Building Applications out of Several Programs

N.M. Maclaren
Computing Service
nmm1@cam.ac.uk

July 2009

1.0 Introduction

This course is a basic introduction to the principles of building applications out of multiple programs, which includes when to split an application up into multiple programs. It does **not** cover the actual programming, though it gives a few examples. It also covers the simplest cases in some detail, and just mentions the complex ones and other tricky issues. It covers mainly the principles of why, when and how – feedback on what aspects are most useful would be appreciated.

While this was written from a Unix viewpoint, almost all of it applies to Microsoft Windows, with only minor changes. The reason for that is that virtually all aspects of the latter have been modelled on or derived from Unix facilities, and the few that have not (e.g. the 'kernel') have been modelled on other systems with a very similar set of concepts to Unix.

These notes do not reproduce the pictures in the foils, and there is a lot of structure in the foils that is not reproduced here, so you should look at the foils in parallel when reading this. The documents are in the same order.

1.1 Overview of Course

The course starts with an elementary overview, then describes chains (also known as pipes, streams, FIFOs and queues) and their use and mentions some other basic structures that are useful and easy to get right. It then gives and introduction to how to design and code interfaces (a very important factor in actually getting the application to work on Real Data!). If time permits, and there are enough people interested, it will describe some problems with monolithic codes and mention some advanced topics, but inexperienced users should omit those.

You should go to Python courses if you don't already know it, as it is by far the best tool for this sort of application building.

2.0 Using Separate Programs

Before starting, it is worth noting the four golden rules, which you should keep in mind throughout:

- KISS (Keep It Simple and Stupid). This is perhaps the first rule of how to write reliable code, and is regrettably often forgotten by people who think they are experts.
- Only divide up your application in 'natural' ways. The more that you have to distort your application's basic design to fit its implementation, the harder it will be to program and debug, and the less reliable and tunable it will be.
- Use a simple, debuggable structure. It is not obvious to non-mathematicians, but the structure of a multi-component design is one of the most important factors in getting it working. This will be clarified by giving examples.
- Interfaces are **AS** important as components. This cannot be overstressed. In complicated systems (such as operating systems, GUIs, networking etc.), most hard problems in the interfaces between components and not within single components. "GUI" means "Graphical User Interface", such as most people expect to use on a desktop or laptop nowadays.

The first question is obviously "Why do this?", and I will give the most common reasons. but there are many, many others.

2.1 Using Existing Programs

You may have an existing program or programs, and want a different sort of interface; for example, you may want it to be simpler, clearer, better targetted for your area, more flexible or GUI-based for a line-mode program. Or you may want to automate some analyses, to simplify your work.

To do this, you may need to combine several programs; some may be available only as binary executables, they may be in different languages, and so on. Keeping them separate and using a higher level wrapper avoids mixed-language executable problems, which can be very nasty indeed.

All of this is just industrial-strength scripting, which you should regard as programming using processes as components, rather than procedures. That is really all that it is!

2.1 Splitting Up Programs

But splitting up complicated applications can be useful, too. It can often increase debuggability, both by simplifying each component and by providing interfaces to locate bugs and problems, and you can often debug components separately. This is nothing more than the principles of structured programming applied at a slightly higher level. And you can use it to avoid library incompatibilities, where you want to use two libraries that can't be linked into the same program.

It is sometimes critical for efficiency, and this is becoming increasingly important. It can enable your application to run in parallel on multi-core systems, and so complete faster, and you can even use some components on remote systems if you have access to several that you want to use in combination. And, lastly, GUI code will 'poison' high-performance (HPC) codes (whether SMP, as in OpenMP, or not, as in MPI); some details of this is given later.

2.2 Choice of Languages

The components (i.e. programs) can be written in anything – a binary-only executable definitely counts as 'anything', as the Unix and Microsoft process interfaces are language-independent. And, yes, each program can be different, so this is a good way to mix languages as disparate as C++ and Fortran 90.

Controlling programs should be in a language suitable for the purpose, which really means Python, Perl and similar ones. It can be done in C++, Fortran 90 etc., but is considerably more effort – however, it can be worthwhile when building complicated applications that are mostly in those languages. Complex shell scripting is for masochists only, as anyone who has tried it can witness.

Python is the recommended tool for simple use, for reasons that will be shown later.

There are some heavyweight solutions for writing advanced controllers, including Iris Explorer from NAG (perhaps the leader), Data Explorer (ex-IBM, now open source) and many others used in various areas of commercial use. They are mostly GUI-based, fairly hard to learn, but there are a few users in the University, and they are worthwhile only for very heavyweight tasks, unless you already know them.

3.0 Basic Structures

Some structures are easy to use and debug, and one can even prove them mathematically correct; while the latter may not appear important, it is actually a major factor in determining whether you will ever finish debugging the code! 90% of applications can use one of these 'safe' structural designs, and 99% can use a clean combination of them.

I will mention places where problems occur, mainly to say "don't go there", but please remember, one golden rule is to use a suitable structure for your application.

3.1 Basic Simplex Chains

These are also known as pipes, streams, FIFOs (first-in, first-out), queues and occasionally sockets – they are all the same mathematical concept. Simplex ones are ones that pass data from one end (the input) to the other (the output), in the serial order of its input, with nothing going backwards; duplex ones are much

trickier. One of the great advantages of simplex chains is that you can use large buffers and as many CPUs or systems as there are stages. Streaming I/O can be optimally efficient, if done well, and so I/O need not be too much of a bottleneck.

The control and data flow are simply linear from input to the output, and this is done automatically by shell pipelines. They are very simple, very reliable and easy to test. I will return to using chains for interactive work later; because you (as the user) can now feed back data from the output to the input, they are more complicated.

3.2 Controlling Chains

The shell creates a pipe (pipes have two ends, remember!), starts program A and feeds its output into one end of the pipe, and only then starts program B taking its input from the other end of the pipe. Unfortunately, no current shell (not even `bash`) handles errors in pipes correctly, and Bourne, Korn and C shells are even worse.

Controlling programs should do the same as bash; no other synchronisation is needed. Using default I/O in components is almost always adequate, though using larger buffers may be desirable for efficiency.

3.3 Python Chain Controller

To show how easy this is to do in Python, consider the three-stage pipeline `'huey | dewey | louie'`. This is merely:

```
from sys import stdin, stdout
from subprocess import Popen, PIPE
p1 = Popen(["huey"],stdout=PIPE)
p2 = Popen(["dewey"],stdin=p1.stdout, \
    stdout=PIPE)
p3 = Popen(["louie"],stdin=p2.stdout)
rc = p3.wait()
```

The library function `Popen` creates and attaches pipes, runs the program and creates an object(e.g. `p1`) to use as a handle on the process. The keyword `PIPE` creates a new pipe and attaches one end, and the other can be attached by referring back through the process handle (e.g. `stdin=p1.stdout`). However, the above code doesn't include any error handling.

The necessary error handling is to wait for or kill **all** subprocesses (i.e. not just the first or last), and to print the subprocess name and the error code for any that failed. It is good practice to trap the Python exception `OSError`, and certain failures will raise that. There is an example in the Python library manual, which is 8 not-very-complex lines for a very similar case. To apply it to the above, just replace the `rc = call(...)` by `p1.wait()`, `p2.wait()` and `p3.wait()`, and replace `cmd` by the command name in the following example:

```
try:
    rc = call(cmd+args, shell=True)
    if rc < 0:
        print >>sys.stderr, cmd+" sig", -rc
    else:
        print >>sys.stderr, cmd+" exit", rc
except OSError, e:
    print >>sys.stderr, cmd+" fail:", e
```

Obviously, the hardness needs to check all of the return codes and not just the last. The best programs check that all input has been sent (i.e. the next stage has read all of its input), and that they get EOF when they have read all of the output from the previous stage. Quite a lot of bugs in poorly written programs show up as early normal termination.

3.4 Other Chain Controllers

Perl users should see "Programming Perl", chapter 16. Some **very** simple cases are easy, but in general it is not much easier than C. There is very little error handling by default, so it is necessary to add a lot more checking than for Python. And, like C, your Perl program should clean up its environment before starting the component, as Perl encourages programmers to make changes to its environment – but this is getting onto an advanced topic.

C, C++ and POSIX controllers are too complicated for this course, and you should avoid this if you possibly can; an example is given later. At the very least, you need the `pipe`, `dup2`, `fork`, one of the `exec` and `waitpid` system calls, plus cleaning up any of programming environment you have reset (including signal handling and open file descriptors). Not doing this can cause confusion or even chaos. Some example code is shown in the extra foils. And that is just for the simple case, where your controller does nothing fancy!

Fortran controllers call C to do the actual process control, but the advanced logic can be in Fortran. This is not worthwhile for simple chain control, but starts being so for master/worker designs. Please ask for help if you want to do this.

4.0 Components

Writing these is trivial; all you need to do is to use standard input and output in the default fashion, and It All Just Works. You can use any programs that can work as filters, like `awk` or `sed`, as well. The following examples show how to code `'cat'`, though they aren't always complete programs.

In Python:

```
from sys import stdin, stdout
while 1 :
    line = stdin.readline()
    if not line :
        break
    stdout.write(line)
```

In Perl:

```
while (<STDIN>) {
    print $_;
}
```

But you will need to add some error handling, as Perl includes very little automatically.

In Fortran:

```
character, len=big_enough :: buffer
do
    read (*,'(a)',end=10) buffer}
    write (*,'(a)') buffer}
enddo
10 continue
```

In C++:

```
string s;
while (cin >> s) cout << s << std::endl;
```

In C:

```
char buffer[big_enough];
while (fgets(buffer,sizeof(buffer)-1,stdin) {
    buffer[sizeof(buffer)-1] = '0';
    fputs(buffer,stdin);
}
```

4.1 Golden Rules of I/O

The basic rule is to use streaming I/O, allow the system (usually the language run-time system) to handle blocking or reblocking, and don't reposition or handshake in any way; all of this is the default in all of the above languages. If you have large amounts of data, use binary (also known as unformatted) I/O, and use large buffers (64 KB or more) if at all possible; these are not the default.

You should check but distrust **all** error codes, even when an operation 'cannot possibly fail', which includes doing an explicit close and check its return code. Unfortunately, some common failures will not show up even if you do this, but you can do nothing about them in even a semi-portable program.

5.0 Other Controller Methods

Returning to controller design, there are other ways of building chains. One is that the first program in the chain spawns the second and waits for it to finish, and so on; the spawning ensures that the programs start in the right order, and the waiting that they finish in the right order. Alternatively, where there is no data passed via pipes, the controlling program can run the first program, wait for it to finish, and then run the second. The ordering can also be forced by sending messages down a pipe, or by using signals (but that is not recommended).

Some such logic is needed if using files to pass data between components, because it is **essential** to close an ouput file before it is opened for input; the chaos that can be caused by allowing early opening is almost indescribable. A couple of Python examples show these approaches.

Program A:

```
output = open(filename,''w'')
output.write(some\_data)
output.close()
p1 = Popen(''B'']) 
p1.wait()
```

Program B:

```
input = open(filename,''r'')
. . .
```

This shows how the first component writes to a file, closes it, and only then starts the second component that reads it.

Program A:

```
output.write(some\_data)
output.close()
stdout.write(filename)
```

Program B:

```
name = stdin.readline()
input = open(name)
. . .
```

This shows how the first component writes to a file, closes it, and then passes its name down a pipe to the second program, which opens it. The message is simply the filename in this case, though it would usually be more complex.

6.0 GUIs – Both X and MS Windows

The most common reason to split programs is because they need a GUI interface; mixing GUI code with almost anything else is a positive nightmare. Part of the reason is that most GUIs are required to be programmed as event loops with no long delays between servicing events. They also do horrible things with networking, often demand specific compiler options, and name clashes and other problems abound.

They are also foul to debug – most bugs are not easy to repeat, and there is no way of sending evidence of most problems to someone else. In extreme cases, a failure may even lock up console and force a reboot,

so there is **NO** way to investigate further. Yes, the X Window System also has the equivalent of the Blue Screen of Death.

The solution is to separate the GUI code from complex analysis and to Keep It Simple. This turns writing GUI code from a nightmare into merely a headache.

6.1 GUI Input and Output

One very common and recommended design is the following:

- One GUI component creates and checks the input in a "user friendly" way and stores its result in a file or files.
- The analysis program then runs non-interactively, reading those input files and creating output files.
- Another GUI component reads those output files, and displays and selects results in a "user friendly" form.

Many commercial and production programs do this, and it is almost universal in HPC (High Performance Computing) environments. Four decades of experience supports this design, and experience is that it can save a **LOT** of debugging time! So why is this?

You can rerun any stage if it fails, which is **very** useful for debugging, error reporting, or if you just need to go to bed! The files provide proof of where the errors lie, and can be mailed as evidence to other people, or to allow someone else to reun the later stages. You can automate (i.e. script) the creation of input and the analysis of the output if you don't want to use the GUI, automate systematic changes to the input, and so on.

In the HPC arena, an analysis may take days, need restarting or may need to be run on another system, and these requirements are not rare even in 'ordinary' computing. The above design enables these.

7.0 Serial Master/Worker

Another very simple structure, which has been hinted at above, and is the one used in the above GUI design, is serial master/worker. The master component runs the worker components serially, possibly interleaved with its own work, and never starts a new task until the previous one has been tidied up. This is simple, reliable, but not parallelisable, and is **precisely** the programming with processes rather than functions that was referred to earlier. You really can approach it in exactly the same way, and it will work.

For example, the master might spawn and wait for component A, do some computation itself, spawn and wait for component B, spawn and wait for component C, and so on. It is hard to describe this in more detail, as there really is nothing more to it – but you must close data files before reopening them in another process.

8.0 More Complex Structures

Once we get onto more complex structures, we need more advanced tools. The key concept is a transaction, which I will describe next; this is effectively an atomic message and its reply. Streaming I/O **can** be used with care, but you must remember that pipes have a finite capacity and they will block if too much is written to them and not yet read. Using files to transfer large quantities of data is much safer.

I will give guidelines for safe use only; experts can and do break the rules, and most production programs are different.

8.1 Simple Transactions

The simplest form of a transaction is that program A writes all of its request, program B reads all of that request, program B writes all of its reply, and program A reads all of that reply; the transaction is now complete.

The critical aspect is that there is **NO** other communication between programs A and B, directly or indirectly, during that process. For reasons mentioned in the extra foils, it is essential not to start the reply while reading request, and essential to read the reply to one request before sending the next request.

Provided that you do that, using transactions is very simple. If you break those rules, you are in expert-only territory.

8.2 Tree Structures

A serial master/worker component can itself be a serial master/worker application, and this gives a tree structure, in just the same way that you can call functions within functions. Doing that adds very few problems.

The standard rules of not being clever when sharing descriptors and watching out for environment pollution get more important when you do this, but you should be following them anyway. And don't expect recursion (i.e. a program being called when it was already called higher up the tree) to work – it can be done, but it is asking for trouble!

The only real problem is handling failures, because killing a process doesn't kill children, so it is very easy to create disconnected trees. This is a soluble problem, but needs advanced (and non-portable) techniques. Systems like MVS and VMS did this automatically, but Unix and its imitators don't.

8.3 Parallel Master/Worker

Parallel master/worker is very common in the HPC and cluster arenas; the master runs the workers in parallel, and the workers talk only to their master. This is a very good way of running multiple programs in parallel on SMP systems (multi-core and multi-socket) and clusters, and is the simplest way of making use of their parallelism.

In the simple cases, the master reads input, divides the work up into suitable units, and then spawns all of the workers, in turn. If it has written the data to files, it just needs to pass each worker the file to use as input; if not, it needs to pass the data as part of spawning them. It then waits for all of the workers, and collects their ouput, in turn, in the reverse of the way that it handles their input. It then combines it and writes the aggregate output.

Master/worker communication can be a problem, and is easiest if the master supplies the input initially and then just collects results at end, as described above; and, as before, it is usually safest when files are used for this. But sometimes ongoing communication is needed; a solution is described later under duplex pipes in the extra foils. But avoid it if you **possibly** can, because it can be very hard to debug, and use simple transactions (i.e. not streams) if you must do it.

8.4 Simple Client/Server

You will have heard of client/server applications; in their simplest form, they are quite easy to program. The server runs as a daemon (i.e. indefinitely), waiting for requests; as it gets them, it responds to each request in turn, and that is a transaction. Clients gather their input, send requests, wait for the reply, and produce output; they may do this repeatedly.

`msntp` is a **very** simple example of the above form, and you are welcome to a copy to look at; `exim` is a more realistic one. I don't recommend writing one until you have some experience with multi-component applications, though it is actually not hard to do (in the above very simple case).

8.5 Combination Structures

There is usually an optimum amount to divide applications up by. The smaller the components are, the easier each one is to debug, and the more interactions there are to worry about. And conversely. So don't go overboard in either direction.

You can combine the above structures in many ways; components of chains and worker components can be themselves master/worker structures, chains or even clients of a client/server one. A `ssh` command is just a client of a client/server design, after all, and using `ssh` in multi-component applications is very useful. But remember to Keep It Simple.

Beyond that, don't go there – **really** don't go there; more complicated designs are virtually impossible to debug. Some widespread distributed applications do this, and they include most schedulers, desktops, much Grid software and so on. Administrators curse them, vigorously, because they so often don't work in almost undebuggable ways.

9.0 Data Interfaces

You should design all inter-component interfaces like **external** interfaces, and your left hand should not trust your right hand. The reason is simply that all humans make errors, and most people make them frequently. Having checks at interfaces is a good way of simplifying debugging, not least by identifying in which component the problem started.

All programs should check their input for validity, even when they are only reading data that another of your programs has written. Checking the results you are writing to a file immediately before doing so can be worthwhile, too. You should be thorough, but there is no need to be paranoid – usually quite simple checks are enough. This approach really will save you time, overall, even though it takes longer to code and debug initially.

9.1 Specific Checks

You should check the format of the input – a bad format may mean that a line has wrapped or been truncated. You should check the validity of data and be wary of assuming that the language run-time system will do it – for example, "NaN" is not a floating-point number but will be read as such by many languages. Your should check that values are in a plausible range; if they aren't, your program is likely to go wrong anyway. You should check the consistency between data – e.g. if the number of elements in an array is supposed to be N, check that it really is N.

Failures can mean that the program that wrote the output crashed, and any check may pick up data corruption, even though no check is guaranteed to do so. And you should add any other checks that you can think of that are fairly cheap to program, because you never know what effect 'finger trouble' can cause.

9.2 Designing Formats

Data formats (including ones that are passed in binary/unformatted mode) should be designed, and you should not just dump internal data structures. This needn't take much time, as the first rule is KISS, again. You should include some cross-checks, so that it is easy for the next program to check for gross corruption, and usually should include some values just for checking. These needn't be complicated, but writing both data and associated counts, maxima, sums, or whatever makes sense can help to pick up problems.

For example, don't write just N values, but a count and then the N values, or the N values and a terminator, or both. Writing the numbers 1.2 2.3 3.4 4.5 5.6 would be better done as 5 1.2 2.3 3.4 4.5 5.6 (including the count), 1.2 2.3 3.4 4.5 5.6 -1.0e30 (including a terminator) or 5 1.2 2.3 3.4 4.5 5.6 -1.0e30 (both).

And you should most definitely document the format you are using, so that the next person to work on your code (possibly you, a year later) can see what it is supposed to do. I favour block comments in the code, because it is easy and is kept together with the code. For example:

```
# All main items and rows start on a new line
# Extra spaces and newlines in numbers ignored
#
# Title and author in free text
# Date in format 01/Apr/2006
# Row and column sizes
# Data by row appended with -1.0e30
```

9.3 Structured Data

If your data have a non-trivial structure, it is worth writing it out in a pseudo-formal notation.

```
Object = < Vector | Matrix >
Vector = Size Newline Values(Size) Newline
Matrix = Row_size Column_size Newline Rows(Column_size)
Row = Values(Row_size) Newline
Value = < 'Missing' | Floating-point >
```

This says that an object is a vector or a matrix. A vector starts with its size on a line by itself, followed by that number of values, ended by a newline. A matrix starts with its row and column sizes on a line by themselves, followed by each row, each of which is ended by a newline. And a value can be either the word Missing or a floating-point number.

The advantage of doing this is that you can spot problems in the format you use, your program can decode it and, with care, it can detect and flag errors. For example, the above makes it easy to spot that there is no ambiguity between a vector and a matrix with one row, as the first line of the former with one integer and the latter with two. And your program can use that test to distinguish the cases.

Congratulations - you're now using BNF (Backus-Naur Form) and can call yourself a computer scientist!

If you need to do this sort of thing, read up about BNF. I advise doing so in Wikipedia, as it makes less of a meal of it than most textbooks do! Despite its name, it is **NOT** complex, and it is very useful. Don't worry about the notation – anything goes – you want it mainly to keep your **own** thoughts clear and to ensure that your code can parse it, and not to feed into some ghastly compiler generator. So, if someone tells you that you are using it wrongly, thank them kindly for their advice and ignore them.

`http://en.wikipedia.org/wiki/Backus-Naur_form`

10.0 Advanced Topics

The foils contain more information (including on some of the forward references made above), but they may not be easy to follow. If you have trouble in these areas, you are recommended to seek advice, as advanced issues in this area can get very nasty, very fast. If there is sufficient interest, this course may be extended to go into them in more detail.

The general recommendation is to Keep It Simple and to follow the advice given above, at least until you become moderately expert and you will probably never come across the harder problems.