of the **boost::regex** library), this is now changing. In a sense it has already changed because I use **regex** to illustrate text manipulation in my new programming textbook [4]. I think **regex** is likely to become the most important new library in terms of direct impact on users – the rest of the new library components have more of the flavor of foundation libraries. To give a taste of the style of the **regex** library, let's define and print a pattern:

```
regex pat (R"[\w{2}\s*\d{5}(-\d{4})?]");
// ZIP code pattern XXddddd-dddd and variants
cout << "pattern: " << pat << '\n';
```

People who have used regular expressions in just about any language will find **\w{2}\s*\d{5}(-\d{4})?** familiar. It specifies a pattern starting with two letters **\w{2}** optionally followed by some space **\s*** followed by five digits **\d{5}** and optionally followed by a dash and four digits **-h\d{4}**. If you have not seen regular expressions before, this may be a good time to learn about them. I can of course recommend my book, but there is no shortage of regular expression tutorials on the web, including the one for **boost::regex** [5].

People who have used regular expressions in C or C++ notice something strange about that pattern: it is not littered with extra backslashes to conform to the usual string literal rules. A string literal preceded by **R** and bracketed by a **[]** pair is a raw string literal. If you prefer, you can of course use a 'good old string literal': **"\\w{2}\\s*\\d{5}(-\\d{4})?"** rather than **R"[\w{2}\s*\d{5}(-\d{4})?]"**, but for more complex patterns the escaping can become 'quite interesting'. The raw string literals were introduced primarily to counteract problems experienced with using escape characters in applications with lots of literal strings, such as text processing using regular expressions. The **"[…]"** bracketing is there to allow plain double quotes in a raw strung. If you want a **]"** in a raw string you can have that too, but you'll have to look up the detailed rules for raw string bracketing (e.g. in the C++0x FAQ).

The simplest way of using a pattern is to search for it in a stream:

```
int lineno = 0;
string line; // input buffer
while (getline(in,line)) {
  ++lineno;
  smatch matches; // matched strings go here
  if (regex_search(line, matches, pat))
  // search for pat in line
    cout << lineno << ": " << matches[0] << '\n';
}
```

The **regex_search(line, matches, pat)** searches the **line** for anything that matches the regular expression stored in **pat** and if it finds any matches, it stores them in **matches**. Naturally, if no match was found, **regex_search(line, matches, pat)** returns **false**.

The **matches** variable is of type **smatch**. The '**s**' stands for 'sub'. Basically, a **smatch** is a vector of sub-matches. The first element, here **matches[0]**, is the complete match.

So what does this all add up to?

C++0x feels like a new language – just as C++98 felt like a new language relative to earlier C++. In particular, C++0x does not feel like a new layer of features on top of an older layer or a random collection of tools thrown together in a bag. Why not? It is important to articulate why that is or many might miss something important, just as many were so hung up on OOP that they missed the point of generic programming in C++98. Of course C++ is a general purpose programming language in the sense that it is Turing complete and not restricted to any particular execution environment. But what, specifically, is it good for? Unfortunately, I do not have a snazzy new buzzword that succinctly represents what is unique about C++. However, let me try:

- C++ is a language for building software infrastructure.
- C++ is a language for applications with large systems programming parts.
- C++ is a language for building and using libraries.
- C++ is a language for resource-constrained systems.
- C++ is a language for efficiently expressing abstractions.
- C++ is a language for general and efficient algorithms.
- C++ is a language for general and compact data structures.
- C++ is a lightweight abstraction language.

In particular, it is all of those. In my mind, the first 'building software infrastructure' points to C++'s unique strengths and the last 'lightweight abstraction' covers the essential reasons for that.

It is important to find simple, accurate, and comprehensible ways to characterize C++0x. The alternative is to submit to inaccurate and hostile characterizations, often presenting C++ roughly as it was in 1985. But whichever way we describe C++0x, it is still everything C++ ever was – and more. Importantly, I don't think that 'more' is achieved at the cost of greater surface complexity: Generalization and new features can save programmers from heading into 'dark corners'.

C++0x is not a proprietary language closely integrated with a huge development and execution infrastructure. Instead, C++ offers a 'tool kit' ('building block') approach that can deliver greater flexibility, superior portability, and a greater range of application areas and platforms. And – of course and essentially – C++ offers stability over decades. ■

Acknowledgements

The credit for C++0x goes to the people who worked on it. That primarily means the members of WG21. It would not be sensible to list all who

A Game of Cards with Baron Muncharris

Baron Muncharris suggests a game of cards.

Greetings Sir R-----! I must declare that it is a most remarkable coincidence to meet with you again in this establishment! Do I take it that you might again be tempted by a small wager to sweeten your wine?

Splendid! Let me recount the rules of a rather inventive card game played, with a little too much enthusiasm if you ask my opinion, by the mole men of Under Mongolia.

Regard this deck of cards from which I have ejected the jokers, knaves and nobles, leaving just the number cards. Each of these shall stand for its face value and, after the fashion of book-keepers, the black cards, that is to say the clubs and the spades, shall be considered positive and the red cards, namely the diamonds and the hearts, shall be considered negative.

By way of demonstration, the 5 of clubs shall stand for +5 and the 5 of hearts for -5 whereas the 3 of spades shall stand for +3 and the 3 of diamonds for -3 etc. It is a simple scheme that any child not dropped over-many times upon its head could grasp.

Another example you say? Oh, very well; the 7 of hearts shall stand for -7.

Now that I have described this in such excruciating detail that the average parakeet should understand it well enough to explain it to his grandchildren, I shall describe the nature of play.

What's that you ask? What shall the ace of spades stand for? It shall stand for +1 and I, sirrah, shall stand for no further interruption!

I shall deal us each 5 cards; mine face down and yours face up. Of yours, you must select 3 and lay them in an orderly row before me. I shall subsequently lay a row of 3 cards of my own face up beneath yours. We must then multiply the pair of numbers in each column and add together the 3 numbers thus produced. My goal shall be to contrive to play such cards that this sum is as close to zero as possible.

For example if you were to play 3♦, 4♠ and 2♣ and I were to riposte with 2♠, 3♥ and 9♣ then I should have a perfect score of zero since:

3♦	4♠	2♣
2♠	3♥	9♣
-6	-12	18

and -6+-12+18=0.

We shall immediately thereafter play a second hand in which you shall take upon my role and I yours. If, after this, my score is closer to zero than yours then I shall have the round. Likewise, if your score is the closer to zero then you shall have it. In the event that our scores are equally close to zero then the round shall be deemed a draw.

After some numerous rounds agreed upon in advance, he who has the most rounds shall be declared the winner and shall claim the stake.

When I explained these rules to my student – and I shudder at even this meagre hint of familiarity – acquaintance, he started babbling incoherently about some fellow named Victor, who was evidently once trapped in some small space in a tiger's domain, and the bearing of his predicament upon the rights of angels.

I assure you that I can speak with authority on these subjects on account of my considerable experience of both.

Of the former, I have some several times found it necessary to duel entire ambushes of tigers, most recently armed with naught but a gilded toothpick for a sword and a pocket kerchief for a cape; an escapade that has been widely reported in the press and that will not therefore bear another telling.

Of the latter, I have been introduced socially to not a few celestial courtiers and must confess that, in my lustier youth, I seduced more than one of their number; being of noble blood, I am restrained from elaboration.

Despite such excellent credentials, I cannot begin to fathom what possible relationship he supposed might exist between the one topic and the other, or for that matter, between either and this remarkable game!

I can only deduce that the louse-ridden sot is back on the gin, and that it has robbed him of his rather pedestrian powers of reason.

See here, I have dealt you your hand; 6♦, 2♣, 5♠, 4♥, 3♣.

Choose your 3 cards with a care to increase to the utmost your chance of taking the round whilst I call for more wine! ■

RICHARD HARRIS

Richard Harris has been a professional programmer since 1996. He has a background in Artificial Intelligence and numerical computing and is currently employed writing software for financial regulation.

What is C++0x? (continued)

contributed here, but have a look at the references in my C++0x FAQ: There, I take care to list names. Thanks to Niels Dekker, Steve Love, Alisdair Meredith, and Roger Orr for finding bugs in early drafts of this paper and to Anthony Williams for saving me from describing threads and locks.

References

[1] Damian Dechev and Bjarne Stroustrup: 'Reliable and Efficient Concurrent Synchronization for Embedded Real-Time Software.' *Proc. 3rd IEEE International Conference on Space Mission Challenges for Information Technology* (IEEE SMC-IT). July 2009. http://www.research.att.com/~bs/smc_it2009.pdf

[2] Anthony Williams: 'Multi-threading in C++0x.' *Overload 93*; October 2009.

[3] Bjarne Stroustrup: 'Possible Directions of C++0x.' ACCU keynote 2002. Note: No C++0x does not provide all I asked for then, but that's a different story.

[4] Bjarne Stroustrup: *Programming: Principles and Practice using C++*. Addison-Wesley. 2008.

[5] John Maddock: Boost.Regex documentation http://www.boost.org/doc/libs/1_40_0/libs/regex/doc/html/index.html

[6] Bjarne Stroustrup: 'C++0x FAQ'. www.research.att.com/~bs/C++0xFAQ.html . Note: this FAQ contains references to the original proposals, thus acknowledging their authors.

On a Game of Dice

A student analyses Baron Muncharris' Dice problem.

You will no doubt recall that the game involved rolling a die some number of times and receiving a sum of coin equal to the number of spots at each roll. Furthermore, in the event that a roll should show just 1 spot that the accumulated pot should thereafter be halved.

Had Sir R----- asked my counsel before playing the Baron's game I should have advised him to take the first bet and shun the second.

In reckoning the fair stake for any such game it is illuminating to consider the expected size of one's pot after any given round with the die.

Let us assume that after the n-1'th roll of the die, one has amassed a pot of x_{n-1}. One should expect to hold $\mathrm{E}(x_n)$ upon having completed the following n'th roll, where

$$\begin{aligned}\mathrm{E}(x_n) &= \frac{1}{6}\times\frac{1}{2}\times(x_{n-1}+1)+\frac{1}{6}\times(x_{n-1}+2)+\frac{1}{6}\times(x_{n-1}+3)+\ldots\\ &= \frac{1}{6}\times\left(5\frac{1}{2}x_{n-1}+20\frac{1}{2}\right)\\ &= \frac{11}{12}x_{n-1}+\frac{41}{12}\\ &= \frac{11}{12}\left(\frac{11}{12}x_{n-2}+\frac{41}{12}\right)+\frac{41}{12}\\ &= \left(\frac{11}{12}\right)^2 x_{n-2}+\frac{11}{12}\times\frac{41}{12}+\frac{41}{12}\\ &= \left(\frac{11}{12}\right)^2\left(\frac{11}{12}x_{n-3}+\frac{41}{12}\right)+\frac{11}{12}\times\frac{41}{12}+\frac{41}{12}\end{aligned}$$

Taking this to its limit, one has

$$\begin{aligned}\mathrm{E}(x_n) &= \sum_{i=1}^{n}\frac{41}{12}\times\left(\frac{11}{12}\right)^{n-i}\\ &= \frac{41}{12}\times\left(\frac{11}{12}\right)^{n-1}\times\sum_{i=0}^{n-1}\left(\frac{12}{11}\right)^{i}\end{aligned}$$

where the capital sigma stands for the sum of the terms on its right over the values of i from the lower to the upper.

The final sum is a geometric series and is consequently compelled to equal

$$\begin{aligned}\sum_{i=0}^{n-1}\left(\frac{12}{11}\right)^{i} &= \frac{1-\left(\frac{12}{11}\right)^{n}}{1-\frac{12}{11}}\\ &= \frac{\left(\frac{12}{11}\right)^{n}-1}{\frac{12}{11}-1}\\ &= 11\left(\left(\frac{12}{11}\right)^{n}-1\right)\end{aligned}$$

The expectation is therefore transformed to

$$\begin{aligned}\mathrm{E}(x_n) &= 41\times\left(\frac{11}{12}\right)^{n}\times\left(\left(\frac{12}{11}\right)^{n}-1\right)\\ &= 41\times\left(1-\left(\frac{11}{12}\right)^{n}\right)\end{aligned}$$

For a game of 10 dice, this equates to a pot of approximately 23.82, and is thus worth playing for a stake of 23¾. But no matter how many dice may be rolled, the expected winnings cannot reach 41 and hence the second game is not one a man well versed in reckoning should willingly enter.

Monsieur L----- [1] suggested an interesting change to the rules in which the player might elect to cease playing if he so desires. If that player is a sensible man, he should do so only if he expects to be worse off after the next roll of the die, implying that he should quit the game if and only if

$$\mathrm{E}(x_n) < x_{n-1}$$

or equivalently

$$\begin{aligned}\frac{11}{12}x_{n-1}+\frac{41}{12} &< x_{n-1}\\ \frac{41}{12} &< \frac{1}{12}x_{n-1}\\ x_{n-1} &> 41\end{aligned}$$

Hence, he should bow out as soon as the pot exceeds 41, which is perhaps not so very surprising since this is, after all, the greatest expected conclusion to any such game.

And what should be a fair stake for such a game?

The formula that yields the expected size of the pot after a roll of the die is now

$$\mathrm{E}(x_n) = \begin{cases}\frac{11}{12}x_{n-1}+\frac{41}{12} & x_{n-1}\le 41\\ x_{n-1} & x_{n-1}>41\end{cases}$$

Alas, whilst this is evidently not a complicated formula, it no longer represents a straight line and is consequently beyond my power to extrapolate to a tidy conclusion.

Nevertheless, one can be certain that the stake will be higher under these new rules since if the pot ever exceeds 41 one can be sure that it shall not subsequently be reduced.

Moreover, since the expected winnings from a game of 10 dice are a fair bit less than that stopping point, the stake should not be expected to be very much greater than that of the original game. Indeed, after some careful calculation, I found it to exceed the original by a shade over 3 parts in 800 of a coin.

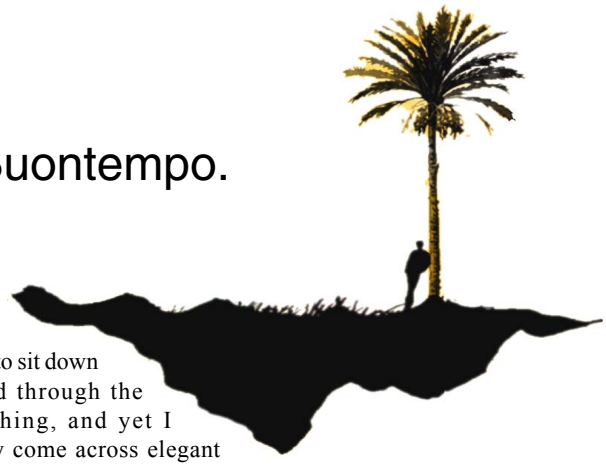
What it should be for a 100 dice game, I shall leave to a more patient calculator. ■

Notes

[1] With thanks to Louis Lavery for the suggestion.

Desert Island Books

Paul Grenyer maroons Frances Buontempo.



Where do I start with Frances Buontempo? The beginning is probably the best place. I think I first became aware of her on accu-general and somehow it came to light that she was living and working in Leeds and had a liking for alternative music. Having spent several years studying and working in Leeds, I sent her an email asking if she'd like to come along to the Wendy House[1] the next time I was going and she said she would.

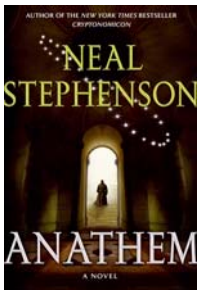
Before I had a chance to get to Leeds, the ACCU conference came around (must have been 2006) and I remember walking into one of the rooms at the Randolf and seeing an uncomfortable looking person trying to hide in the corner (although she still claims she wasn't trying to hide!). We've been friends since and after lots of interviews at various investment banks, sometimes for the same position, Frances was offered a position at BarCap and a couple of weeks later I was offered one at Lehmans.

I got married and moved away from London a year later, but we're still friends, bump into each other at various ACCU events, sometimes Wendy House (although it's been a while) and the Bloodstock Metal Festival[2] each year.

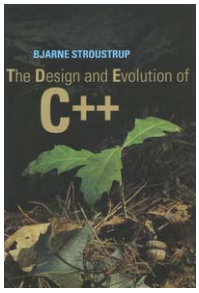
Frances Buontempo

A desert island you say? Music? I demand *Tin Omen* by Skinny Puppy – the words are delicious nonsense. And *Killing Joke*, by Killing Joke – their second eponymous album. But I want some KMFDM as well. And something I can sing along to, maybe Ian Dury. And there must be Laibach. Oh, just two albums? OK, *Jesus Christ Superstar* by Laibach and something by J.S.Bach. Or Front Line Assembly. No I can't decide. That's far too hard.

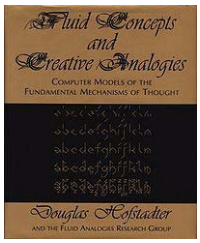
Books? Let's get this straight – five books? One a novel, and four programming books. Great. I presume I am allowed the Bible and the complete works of Shakespeare? Good show. Now, I've been trying to read *Anathem* by Neal Stephenson for a while now. I was delighted by his Baroque trilogy, and I'd be tempted to buy that, but I suspect you might not allow three one thousand or so page books to count as a single choice. They had a beautiful mix of the history of science, mathematics and finance entwined with tales of pirates and adventures on the high seas. Watching accurate, well researched details of the fights over notation of the calculus between Leibniz and Newton unfold was a delight. Of course, Newton's involvement introduces us to coinage, alchemy and the founding of the Bank of England. And therefore, of course, more pirates. The historical details are lovingly presented. As a mathematician, I love novels that contain such under-told treasures. There are so many other mathematically inspired stories. Abbott's *Flatland*, obviously. Is his middle name really Abbott? Who calls their child Edwin Abbott Abbott? But I digress. Mathematical story books. Oh, and films too, such as *Cube*[3] and *Pi*[4]. Genius. Don't forget Rudy Rucker[5]. Oh now look – I am in danger of buying more books now. 'Alma is swept away into a higher world of mathematician cockroaches and cone shells bent upon using our world as an experimental set-up for deciding an arcane point of metamathematics.' (*Mathematicians in Love* by Rudy Rucker). Botherations. I must read that. So anyway, I won't take any of those books. I'll take *Anathem* since I haven't finished reading it yet.



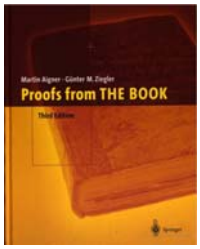
Now to the non-fiction. I've never read Knuth's *The Art of Computer Programming*. The box set does count as one book, right? I would love the time to sit down and read through the whole thing, and yet I regularly come across elegant algorithms that are attributed to Knuth. I've never read Stroustrup's *The Design and Evolution of C++* either, which I suspect would be a pleasure to read, having talked to others who have.



Now, I suspect you wanted me to tell you why I had chosen a book based on having actually read it. So, time to consider some books I have read. I started reading Douglas Hofstader's *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought* a while ago. Hofstader has a particular take on human thought, consciousness and freewill. He conflicts with Roger Penrose on this, which has led to several other interesting books. I personally think they are both incorrect. Nonetheless, aside from the underlying philosophical viewpoint on thought, the *Fluid Concepts* book contains many puzzles and shows how to program a computer to start solving them. It also explores language use. Several of you will have read his *Gödel Escher Bach: An Eternal Golden Braid*. This is a far less technical book than *Fluid Concepts*, and also a delight. It introduced a couple of translations of Lewis Carroll's 'Jabberwocky'. The idea of translating a nonsense poem is extremely interesting. It has the appeal of many pure mathematics problems: abstraction, generalisation and a degree of rank stupidity, giving rise to further interesting ideas. *Fluid Concepts* formalises some of these approaches and would inspire me to write several programs. So I hope you allow me pen and paper to jot down ideas as I read. I never finished reading it, because I want to devote time to exploring the ideas it raises. A desert island holiday would certainly come in handy.



OK, that's three books I've never read. Well, two I've never read and one I started reading. Perhaps I should select one book I have read as my final choice. I currently have Meyers' *More Effective C++*, Vandervoorde and Josuttis' *C++ Templates*, Gamma et al's *Design Patterns*, and Hull's *Options, Futures and other Derivatives* on my desk at work. They are all excellent books – though I find Hull has a lack of balance, swinging from in-depth explanations of the addition and multiplication aspects of pricing some financial products to gigantic hand waviness over the more complicated mathematics. The other books are brilliant and all C++ programmers have read them. Or will do at some point. However, those can stay on my desk. How difficult. There are so many good books out there. I am tempted by Teuvo Kohonen's *Self-Organising Maps*, and by Marvin Lee Minsky and Seymour Papert's *Perceptrons*, and by Thom Mitchell's *Machine Learning*. These three books are classics on artificial intelligence. All three were clearly written and a pleasure to read. *Perceptrons* is frequently cited as demonstrating that you cannot model XOR (exclusive or) with a single perceptron (a simple node in an early neural network). This promoted



Inspirational (P)articles

Frances Buontempo introduces Andrew Holmes' inspiration.

During an ACCU London social, those present were on the verge on sinking into despair and cynicism, but Andrew Holmes managed to lift the tone by remembering that computing machines are cool and can be the source of great joy. He shares with us details about a marvelous machine that made him smile.

The original HP 12C financial calculator, introduced in 1981, is HP's best selling product.

It uses Reverse Polish notation input, which is way faster than algebraic mode on other calculators. The stack is made up of four registers and you can swap the top two and rotate the stack. It's also much faster and more convenient than a spreadsheet for simple stuff.

It's also got a cool, 80s retro look.

I've just bought the 2003 updated version, since it's one of only two models of calculator that can be used on the Chartered Financial Analyst exams.

Very cool toy and actually carefully designed to solve a problem well.

Desert Island Books (continued)

much further work and led to far more powerful neural networks. However, I read the book from cover to cover and XOR is not mentioned once. It's remarkable to note how rumours circulate even in academic work. However, back to my final book choice. It has to be *Proofs from the Book* by Martin Aigner and Günter M. Ziegler. This book collects together various magnificent mathematical proofs, inspired by Paul Erdos' claim that God keeps a book of the most elegant proofs. He is reputed to have said 'You don't have to believe in God, but you should believe in the book'. It has simpler proofs, such as showing e is irrational, to more-brain aching combinatorics problems. Some theorems are proved in more than one way, because sometimes many beautiful proofs of the same thing are possible. What makes a proof elegant? In some ways, the same things that make an algorithm or program elegant. Seeing something that is a neat trick, works, is clear to read, even though you may have to concentrate to follow how it works, is exciting and inspiring. I could spend hours re-reading some of the proofs.

References

[1] The Wendy House, http://www.thewendyhouse.org/

[2] Bloodstock, http://www.bloodstock.uk.com/

[3] Cube, http://en.wikipedia.org/wiki/Cube_(film)

[4] Pi, http://en.wikipedia.org/wiki/Pi_(film)

[5] Rudy Rucker, http://www.cs.sjsu.edu/faculty/rucker/works.htm

Next issue: Allison Lloyd

For those of you who were looking forward to Michael Feathers' Desert Island Books, despite approaching me at the conference and exchanging a couple of emails, Michael is currently eluding me. I shall keep trying to track him down and hopefully he'll be featured in a later edition.

What's it all about?

Desert Island Disks is one of Radio 4's most popular and enduring programmes. The format is simple: each week a guest is invited to choose the eight records they would take with them to a desert island (http://www.bbc.co.uk/radio4/factual/desertislanddiscs.shtml).

The format of 'Desert Island Books' is *slightly* different from the Radio 4 show. You choose about five books, one of which must be a novel, and up to two albums. Some people even throw in the odd film. Quite a few ACCUers have chosen their Desert Island Books to date and there are plenty more to go.

The rules aren't too strict but the programming books must have made a big impact on your programming life or be ones that you would take to a desert island. The inclusion of a novel and a couple of albums helps us to learn a little more about you. The ACCU has some amazing personalities and Desert Island Books has proved we only scratch the surface most of the time.

Each issue of CVu will have someone different. If you would like to share your Desert Island Books please email me: paul.grenyer@gmail.com.

Code Critique Competition 60

Set and collated by Roger Orr.

Please note that participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

Listing 1

```
#include <stdio.h>
#include <iostream>
#include <stack>
using namespace std;
stack<char> cStack;
void decToHex(int num){
  int showNum = num;
  int storeNum;
  while(num != 0){
    storeNum = num % 16;
    switch(storeNum){
      case 10:
        cStack.push('A');
        break;
      case 11:
        cStack.push('B');
        break;
      case 12:
        cStack.push('C');
        break;
      case 13:
        cStack.push('D');
        break;
      case 14:
        cStack.push('E');
        break;
      case 15:
        cStack.push('F');
        break;
      default:
        cStack.push( (char)storeNum );
        break;
    }
    num = num / 16;
  }
  cout << showNum << " in hexadecimal is ";
  while(!cStack.empty()){
    cout << cStack.top();
    cStack.pop();
  }
}
int main( int argc, char ** argv )
{
  // test it
  int integer;
  cin >> integer;
  decToHex( integer );
}
```

ROGER ORR

Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

Last issue's code

Can someone please help me to understand why the following program crashes? I tried to fix it by using `strncat` but it still doesn't work.

Last issue's code is shown in Listing 1.

Critiques

Damien Ruscoe <Damien.Ruscoe@imgtec.com>

Allow me to introduce my solution to this problem by first telling you a little about myself. After many, many years of being a hobbyist developer I have recently started my first software development role. The company have strong belief in the quality of software that is produced and follow development using the TDD methodology. This was my first insight into the automated testing domain of which I myself am becoming a firm advocate.

Upon analysing the dec to hex problem I found myself slipping back into old habits and allowing my eyes to traverse the execution thread for a sample piece of data I plucked from thin air. Upon realising my irresistible haste to get stuck in I decided to do the thing I love to do and wrote some code to execute the algorithm automatically.

I set about creating a test harness. It was difficult. The algorithm firstly converts the decimal value into a sequence of hex characters and then outputs this data to the standard output leaving the option for creating a test to compare an expected result to the actual result. This is not what the function name suggests it does, surely a more appropriate name would be PrintHex or PrintHexFromDec?

After decomposing the algorithm to reflect the function name, I wrote a suite of tests to execute the algorithm. Within seconds of execution I was presented with a list of errors contained with the algorithm. Amongst others one error was instantly caught my attention:

```
Assertion Failure: expected:  0 but got:
```

Ahh, an all too common error prevalent in many applications; a corner case that your application may heavily depend, in which case the error is usually identified very quickly, or more often than not your application uses less frequently, is overlooked and is left as a ticking time bomb waiting to detonate in production code.

Using a while loop for the conversion does not allow the execution thread to enter the loop if the initial data is zero. An alteration to a do-while loop removes this bug and the test remains to ensure the same error is not duplicated during future modifications.

```
Assertion Failure: expected: 1 but got: ?
```

What is that cheeky chappy doing there when 1 was expected? It is time I roll up my sleeves and get deep into manually reading the code to understand where this little fellow was born. That's right, at this point I have only briefly read the algorithm enough to understand some of its semantic properties and how I can interact with it as opposed to trying to understand how it is doing it. I have already established a list of errors and corrected one of them. That is quite an impressive feat don't you agree?

The tests also highlight that most, if not all, numeric character representations are not being converted correctly. I want to find the area of the code that handles this behaviour. The numeric conversion, located in the default branch of the switch statement, is simply not converting a number to ASCII correctly a simple modification and I run my tests yet again.

```
cStack.push ((char) (storeNum + (int) '0' ))
```

All my tests are now passing and hence I have definite proof that the algorithm is behaving as expected: the code works! This obviously is a good

place to be and I can begin the critique the code as I can manipulate the structure while being reassured by the tests, that no bugs have been created.

The algorithm appears to iterate over increasing powers of the hex base, 16, and creating a character representation and pushing this onto a (global?) stack. The large switch statement is looking subtlety redundant as the majority of branches have similar behaviour although is acting with different data; namely the characters A though F. This has a fowl smell [cf Martin Fowler] of code duplication. The numeric data conversion alternatively has a more algorithmic implementation of identifying the character to push onto the stack. Like the numeric conversion the character conversion is a linear isomorphic map onto the ASCII char set and so can be reduced to:

```
cStack.push ( storeNum -10 +(int) 'A' )
```

The switch now only contains 2 branches which, if memory serves, is an if-else construct.

Time to scratch this itch, this stack: What is its purpose and why is it declared in global namespace? The name **cStack** is not the most helpful prompt at discovering its purpose; this is unfortunately distributed over the algorithm described implicitly by the way in which the stack is being used.

Its purpose is to reverse the order of the hex representation characters as the representation is created from least to the most significant digit. I agree that the characters should be create in this order as the converted digits are then disposed from the conversion decimal and using the stack is a wonderful exploitation of one of our most faithful data structures, although, wouldn't it be just a little simpler in this case to prepend the character to the hex representation string?

The author of this code has obviously paid attention during his computer science days although this may be a case of an over-engineered solution. Replacing the stack for a string I was able to delete the global stack definition. I do hope no other system was relying on that definition but you can never be too sure with global data. The author may have been sick on that day of school.

Almost done, I have working code containing no code duplication. The remaining work is to be concentrated on making the algorithm easier to read aiding other developers to critique this code. I have replaced the magic number 16 with a clearer **HEX_BASE** definition and similarly **DEC_BASE** for the magic number 10 I myself am guilty of introducing. BTW, that annoying redundant variable named **showNum** was deleted with such ease and joy that I was oblivious of documenting that activity.

Oh 'eck! I've completely overlooked negative numbers. Again, if I include another set of test data I will allow the test suite to indicate if the algorithm is behaving in a predictable way. What test do I write, or rephrasing, what result do I expect from Hex(-10). I have to think about this before any coding is done. For the moment I will assume that -A is expected

```
Assertion Failure: expected: -1 but got: /
Assertion Failure: expected: -A but got: /0
```

Something is misbehaving. It took me a moment to realise but I eventually discovered I was overlooking a fundamental mathematical property. That is the modulus function has the property $-n < \text{mod}(n) < n$ and not $0 < \text{mod}(n) < n$. This was subsequently affecting the mapping to hex character. An exploit of the strong C type system I overloaded the function to accept both signed and unsigned int types explicitly and separately. As I already have proof that my hex conversion works for positive numbers the simplest solution is to test for negativity within the signed overload and dispatch correctly to the unsigned overload. My tests now execute successfully.

I have established an extensive set of test data to exercise over the algorithm which, to my belief, covers all corner cases of the potential pitfalls. The final code is listed below which contains language constructs that may not be to everybody's personal taste. Although the code is an implementation detail of the function name and coding style comes second to both working code (successful tests) and simplification by removing code duplication.

```
#include <iostream>
#include <string>
using namespace std;
const int DEC_BASE = 10;
const int HEX_BASE = 16;
const string Hex(unsigned int num)
{
  unsigned int temp_digit;
  string result;
  do {
    temp_digit = num % HEX_BASE;
    temp_digit += temp_digit < DEC_BASE ?
      (int) '0' :            // 0-9 char
      (int) 'A' -DEC_BASE;   // A-F char
    result = (char)temp_digit + result;
  } while (num /= HEX_BASE);
  return result;
}
const string Hex(int num)
{
  if (num < 0)
    return "-" + Hex (abs(num));
  return Hex ((unsigned int)num);
}
void PrintHex (int num)
{
  cout << Hex(num);
}
void assertEqual (const string& expected,
                  const std::string& input)
{
  cout << ".";
  if (input != expected)
    cout << "Assertion Failure:"
         << " expected: " << expected
         << " but got: "  << input << endl;
}
struct hex_test
{
  int input;
  string expected;
  void assert ()
  {
    assertEqual ( expected, Hex(input) );
  }
}
hex_test_data [] = {
 {0,"0"}, {16,"10"}, {-0,"0"}, {-16,"-10"},
 {1,"1"}, {17,"11"}, {-1,"-1"}, {-17,"-11"},
 //...
 {15,"F"}, {31,"1F"}, {-15,"-F"}, {-31,"-1F"},
 { 32, "20" },  { 253, "FD" },
 { 41, "29" },  { 254, "FE" },
 { 42, "2A" },  { 255, "FF" },
 { 47, "2F" },  { 256, "100" },
                { 257, "101" },
 { 159, "9F" }, { 258, "102" },
 { 160, "A0" }, { 265, "109" },
 { 169, "A9" }, { 266, "10A" },
 { 170, "AA" }, { 267, "10B" },
 { 175, "AF" },
};
int main ( int&, char** )
{
  int test_count = sizeof(hex_test_data)
                 / sizeof(hex_test_data[0]);
  for (int i=0; i<tests_count; ++i)
    hex_test_data[i].assert();
}
```

If you find that any part of the algorithm is not clear enough for yourself to read you are more than welcome to make any modifications you feel it

needs. The compiler and test suite will check for any syntax or semantic bugs introduced during any amendments.

Martin Moene < m.j.moene@eld.physics.LeidenUniv.nl>

Initially, I thought I'd talk to Roger to learn more about what the decimal to hexadecimal converter should be able to do and – no less important – what not[1]. Maybe that's not a usual practice in this competition, maybe it's just 'not done'. Anyway, I didn't.

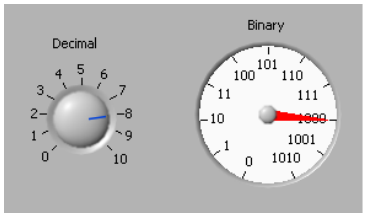
One of the things that isn't immediately clear is if the conversion should handle negative numbers and if so, in what way. What does show through the implementation though, is an interactive decimal to hexadecimal converter that does not provide additional formatting and does not handle negative numbers. Actually, it only handles a subset of N_1.

On to the code at hand. To begin with, the decimal to hexadecimal conversion fails for values where the result of the modulo 16 division (value % 16) is in the range 0..9. In the switch's default case, the value of the numbers 0..9 are used as-is, whereas they should be converted to the characters '0'..'9'. The single hexadecimal-digit conversion that enumerates values 11 to 15 can easily be replaced with a compact algorithmic approach and subsequently the whole single-figure conversion can be extracted[2] to a separate function such as:

```
inline char to_char( const int x )
{
   return static_cast<char>(
      x <= 9 ? '0' + x : 'A'-10 + x );
}
```

The function's name does not hint at the hexadecimal notation, and that's not without reason. When I use the hexadecimal notation, it often is when programming hardware registers whose bits represent operations such as select ADC number x, or clear FIFO. It can be helpful to see the value of those bits separately, that is in binary notation. It may well be that we can present the numbers in hexadecimal (base 16) as well as binary (base 2) notation with very little extra effort.

On my Palm Zire 72 I use Ding Zhaojie's Megatops BinCalc[3] . It has a nice equal opportunity[4] user interface for the binary input-output.



Presented an example how easy visual programming is, I can't help to ask: Ah, can I also grab and move the needle and read the corresponding decimal value on the knob? But I digress...

What then are the things that raise our eyebrows or that we can't resist to comment on? There are quite a few:

- omit the C header file `stdio.h`, this is a C++ program that uses iostreams
- there's also no need for `stdio.h` at all in the current program
- variable **`cStack`** can be declared locally in function **`decToHex()`**
- make the conversion reusable by separating conversion and output
- this also makes it easier to test[5]
- **`showNum`** can be declared const
- variable **`storeNum`** can be defined at its first use in the **`while`** loop
- literal 16 should be replaced by a constant such as **`base`** and defined as **`const int base = 16`**
- prefer to **`declare decToHex()`**'s argument **`const int num`** and define **`int todo = num`** and use it as **`todo /= base`** in the computation
- a C++-style cast should be used in the default case, not a C-style cast
- the **`/=`** operator can be used instead of **`=`** and **`/`** separately
- the conversion function accepts signed numbers but it does not handle negative numbers properly
- it would be useful for example to show -1 as FFFF (the number of Fs depend on **`sizeof(int)`**)
- it took me some time to realise that **`decToHex(0)`** is a special case of error: it generates no output at all[6]
- this can be corrected by replacing the **`while`** loop with a **`do-while`** loop
- the program lacks error handling

Given what we know now, I suggest using the following adapted main function:

```
#include <string>       // std::string
#include <iostream>     // std::cout, cerr
int to_int( std::string text )
{
  return atoi( text.c_str() );
}
int main( int argc, char* argv[] )
{
  const int default_base = 16;
  try
  {
    const int base =
       (argc > 1) ? to_int( argv[1] )
                  : default_base;
    report( std::cout, "cout",
     read( std::cin , "cin" ), base );
  }
  catch ( std::exception const& e )
  {
    std::cerr << e.what() << std::endl;
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Note that the program accepts the base to present the number in as the program's first argument. It also contains the try-catch skeleton to handle errors. Implementation of various other ideas to make the program 'more wonderful' are left to the reader:

- accept a list of bases to present the number in
- accept input also in binary, octal and hexadecimal notation
- provide help via a commandline option such as -h or -help

The supporting input-output functions include checks on the streams and they report bad luck by throwing an exception with an informative message.

```
const std::string progname = "decToBase";
int read( std::istream& is, std::string name )
{
   int integer = 0;
   is >> integer;
   if ( !is )
   {
      throw std::runtime_error(
         progname +
         ": cannot read number from stream '"
         + name + "'" );
   }
   return integer;
}
void report( std::ostream& os,
             std::string name,
             const int x, const int base )
{
   os << x <<" in base '" << base << "' "
      "is '" << to_string( x, base ) << "'"
      << std::endl;
   if ( !os )
   {
```

```
      throw std::runtime_error(
         progname +
         ": cannot not write to stream '" +
         name + "'" );
   }
}
```

The new prototype of the conversion function is:

```
std::string to_string( const int x,
                       const int base );
```

What I like about the converter's original implementation is its use of **std::stack**, well actually I like its use of a container from the C++ standard library. In the following implementation, I've continued this approach.

```
std::string to_string( const int x,
                       const int base = 10 )
{
   REQUIRE( base >= 2);
   REQUIRE( base <= 36 );
   unsigned int ux( x );
   unsigned int ubase( base );
   std::string result;
   do
   {
      result.insert(
         0, // position
         1, // count
         to_char( ux % ubase )
      );
   }
   while( ( ux /= ubase ) > 0 );
   return result;
}
```

Indeed, **std::string** is a container, be it a rather awkward one. Kevlin Henney wrote several interesting articles on the container aspect of **std::string**[7]. He also argues that using high-level (generic) programming constructs are part of a modern approach to teaching C++[8].

A simple yet effective signed-to-unsigned transformation enables us to present negative numbers. Represented unsigned, an originally negative number can be treated as its corresponding bit pattern. To prevent signed-unsigned mismatch in the computations, these also use base as an unsigned number[9]. To make the signed-unsigned transformation more explicit than it is now, a **static_cast<unsigned int>()** can be used.

Using the requested number-base in two computations is all that's needed to present the converted number in base 2 with a highest figure of '1' up to and including base 36 with a highest figure of 'Z'. Zero will convert properly now, with the while loop replaced by a do-while loop with the extracted **to_char()** function. The loop still collects the figures from least to most significant and puts them one by one at the front of the string as they're computed.

In the original while loop the test and the advance operations are rather far apart. Contrast this with the do-while loop that combines them into a single expression. I would have used a for loop hadn't the loop required to be executed at least once. However I slightly prefer the do-while over the following for loop that contains zero as a special-case.

```
std::string result( 0 == ux ? "0" : "" );
for( ; ux > 0; ux /= base )
{
   result.insert(
      0, // position
      1, // count
      to_char( ux % base )
   );
}
```

And here are the results:

```
prompt>decToBase.exe
123
123 in base '16' is '7B'
prompt>decToBase.exe 2
123
123 in base '2' is '1111011'
prompt>decToBase.exe
-1
-1 in base '16' is 'FFFFFFFF'
```

Initially, before I revisited **std::string** and found a usable insert method, I worked out another solution that I like to share.

```
#include <iterator>     // std::inserter
#include <vector>       // std::vector<>
#include <string>       // std::string
template< typename range, typename output >
output rcopy(range const& source, output sink)
{
  return std::copy(
    source.rbegin(), source.rend(), sink );
}
std::string to_string( const int x,
                       const int base = 10 )
{
  REQUIRE( base >= 2 );
  REQUIRE( base <= 36 );
  unsigned int ux( x );
  unsigned int ubase( base );
  std::vector<char> reversed;
  do
  {
    reversed.push_back(to_char( ux % ubase ));
  }
  while( ( ux /= ubase ) > 0 );
  std::string result;
  rcopy(reversed, std::back_inserter(result));
  return result;
}
```

The **std::stack** container is replaced with the more general **std::vector**. As a result, the **std::copy** algorithm can be used to insert the collected figures in a **std::string** in reverse order, by employing reverse iterators on the vector named **reversed**. The idea to condense the copy operation as shown is taken from *Software As Read*, by Jon Jagger[10].

Other solutions spring to mind, for example using a string stream.

```
#include <iomanip>      // std::hex
#include <sstream>      // std::stringstream
std::string to_hex( const int x )
{
   std::stringstream oss;
   oss << std::hex << x;
   return oss.str();
}
```

Yet another uses the C **function _ltoa()**.

```
std::string to_string( const int x,
                       const int base )
{
   REQUIRE( base >= 2 );
   REQUIRE( base <= 36 );
   char buf[ 8 * sizeof( x ) + 1 ];
   return _ltoa( x, buf, base );
}
```

What are your expectations? Recall the exception handling in **main()** and imagine you enter a base on the commandline that is outside the valid range of 2-36 and request a conversion. The conversion function receives a base that it cannot handle in a constructive way. This is a situation that should not happen and thus it is a programming error[11].

Here is macro **REQUIRE**'s job: throw an exception if the stated requirements, or preconditions[12] **base >= 2** and **base <= 36** are not true, as it occurs in[13]:

```
prompt>decToBase.exe 1
123
decToBase.cpp(60): expected 'base >= 2'
```

Note that a user entering an invalid base in fact represents an environmental error, that is, an error that is not unexpected to happen. It is the program that should prevent the invalid base to reach the conversion function that is known to be unable to handle it[14]. Contrast this with a programming error that leads to the same situation: that is not expected to happen.

Macro **REQUIRE** is defined as follows.

```
#include <stdexcept> // std::runtime_exception
#define REQUIRE( expr ) \
  if ( !(expr) ) \
  { \
    throw std::runtime_error( \
      to_string(__FILE__) + \
      "(" + to_string(__LINE__) + \
      "): expected '" + #expr + "'" ); \
  }
```

Macro **REQUIRE** uses two conversion shims[15], one of which is the already familiar number-to-string converter. The story recurses.

```
std::string to_string( std::string text )
{
  return text;
}
std::string to_string( const int x
                    /*, const int base = 10 */)
{
  // guess what
}
```

The choice of the parameter type of the first **to_string()** may come as a surprise. However, with **std::string** as its parameter type, the function can also be used with arguments that are convertible to that type, such as **__FILE__**'s type **char***.

What's left of the original program? One line: there where one comes to provide his or her input:

```
std::cin >> integer;
```

Especially in this context I like the word integer.

Notes and references

[1] Don Wells. 'The Customer is Always Available', http://www.extremeprogramming.org/rules/customer.html

[2] Martin Fowler. *Refactoring: Improving the Design of Existing Code*, http://martinfowler.com/books.html#refactoring

[3] Ding Zhaojie. Megatops BinCalc 1.0.3 (for PC and Palm), http://bincalc.googlepages.com/

[4] Leogo: An Equal Opportunity User Interface for Programming, section 2.1 *Equal Opportunity Interfaces*, http://www.cosc.canterbury.ac.nz/andrew.cockburn/papers/jour_leo.pdf

[5] Kevlin Henney. Put to the test, http://www.curbralan.com/papers/PutToTheTest.pdf

[6] Indeed, I should have created a unit test right at the start.

[7a] Kevlin Henney. Stringing Things Along, http://www.curbralan.com/papers/StringingThingsAlong.pdf

[7b] Kevlin Henney. Highly Strung, http://www.curbralan.com/papers/HighlyStrung.pdf

[7c] Kevlin Henney. The Next Best String, http://www.curbralan.com/papers/TheNextBestString.pdf

[8] Kevlin Henney. The miseducation of C++, http://www.two-sdg.demon.co.uk/curbralan/papers/TheMiseducationOfC++.pdf

[9] Prefer to reserve unsigned types for types that represent bit patterns or flags; to reduce occurrences of signed-unsigned mismatch, prefer signed types for numbers even if these numbers are non-negative only. See *Google C++ Style Guide*, Integer Types, http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Integer_Types

[10] Jon Jagger. Software As Read, http://www.jaggersoft.com/pubs/SoftwareAsRead.htm

[11] Eiffel Software. Building bug-free O-O software: An Introduction to Design by Contract™, http://www.eiffel.com/developers/design_by_contract_in_detail.html

[12] A precondition violation is an unexpected error for the implementor, but not for the user. A postcondition violation is an unexpected error for the user if the precondition was valid on entry, but not for the implementor.

[13] I used literals in the precondition test, so they show up as 2 and 36 in the error message, not as **BASE_MIN** and **BASE_MAX**.

[14] To illustrate the precondition violation, I omitted **base = checked_base(...)** from **main()**.

[15] M. D. Wilson. 'Shims – A Definition', http://www.synesis.com.au/resources/articles/cpp/shims.pdf

Joe Wood <joew@aleph.org.uk>

Preliminaries

Before we get too involved in the code, it is worth spending a short while looking at the problem. The main function, **decToHex**, finds the hexadecimal string representation of a number. Just to start us off, lets drop talk of hexadecimal and stick to plain decimal which we all grew up with. For this discussion we must distinguish between the number 513 and the string "513" which is a representation of the number. We will put string representations in quotes for clarity.

The basic approach used in the code is to take an initial number, *N*, and an empty stack (string "") and to find the least significant part of *N* by taking the modulus in the required base. For example 513 modulo 10 yields 3 and so "3" is pushed onto the empty stack giving "3". It should be stressed that "3" is just a symbol to represent the number 3, it could be "c" (say). However, we must have unique symbols to represent each of the elements in base. Next we divide 513 by 10 yielding 51. Since 51 is not 0, we process the 51 as above. After a couple more iterations as above we get *N* equals 0 and the stack/string "315", which is clearly back to front, but a simple string reversal fixes that problem. Table 1 clarifies the calculations.

Note that "513" is a short hand for 5 x 10^2 + 1 x 10^1 + 3 x 10^0 = 513

N	String	N%10	Symbol (N%10)	New String	N / 10
513	""	3	"3"	"3"	51
51	"3"	1	"1"	"31"	5
5	"31"	5	"5"	"315"	0
0	"315"				

The above discussion has been somewhat laboured, but it is useful to do so for two reasons. Firstly, this representation of numbers highlights the brilliant evolution of Indian and Arabian mathematics, try multiplying in Roman Numerals. Secondly, exactly the same algorithm works for hexadecimals (base 16). In this case we have Table 2, for example.

Note that "201" base 16 is a short hand for 2 x 16^2 + 0 x 16^1 + 1 x 10^0 = 513 decimal.

N	String	N%16	Symbol (N%16)	New String	N / 16
513	""	1	"1"	"1"	32
32	"1"	0	"0"	"10"	2
2	"10"	2	"2"	"102"	0
0	"102"				

So the function **decToHex** is really a special case of **decToBase**.

Code issues

There are a number of issues with the current code, as follows (in no particular order)

- The original line (default part of switch), **cStack.push((char)storeNum);**, does not do as intended. It should push the

string representation of **storeNum** onto the stack. In fact it pushes the numeric value of **storeNum** onto the stack.

- **cStack** is never initialised and indeed is shared by multiple users of **decToHex**; making **decToHex** neither re-entrant nor suitable for a multi-threaded environment. Both problems are solved by declaring **cStack** inside **decToHex**.
- **decToHex** uses a separate case clause for each number over 10, this makes the code longer and more complex and hence harder to test and maintain.
- **decToHex** directly prints out the result, making it less generally useful then returning the string representation.
- **decToHex** produces no output for an input of zero.
- **decToHex** only processes non-negative numbers and so we should limit the input domain accordingly.

Discussion

There are two basic options for converting a (small) number to its corresponding string representation, either via a look-up table or via calculation.

In the first case, we initialise the table with the required symbols and just index into the table (in pseudo-code)

```
const string symbol ( "0123456789abcdef" );
    …
const char letter = symbol ( storeNum );
```

This is very general, as we can change the character representation in symbol as required. But we must check that the initial string is correct! It might be a tad slower than the second method.

In the second case, we observe that in ASCII and its supersets the characters "0" .. "9" and the characters "a" .. "z" ("f") are both contiguous subsets with one single discontinuity, so we can just calculate the require letter (in pseudo-code)

```
const char letter = ( storeNum < 10
    ? '0' + storeNum : 'a' + storeNum - 10 );
```

This is probably quicker than the first method, but is less general and must not be used on an EBCDIC system for large bases.

Both are valid and both work. Which you prefer is largely a matter of personal taste and trade-offs between speed, maintainability, readability and intended throughput. You pays your money and makes your choice.

The original implementation uses a stack to store the individual character representations, which are then read back to form the correct result, in effect we just reverse the stored strings. Given that string has **push_back (append)** and supports **reverse**, we will use string in place of stack.

A possible solution

Below is one possible solution. This is done using namespace **std**, purely for space reasons.

```
static const string symbols("0123456789abcdef");
string decToBase ( int base, unsigned int num )
{
  // Initialise an empty string as the result
  string result ( "" );
  if ( base < 1 || base > 16 ) {
    throw domain_error( "Illegal base value" );
  }
  // Loop over the num and find the remainder when
  // we divide by 16 and convert the remainder to
  // base 16. Then we divide the current value of
  // num by 16. Stop when we get to zero.
  // Note: putting the test at the end
  // ensures that we handle an input of 0 correctly
  do {
    // Find the modulus to base 16
    const char storeNum = char ( num % base ) ;
    // Append character representation to result
    // Note: We know that the modulus must be
    //  between 0 and 15
    result += symbols [ storeNum ] ;
    // Calculate next part
    num = num / base;
  }
  while ( num != 0 ) ;
  // BUT a smart person will observe that result
  // is back to front, so reverse the result
  reverse ( result.begin(), result.end() );
  return result;
}
// Given the input number, num,
// return its hexadecimal string representation
string decToHex ( int num )
{
  return decToBase ( 16, num );
}
// Add some infrastructure for testing
typedef map<unsigned int, string> results_type;
static results_type mk_results ( void ) {
  results_type results ;
  results[0]   = "0";    results[1]   = "1";
  results[2]   = "2";    results[3]   = "3";
  results[4]   = "4";    results[5]   = "5";
  results[6]   = "6";    results[7]   = "7";
  results[8]   = "8";    results[9]   = "9";
  results[10]  = "a";    results[11]  = "b";
  results[12]  = "c";    results[13]  = "d";
  results[14]  = "e";    results[15]  = "f";
  results[16]  = "10";   results[511] = "1ff";
  results[512] = "200";  results[513] = "201";
  results[522] = "20a";
  results[INT_MAX]  = "7fffffff";
  results[UINT_MAX] = "ffffffff";
  return results;
}
static void basic_tests ( void ) {
  const results_type tests (mk_results());
  results_type::const_iterator it;
  bool passed = true ;
  for ( it = tests.begin() ; it != tests.end();
        it++ ) {
    const int test = (*it).first ;
    const string expected = (*it).second;
    const string result = decToHex ( test );
    if ( result != expected ) {
      passed = false;
      cout << "Warning: Test failed, decToHex("
           << test << ") yielded " << result
           << "   expected " << expected
           << endl;
    }
  }
  if ( passed ) {
    cout << "Good basic tests passed, over to"
            " you" << endl;
  } else {
    cout << "Warning: At least one basic test"
            " failed" << endl;
  }
}
// Main routine
int main ( int argc, char* argv[] )
{
  basic_tests();
  // test it
  while (true) {
    int dec_int;
    cout << "Please enter a decimal number ";
    cin >> dec_int;
    if (!cin.good()) {
      throw runtime_error(
```

```
        "Error detected on input stream" );
    }
    if (dec_int < 0) {
      break;
    }
    cout << dec_int << " in hexadecimal is "
         << decToHex ( dec_int ) << endl;
  }
  return 0;
}
```

Nevin Liber <nevin@eviloverlord.com>

The basic algorithm is, for each 4-bit nybble from least significant to most significant, append the corresponding printable character to a string. When done, reverse the string.

As written, there are three bugs in the program:

- Bug #1: 4-bit nybbles are being converted from their internal value (what the author calls 'decimal', but that is really a misnomer) to a corresponding printable hexadecimal character using the system encoding (ASCII, EBCDIC, etc.)

  For the nybbles 0 through 9, the encoding is not 0 through 0 but rather '0' through '9'. '0' converted to an ASCII encoding would have a value of 48, converted to EBCDIC would be 240, etc.

  While I know that both ASCII & EBCDIC are contiguous with respect to the encodings of '0' through '9', I don't actually need to make that assumption in this program. At this point, I'll just add 10 more case statements (and show a more elegant solution later; first get it working, then improve the style), and eliminate the default case statement (since it is not necessary).

- Bug #2: It doesn't work for 0, as the entire conversion is skipped.

  Whenever possible, I prefer code which doesn't have a bunch of special cases, as I find it easier to reason about the purpose and correctness of that code. For this problem, I noticed that it will correctly process 0 if it goes through the loop at least one time (and for all other numbers it will go through the loop at least one time anyway), and I can get this behavior by changing the `while { /* ... */ };` loop to a `do { /* ... */ } while ( /* ... */);` loop.

- Bug #3: It doesn't work for negative numbers. In C++, division rounds towards 0, which surprisingly may involve rounding up (example: -1 / 16 == 0) and the modulus operator involving any negative numbers has a sign that is implementation-defined.

  To quote the 2003 C++ Standard section 5.6 Multiplicative Operators:

  "If the second operand of / or % is zero the behavior is undefined; otherwise (a/b)*b + a%b is equal to a. If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined."

  This is not the behaviour we want. The fix for this is to internally use unsigned types.

Putting it all together:

```
// Added cases 0..9
// Changed while (...) {} to do {} while (...)
// (so num == 0 works) Changed int to unsigned (so
// negative numbers work)
void decToHex(int showNum){
  unsigned num = showNum;
  unsigned storeNum;
  do {
    storeNum = num % 16;
    switch(storeNum){
      case 0:
        cStack.push('0');
        break;
      case 1:
        cStack.push('1');
        break;
      // ...
      case 14:
        cStack.push('E');
        break;
      case 15:
        cStack.push('F');
        break;
    }
    num = num / 16;
  } while (num != 0);
  cout << showNum << " in hexadecimal is ";
  while(!cStack.empty()){
    cout << cStack.top();
    cStack.pop();
  }
}
```

Now that it is working, we can make improvements.

Side note: if all we cared about is making it work, we could just use the conversion routine that comes with the stream library, as in:

```
void decToHex(int showNum)
{ cout << dec << showNum << " in hexadecimal is "
   << hex << uppercase << showNum; }
```

and call it a day. However, the algorithm as implemented is certainly worth looking at and improving.

Improvement #1: `cStack`. Besides being a global, it is a very slow and inefficient way to reverse a string. Internally it uses a `deque`, which means memory allocations are going on behind the scenes. We can do better.

Instead of using a stack to reverse the string, let us just build a (C-style) string from right to left and output it (which works left to right) when we are done.

How big should the array holding the C-style string be? We need one output character for every 4 bits (nybble) in the source number. This comes out to ceiling of the number of bits in an unsigned int divided by 4; in other words, rounding the result up to the next integer.

In integer math, the way to calculate that is by adding 3 (the denominator - 1) to the numerator before performing the division by 4. We also need space for the null-terminator. The resulting calculation (consume 4 bits of input for each character output, plus the null-terminator):

```
(sizeof(unsigned) * CHAR_BIT + (4 - 1)) / 4 + 1
```

Note: this implementation is independent of the number of bits in `CHAR_BIT` (which is found in the header `<climits>`).

Improvement #2: Replacing the switch statement. The switch statement is converting a number from 0 to 15 to a corresponding character encoding. We can do that directly by using an array. In addition, a string literal is a succinct way of initializing that array. Note: while it is possible to use the string literal directly, I'll need it in an array variable for the last thing I want to do to this algorithm.

The resulting implementation is:

```
void decToHex(int showNum)
{
  static const char encoding[] =
    "0123456789ABCDEF";
  char cString[(sizeof(unsigned) * CHAR_BIT +
                3) / 4 + 1];
  // just past the end of cString
  char* converted(cString + sizeof(cString));
  // null terminate the string
  *--converted = '\0';
  *--converted = '\0';
// Build the cString in reverse order
  unsigned num(showNum);
  do { *--converted = encoding[num % base]; }
  while (num /= base);
```

```
  cout << showNum << " in base 10 (signed) is "
       << converted << " in base " << base
       << " (unsigned)";
}
```

Commentary

The original code did have a number of problems – both in terms of actual bugs but also in terms of poor design. I think between them the entrants covered the code in great detail.

As is common with the format of this code critique column, a precise definition of the problem being solved was missing. So Martin's opening remark ('Initially, I thought I'd talk to Roger to learn more …') was a very good place to start.

The four entrants made different decisions about what the program ought to do with negative numbers – who knows which was correct? Perhaps this might have been a chance to discuss how to help the writer of the code learn to anticipate this sort of question earlier.

My view is that this is a bad function to write yourself – unless there are very strong reasons I'd suggest using the streaming operators as the resultant code is then 'obviously correct'. Additionally, the decision about what to do with negative numbers then becomes simply a choice of using int or unsigned int.

The Winner of CC 59

It was very hard to pick a winner this time round; all four entrants did a good job of providing a critique of the code. I liked Damien's treatment using a test driven style and Joe's explanation with pictures of what was going on. Nevin's solution demonstrated a more optimised solution which is portable across different architectures.

On balance I've decided to award the prize to Martin, for a combination of the main solution and the other alternative ways that he presented to solve the problem.

As usual, thank you to all those who entered the competition.

Code Critique 60

(Submissions to scc@accu.org by Dec 1st)

I'm trying to write a simple high precision integer class, using an array of chars as the representation. I've started, but have problems with my tests – the second output is wrong. I also want to be able to stream a bignum as a string, but I can't – any ideas?

The code is in Listing 2. You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 2

```
-- bignum.h --
#include <string>
#include <vector>

class bignum
{
public:
  bignum( int i = 0 );

  operator std::string();

  bignum & operator +=( bignum const & rhs );
private:
  std::vector<char> value_;
};

-- bignum.cpp --
#include "bignum.h"

#include <algorithm>
#include <iostream>
#include <sstream>

bignum::bignum( int i )
{
  value_.push_back( i & 0xff );
  while ( i & 0xffffff00 != 0 )
  {
    i >>= 8;
    value_.push_back( i & 0xff );
  }
}

bignum::operator std::string()
{
  std::ostringstream oss;
  for (int idx = 0; idx < value_.size();
       idx++)
  {
    int v = value_[value_.size() - idx - 1];
    oss << std::hex << v;
  }
  return oss.str();
}

bignum & bignum::operator +=(
   bignum const & rhs )
{
  value_.resize(std::max( value_.size(),
                rhs.value_.size()));
  char carry = 0;
  int idx = 0;
  for ( ; idx < std::min(value_.size(),
                rhs.value_.size()); idx++ )
  {
     value_[idx] += rhs.value_[idx] + carry;
     carry = value_[idx] < rhs.value_[idx];
  }
  for ( ; idx < value_.size(); ++idx)
  {
     value_[idx] += carry;
     carry = value_[idx] < carry;
  }
  if ( carry )
    value_.push_back(1);
  return *this;
}

-- test.cpp --
#include "bignum.h"
using namespace std;
int main()
{
  bignum b(0x1234567);
  b += 1;
  // won't compile: cout << b << endl;
  cout << b.operator std::string() << endl;
  b += 0x234;
  cout << b.operator std::string() << endl;
}
```

Listing 2 (cont'd)

Bookcase

The latest roundup of book reviews.

If you want to review a book, your first port of call should be the members section of the ACCU website, which contains a list of all of the books currently available. If there is something that you want to review, but can't find on there, just ask. It is possible that we can get hold of it.

After you've made your choice, email me and if the book checks out on my database, you can have it. I will instruct you from there. Remember though, if the book review is such a stinker as to be awarded the most un-glamourous 'not recommended' rating, you are entitled to another book completely free.

I must thank Blackwells and Computer Bookshop for their continued support in providing us with books.

Jez Higgins (jez@jezuk.co.uk)

FORTRAN 77: A Structured, Disciplined Style Based on 1977 American National Standard FORTRAN and Compatible with WATFOR, WATFIV, WATFIV-S, and M77 FORTRAN Compilers

By Gordon B. Davis and Thomas R. Hoffmann, published by McGraw-Hill, 2nd edition, copyright 1983, ISBN 007Y662622

Reviewed by Colin Paul Gloster

As I have become involved with porting legacy FORTRAN 77 code, the appearance of the words 'Structured' and 'Disciplined' in the title caused me to hope that this would be a relatively good book, considering the inherent drawback of the language involved. Unfortunately my optimism was too... optimistic. I was not so naive at the beginning as to hope that the co-authors would overcome the deficiencies of FORTRAN 77 (for example, the maximum identifier length is still six characters long in FORTRAN 77), but I hoped that it would be more blatant in its reservations concerning error-prone parts of the language and actually encourage fairly good practice for its time. Though early parts of the book contain acceptable advice, better programming techniques should have been introduced much earlier (so as to be mentally absorbed by repetition throughout the book), examples include the late introduction of functions on Page 237; and symbolic names were used for constants throughout the book, but these symbolic names were for FORTRAN variables instead of FORTRAN constants and it was only in the last chapter that it was shown how to use the language to enforce constancy with the **PARAMETER** keyword. Much worse, promotion of error-prone techniques (examples include not bothering with explicitly declaring the datatypes used; and the **COMMON** keyword for placing variables in the same memory location) and dangerously incomplete descriptions of major parts of the language prevent any recommendation to use this book, despite some good tips. The main dangerously incomplete description is the lack of a warning that FORTRAN compilers accept mismatches between formal parameters and actual parameters, which in the year 2008 is still a common problem with FORTRAN.

In 2007, looking at the Usenet newsgroup news:comp.lang.fortran was more productive in discovering major extant flaws in Fortran than reading this book. However, newer Fortran standards have added more problems than those restricted to FORTRAN 77. I learnt from this book of the bizarre addition in FORTRAN 77 of a rule making whitespace in an identifier irrelevant, but that is mild in relation to newer examples of how the language becomes worse.

Otherwise better languages rely on libraries or work-arounds for some of the in-built mathematical capabilities of FORTRAN which boasts non-integer exponents and a **COMPLEX** data type. Of the four books which I have read from cover to coverwith substantial FORTRAN in them (two books on numerical methods and two books on FORTRAN programming), this one has by far the most coverage and usage of the **COMPLEX** data type. How much? Barely more than a paragraph. However, an in-built **COMPLEX** data type mattered to one FORTRAN programmer I once met at a conference.

The first apparent syntactic influence of FORTRAN 77 (as opposed to earlier FORTRAN versions) on another language seems to be the adoption of the colon for an array range in **MATLAB**. Though using a colon in a range was copied from FORTRAN 77, it was in turn copied from ALGOL, which may have copied it from somewhere else for all I know. (Did you know that ALGOL 68 was not the last version of ALGOL? A newer version was published after FORTRAN 77. Some people do not know when to give up.)

This edition of the book is dated 1983, and the first edition was dated 1978 and was also supposedly on FORTRAN 77. However, the aforementioned issue with the **PARAMETER** keyword is a piece of evidence that FORTRAN 77 was not the primary educational theme throughout the 1983 edition so I doubt that the 1978 edition differed in this regard. Too much was written with a FORTRAN 66 mindset, with some FORTRAN 77 enhancement tacked on before it went to print.

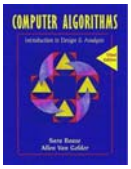
In the preface, the style of programming taught in the book was misleadingly described as 'structured' and leading to 'well-designed' programs. Such immodest boasts have been made for many things throughout the history of computers, though you may need to substitute the word 'structured' for another word in other decades. Instances of code replication encouraging unmaintainable messes when scaled up abound in this book such as on pages 74; 79; 101; 107; 256; and 358, for example Listing 1 appears on page 107 which reminded me of a more recent book with a giant case statement every branch of which contained nothing except identically copied and pasted instructions. That was a book by the overrated Grady Booch on software 'engineering' with 'object-oriented techniques' to address the 'software crisis'..

Listing 1

```
IF((HRSWRK .LT. 0.0) .OR. (HRSWRK .GT. 60.0)) THEN
     PRINT *, 'ERROR IN INPUT DATA'
     GOTO 101
   ELSIF((WGRATE .LT. 0.0) .OR. (WGRATE .GT. 10.0)) THEN
     PRINT *, 'ERROR IN INPUT DATA'
     GOTO 101
   ELSIF((DEDUC .LT. 0.0) .OR. (DEDUC .GT. 35.0)) THEN
     PRINT *, 'ERROR IN INPUT DATA'
     GOTO 101
ENDIF
```

If you read something in C Vu that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?

Computer Algorithms: Introduction to Design & Analysis

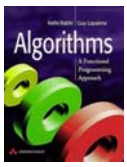


By Sara Baase and Allen Van Gelder, published by Addison Wesley, 3rd edition, 2000, 675 pages

Introduction to Algorithms

By Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, published by The MIT Press, (C) 1990, 18th printing, 1997, 990 pages

Algorithms: A Functional Programming Approach



By Fethi Rabhi and Guy Lapalme, published by Addison-Wesley, 2nd edition, 1999, 232 pages

Reviewed by Colin Paul Gloster.

Computer Algorithms: Introduction to Design & Analysis and *Introduction to Algorithms* are normal textbooks on general-purpose algorithms for a wide variety of problems. *Algorithms: A Functional Programming Approach* is also a general-purpose book, but the choice of Haskell and the small page count (arising from fewer topics and shallow coverage instead of any supposed advantage of Haskell) make it obviously different at a first glance. However, a surprising difference which became apparent after I started reading it is that Rabhi and Lapalme concentrated on space complexity instead of time complexity. Bearing that major drawback in mind, it is a fairly good book, though it is yet another unconvincing attempt at functional advocacy, and it is overpriced.

As for the other two books, they are big as books on algorithms should be. More algorithms are covered (or at least mentioned) in *Introduction to Algorithms*, but there are more details in *Computer Algorithms: Introduction to Design & Analysis* for the algorithms covered therein. (Fewer problems are covered in *Algorithms in C++: Parts 1-4: Fundamentals; Data Structures; Sorting; and Searching* by Robert Sedgewick and Christopher J. Van Wyk, but at even greater depth.)

Cormen et al. used pseudocode with various mathematical symbols such as arrows which are not on normal keyboards (unless old APL keyboards are more common than I thought), whereas Baase et al. used a pseudo-language based on what used to be Java. Their preconditions and postconditions in Javadoc comments indicate that Eiffel would have been better but Java's marketing was not mentioned as one of the factors contributing to the choice to use Java. A lot more recursion and fewer overwriting assignments were used than in other imperative programs, but not to an extent qualifying as functional programming. Sometimes they used zero-based indexing, sometimes they used one-based indexing: Ada would have been better than Java.

Baase et al. suggested 'Accelerated Heapsort may become the sorting method of choice' but I do not believe that this has happened. They waited until Page 650 to warn 'With currently available compilers, a program written in Java runs more slowly than a program written in C'. They recommended converting exceptions into errors because handling exceptions would be necessary but handling errors are merely optional.

The books under review are not particularly biased towards a specific application domain, but *Introduction to Algorithms* has a few examples of how algorithms are useful for electronic engineering and it also has a chapter on arithmetic hardware.

The books under review mainly deal with sequential algorithms for tractable problems. The chapter by Baase and Van Gelder on parallel algorithms conveyed the point that the relative merits of algorithms changes if they are to be run on uniprocessors or multiprocessors better than the other books. Instead of plainly revealing important points in the bodies of the chapters, Cormen et al. left too much to exercises. Their chapter on dynamic programming is better than the one by Baase and Van Gelder.

These books are fairly old so smoothed analysis is not in them.

Only a very small selection of numerical methods is presented in these books, but *Introduction to Algorithms* and *Computer Algorithms: Introduction to Design & Analysis* are much closer to the state of the art on multiplying dense matrices than every book dedicated to numerics which I have read.

Bookshops

The following bookshops actively support ACCU (offering a post free service to UK members – if you ever have a problem with this, please let me know – I can only act on problems that you tell me about). We hope that you will give preference to them. If a bookshop in your area is willing to display ACCU publicity material or otherwise support ACCU, please let us know so they can be added to the list

- **Holborn Books Ltd**
  (020 7831 0022)
  www.holbornbooks.co.uk
- **Blackwell's Bookshop**, Oxford
  (01865 792792)
  blackwells.extra@blackwell.co.uk

View From The Chair

Jez Higgins
chair@accu.org

Loud, energetic, chaotic? If you have kids, do they match that description? No? Lucky you. Mine do. Since they were big enough to walk in a reasonably competent way, my kids, currently 9 and 6, have been keen on sport. Since we live in the middle of a thumping great city, there are a wealth of sporting opportunities available more or less on our doorstep, which we take fair advantage of. As a parent, there are two main ways to engage in your kids' sports clubs. One common choice is acting as little more than a taxi service – deliver child, take child home again, fill the gap inbetween reading the paper/Cosmo/email/paperback. Slightly less popular is actually engaging – simply knowing the coach's name is a start, watching the match/training session/etc, learning a bit more about the sport (my kids don't do football), going to see a high-level game or two (and generally quite cheap), learning when the team/squad/etc have done well or badly, and so on. I'd recommend engaging, because it's bags more fun.

Last weekend, I was gently manhandled into considering becoming Chair of the junior section of my kids' hockey club. I wasn't surprised, although I did fleetingly think 'no good deed ..'. The various advantages of my taking the position were laid out. I'm not an insider because I'm not a member of the senior club. I have a long term interest because if they both carry on playing hockey I could spent two hours on the astroturf every Sunday for the next ten or eleven years, and so on. On the plus side, the meetings aren't particularly frequent and I wouldn't have to take up the position until April next year. So I'm considering.

Next April coincides with my stepping down from my position here as ACCU Chair. Happily we won't need to 'suggest' people into the role, as a candidate has expressed an interest in standing. If you are interested, or even merely inclined towards, getting involved in running the organisation, you don't have to wait for the AGM to roll around. The mentored developer programmes, for instance, are driven by people's interest, as are the local groups. Even the conference, on which Giovanni and the conference committee to expend a great deal of time and energy, would amount to nothing were it not for those who propose sessions and those who attend them. That may seem a little obvious, but it is worth repeating. Engage, it's more fun.

Erratum

The article 'Interpreting Custom Unix Shell Scripts in C' by Ian Bruntlett in the September issue of CVu should have included a program listing, which unfortunately got left out during editing. Here it is, including some minor changes suggested by the readership of ACCU-General.

```
// hashbang.c

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

int main (int argc, char *argv[])
{
  char buffer[100];

  // handle parameters
  printf( "Hello World from %s with %d arguments\n", argv[0], argc );
  printf( "argv[0] interpreter name\n"
    "argv[1] shell script name\n"
    "argv[2] etc - shell script arguments\n"
    );

    int i=0;
    for ( i=0; i<argc; ++i )
  {
    printf ( "Interpreter : %d arg is %s\n", i, argv[i] );
  }

  // read interpreter commands
  if ( argc < 2 )
  {
    printf( "Interpreter : Insufficient parameters to interpreter\n" );
    return EXIT_FAILURE;
  }

  FILE *fpScript = fopen( argv[1], "r" );
  if ( fpScript == NULL )
  {
    printf( "Interpreter : unable to open script file %s\n", argv[1] );
    return EXIT_FAILURE;
  }

  while ( fgets(buffer, 100, fpScript) != NULL )
  {
    printf( "SCRIPT LINE : %s", buffer );
  }

  return EXIT_SUCCESS;
}
```