EMBOSS: Guide to Writing EMBOSS Programs

Programming EMBOSS Applications

ableasby@embnet.org

1: INTRODUCTION

This booklet is an introduction to programming EMBOSS and assumes the reader knows, in essence, what the package is and does. The package is at http://www.uk.embnet.org/Software/EMBOSS and will have been downloaded and installed from there. This guide begins with a general introduction to the internal structure of the package. It then moves onto the command line as a prelude to writing ACD files and finally gets to grips with programming in C per se.

2. PACKAGE STRUCTURE

There are 5 main levels in the EMBOSS hierarchy:

- **AJAX:** This directory contains all low-level library functions. For example, sequence reading and writing, file handling, mathematical operations, string handling, memory handling, list processing etc are all here.

- **NUCLEUS:** This directory contains high-level library functions. These are almost exclusively molecular biology algorithms for alignments, pattern matching, restriction, isoelectric point calculation etc.

- **PLPLOT:** This directory contains the graphical output routines. It is an LGPL library developed outside the EMBOSS project. There are higher-level calls to this library in AJAX. This library may be replaced at some stage however, the AJAX interface to it should remain the same.

- **EMBOSS:** This directory contains the applications (programs) released with the package. The source code is a useful repository of examples of how to do things although, of course, some examples are better than others; we all have our bad habits.

· **DOC:** This directory, or rather its subdirectories, contains the online documentation for the programs as both html and text, the latter being used by the *tfm* program.

Some, but not all, of these levels have sublevels. The AJAX and NUCLEUS libraries do not; this makes it easy for the programmer to find things. As for the others, here is a brief description.

· **emboss/acd:** every EMBOSS application has an associated ACD (Ajax Command Definition) file and this directory is where they live. ACD files are described later. So, for the program *fred.c* the file *fred.acd* must be added to this directory.

· **emboss/data:** many of the EMBOSS applications have associated data files, for example BLOSUM matrices. This is where such files are kept. Putting files in here has the great advantage of the directory being in the search path the AJAX library uses. Data files for some common molecular biology databases are numerous and these appear in separate directories within the data file hierarchy so, there are directories **emboss/data/PROSITE** , **emboss/data/REBASE**, **emboss/data/PRINTS**. The EMBOSS applications *prosextract, rebaseextract* and *printsextract* are used to populate these directories.

· **plplot/lib:** an important run-time directory for the graphics applications. This is where EMBOSS currently looks for any font files. The environment variable PLPLOT_LIB is used to point here.

· **doc/programs/text** and **doc/programs/html:** everybody always writes the documentation before even starting on the program. In an ideal world at least. These directories are where the finely crafted prose is kept. The documentation must adhere to the format used by existing applications otherwise the online manual commands, like *tfm*, won’t work.

There are other directories in EMBOSS but the above are the ones the programmer is most directly concerned with.

3. HOW TO ADD AN APPLICATION TO EMBOSS

This is easy, write the program, add the relevant files to some or all of the above directories, alter the *Makefile*-ish whatzits and you’re done. EMBOSS compilation is controlled by GNU utilities and therefore have GNU-style Makefiles. The golden rule is **don’t touch the Makefile files**. The only "makefile" you need to alter is:

emboss/Makefile.am

This contains a list of the applications supplied in the version of EMBOSS you compiled. If you edit the above file then the GNU utilities will recognise it has changed the next time you type *make* and recreate the **Makefile** file, which is the behaviour you want. Before you edit the **Makefile.am** you'll have put your application **wibble.c** in the emboss subdirectory. Here is a cut-down version of Makefile.am:

```
SUBDIRS = acd data

check_PROGRAMS = ajtest ajtest2 ajtest3 ajtestajnam \
cluster \
demofeatures demolist demosequence demostring demotable \
entrails histogramtest messtest \
patmattest plplottest proteinmotifsearch \
seqinfo \
seqretallfeat seqretfeat seqtofeat \
simplesw testplot treetypedisplay

bin_PROGRAMS = acdc antigenic \
backtranseq banana \
chaos checktrans chips cirdna codcmp complex \
* lines deleted for clarity *
tmap transeq trimseq \
vectorstrip \
water wobble wordcount wordmatch \
wossname

#FLAGS=-lX11

INCLUDES = -I$(top_srcdir)/nucleus -I$(top_srcdir)/ajax -I$(top_srcdir)/plplot

acdc_SOURCES = acdc.c
antigenic_SOURCES = antigenic.c
backtranseq_SOURCES = backtranseq.c
banana_SOURCES = banana.c
chaos_SOURCES = chaos.c
checktrans_SOURCES = checktrans.c
chips_SOURCES = chips.c
* lines deleted for clarity *
trimseq_SOURCES = trimseq.c
vectorstrip_SOURCES = vectorstrip.c
water_SOURCES = water.c
wobble_SOURCES = wobble.c
wordcount_SOURCES = wordcount.c
wordmatch_SOURCES = wordmatch.c
wossname_SOURCES = wossname.c

* lines deleted for clarity *

LDADD = ../nucleus/libnucleus.la ../ajax/libajaxg.la ../ajax/libajax.la ../plplot/libplplot.la $(XLIB)
```

```
if PURIFY
  LINK = purify $(LIBTOOL) --mode=link $(CC) $(CFLAGS) $(LDFLAGS) -o $@
else

endif
```

There are two sections which concern us for development. The **bin_PROGRAMS** line and the **_SOURCES** lines. The former is the place to put the name of the executable which, in this example will be **wibble** and the latter is where the name of the source file **wibble.c** needs to go. It is recommended that you add these in alphabetical order. The edited file will look like the following:

SUBDIRS = acd data

check_PROGRAMS = ajtest ajtest2 ajtest3 ajtestajnam \
cluster \
demofeatures demolist demosequence demostring demotable \
entrails histogramtest messtest \
patmattest plplottest proteinmotifsearch \
seqinfo \
seqretallfeat seqretfeat seqtofeat \
simplesw testplot treetypedisplay

bin_PROGRAMS = acdc antigenic \
backtranseq banana \
chaos checktrans chips cirdna codcmp complex \
*** lines deleted for clarity ***
tmap transeq trimseq \
vectorstrip \
water **wibble** wobble wordcount wordmatch \
wossname

#FLAGS=-lX11

INCLUDES = -I$(top_srcdir)/nucleus -I$(top_srcdir)/ajax -I$(top_srcdir)/plplot

acdc_SOURCES = acdc.c
antigenic_SOURCES = antigenic.c
backtranseq_SOURCES = backtranseq.c
banana_SOURCES = banana.c
chaos_SOURCES = chaos.c
checktrans_SOURCES = checktrans.c
chips_SOURCES = chips.c
*** lines deleted for clarity ***
trimseq_SOURCES = trimseq.c
vectorstrip_SOURCES = vectorstrip.c
water_SOURCES = water.c

```
wibble_SOURCES = wibble.c
wobble_SOURCES = wobble.c
wordcount_SOURCES = wordcount.c
wordmatch_SOURCES = wordmatch.c
wossname_SOURCES = wossname.c

* lines deleted for clarity *

LDADD = ../nucleus/libnucleus.la ../ajax/libajaxg.la ../ajax/libajax.la ../plplot/libplplot.la $(XLIB)

if PURIFY
  LINK = purify $(LIBTOOL) --mode=link $(CC) $(CFLAGS) $(LDFLAGS) -o $@
else

endif
```

There are other interesting things you can do with this file but the above is all you need to know to develop an application. The operation of the file can safely be left as a black box. One *gotcha* is worth a mench though. The **bin_PROGRAMS** section is one logical line, it has continuation (\) characters purely for ease of editing and beautification. Strange and wonderful things can happen if you accidentally delete a continuation character or , if necessary, forget to add one. On the other hand the **_SOURCES** section has discrete lines which should not have continuation characters.

4. THE COMMAND LINE

Before describing programming of both ACD and applications it is necessary to describe the EMBOSS command line so you can see how everything fits together. I'll mention this in little detail as all the grisly details are more the scope of how to use the applications rather than how to program one.

The command line is pretty standard, the main distinction to have clear in your mind is the difference between a **parameter** and a **qualifier**. Here is an example command line:

```
restrict embl:ecompa -nosticky results.restrict
```

The simple difference is that a qualifier is the thing beginning with hyphen, everything else apart from the program name is a parameter. In the above example **embl:ecompa** is a sequence to input and **results.restrict** is the name for an output file. **-nosticky** happens to tell the program that restriction enzymes producing sticky ends aren't wanted.

· **Qualifier:** these always have a qualifier name which is preceded on the command line by a hyphen. Qualifiers can have associated values which appear immediately after them.

· **Parameter:** These have an optional parameter name, if they do they can appear anywhere on the command line. If the optional name isn't given they must appear in the order they are given in the ACD file for the program. The optional names begin, again, with a hyphen.

5. THE ACD FILES

AJAX Command definition files are programmed in the **emboss/acd** directory. They control all the user input operations. All EMBOSS programs have all their user input prompted for before the main part of the application begins. An EMBOSS application cannot ask the user for more information after several hours of processing!

An important thing to remember about any EMBOSS application is **all input is read and held somewhere in memory before the application *per se* starts with a vengeance**.

This document tells you just enough ACD to write most applications. For a full description of ACD please read the **ACD syntax document** available on the EMBOSS web pages.

5.1 The *acdc* application

EMBOSS provides the programmer with an application to test their ACD files as they're being written. This application is called *acdc* and which stands for "ACD compiler". Assuming you are writing the *wibble* application you'll have a file in the emboss/acd directory called *wibble.acd*. This acd file can be tested as follows:

acdc wibble

The rule that all input is read in by EMBOSS applies to *acdc* in that, if you've specified an input sequence, then that sequence must exist otherwise the acdc application will exit with an error. In short, you'll see, using *acdc*, exactly what the user will see if they type the same thing as you at the prompts.

This means that in EMBOSS **you can write the acd before the application itself!** This is good programming practice but usually the acd file will be written concurrently with the C source code file.

5.2 The Application Line

All ACD files must start with an **appl** construction. This construction gives three pieces of information. The first is the name of the application. Then, in the body of the construct, there are **doc** and **groups** lines. The doc line provides a string of text which will be printed to stdout whenever the application is run. The groups line associates the application to programs which do similar things or different things in the same general area. This type of information is used by the *seealso* application. Here is an example:

```
# AJAX COMMAND DEFINITION (ACD) FILE


# ajb 14th April 1999
appl: restrict [
        doc: "Finds restriction enzyme cleavage sites"
        groups: "DNA:sequence features, restriction enzymes"
]
```

Any empty lines, or lines beginning with ?#’, are comments. The application is called "restrict" and the string "Finds restriction enzyme cleavage sites" is printed whenever the application is run. The application operates on the subgroup of DNA sequences and the groups it belongs to are "sequence features" and "restriction enzymes". Applications can belong to more than one group and they are separated by commas.

You can use either **DNA:** or **PROTEIN:** if they are appropriate. If they are not just leave them out and type in the relevant groups. The most recent set of groups can be found on the EMBOSS home page; at the time of writing they are:

Alignment
Alignment global
Alignment local
Alignment multiple
Coding regions
Comparison
CpG islands
Database entry extraction
Database indexing
Database information
Display alignment multiple
DNA sequence composition
DNA sequence features
DNA sequence properties
Enzyme kinetics
Gene finding
Hydropathy
Motifs

Mutation
Pattern matching
Primers
Profiles
Protein sequence composition
Protein sequence features
Protein sequence properties
Reformatting
Repeats
Restriction enzymes
Secondary structure
Sequence comparison
Sequence composition
Sequence display
Sequence editing
Sequence information
Text search
Transcription
Translation
Utilities
Utilities help
Utilities keyword search

If your application doesn't fall into any general group then by all means write in a new group type but don't make a habit of it though.

5.3 General ACD syntax

All ACD definitions have the same general syntax which is:

datatype: name [optional arguments]

The datatypes are built in to emboss and include sequence, integer, float types and many more. The name can be anything you like within reason. This will be the name by which qualifiers (and optionally parameters) are known on the command line. The optional arguments allow you to specify information strings, default values, maxima and minima , and more. Any whitespace is ignored. There is an option to all applications which will print the ACD in the required/desired EMBOSS format though. That application switch is called **-acdpretty**.

The optional arguments you will use mostly are:

· **param:** Specified this is a parameter. It is followed by a Y/N value that says whether or not EMBOSS should prompt for the value if it isn't given on the command line.

On the other hand qualifiers are specified using the following two arguments:

· **req:** Specifies that this is absolutely required by the application. The Y/N value associated with it has the same meaning as for the param definition

· **opt:** Specifies this is an optional (an advanced) qualifier. The Y/N value associated with it has the same meaning as for param and req.

It is, of course, the case that the application should use or test anything from the ACD specification somewhere within it otherwise any unused definitions shouldn't be in the ACD file at all. It is also quite common that a required value isn't prompted for; this usually happens when a default value has been set such as:

· **def: 5.6** Which sets the default value of a floating point datatype to 5.6. What follows the default definition depends on the kind of datatype you've defined. Similarly you can use:

· **max:** To specify a maximum value and

· **min:** To specify a minimum value.

All the EMBOSS dataypes have a default prompt associated with them. For specifying the more fundamental datatypes such as sequences and input/output files you should generally let EMBOSS use its defaults. It is good practice with qualifiers to give them some text which will be used as a prompt. This is usually done with

· **info: "Enter a friggin displacement"**

or somesuch.

We can take the simple integers, floats and booleans, as examples:

· **int: garibaldi [req: Y min: -7 max: 38 def: 10 info: "Enter a garibaldi value"]**

· **float: vir [opt: Y def: 5.6 info: "Set your vir here"]**

· **bool: londo [opt: N def: Y]**

The first two will be prompted for if not given on the command line, the last won't but will have a default true value. When using the **-help** option available to all applications the first one will be shown as a required qualifier and the last two as advanced qualifiers. There are many tricks with this sort of thing and some of the more common ones appear below.

5.4 Specifying sequence files

You will usually specify sequences as parameters and you ?ll want them prompted for if not specified on the command line. Their specification is generally:

· **sequence: delenn [param: Y]**

· **seqout: sheridan [param: Y]**

The first of these specifies that a single input sequence is required, the second that a sequence output file is a prerequisite. The fixed datatypes are "sequence" and "outseq", the names (which in this case will be optional as these have been specified as parameters) are"delenn" and "sheridan". Whilst this is legal naming it isn't entirely clear. We therefore have a convention that the following definitions are used instead:

· **sequence: sequence [param: Y]**

· **seqout: outseq [param: Y]**

After all, a user will be expecting a sequence to be called "sequence".

Often an application will either allow or require more than one sequence to be specified. In this case, instead of the **sequence** datatype you should use the **seqall** datatype. You will frequently see the following definition in EMBOSS ACD files:

· **seqall: sequence [param: Y]**

Sequences also have an optional argument called **type:** which allows you to tell the ACD processing that a DNA or protein must be read. This argument even allows you to specify whether the sequence is allowed to have any ambiguity codes or unknown residues. The common types are:

dna
rna
puredna
purerna
nucleotide
purenucleotide
protein
pureprotein

Others are available for allowing sequences with gaps or any sequence at all. See the ACD syntax document for these.

5.5 Specifying other input and output files

Sequence files are rather special in that, not only does the file need to be opened, one sequence has to be read in before ACD processing finishes, no matter what format that sequence is in or where it is. Sometimes, though, you just need a general input file or a general output file. This can be achieved by the following datatypes:

- **infile: gkar [req: Y]**
- **outfile: lennier [req: Y]**

As usual you can give these default values which will be treated as strings or any **info** prompt you wish.

5.6 Sequences have built in attributes

A common requirement is that some ACD values need to be specified in terms of other values which will not be known until the data (most notably the sequence) is read in. Sequences therefore have attributes. The most commonly used of these are its length, start position and end position. These are accessed using the "**$**" and "**.**" characters. Suppose that you want an integer to be input which is allowed to have a value which does not exceed the sequence length. You would use something like the following to achieve this:

sequence: zocolo [param: Y]

· **integer: marcus [req: y min: 1 max: $(zocolo.length)]**

Other useful attributes for sequences are:

· **$(xxx.begin) -** The sequence begin value

· **($xxx.end) -** The sequence endpoint

· **($xxx.protein)** - A boolean saying whether this sequence is a protein

Again, for less frequently used attributes see the ACD syntax formal specification.

5.7 Strings and Lists

By far the most common "other" datatypes are the strings, lists and select lists. The use of the string datatype is obvious:

· **string: sinclair [opt: N def: "EBLOSUM50"]**

The list datatypes are used whenever the user needs to be presented with a choice. The "select" lists give numerical options from 1->n whereas the "list" lists allow text to be typed as the keys. As you can use **list** to perform both types of operation this is the most frequently used. An example is:

```
list: table [
        opt: Y
        default: "0"
        min: 1
        max: 1
        header: "Genetic codes"
        values:
                "0:Standard";
```

```
                    "1:Vertebrate Mitochondrial";
                    "2:Bacterial"
          delim: ";"
          codedelim: ":"
          info: "Code to use"
]
```

This will produce a list from which one and only one value must be selected (min: 1 **max:**1). The **delim** and **codedelim** fields refer to the list specification in the **values** field. The values you can select are really strings and are "0", "1" or "2". This is an example of how a list can be used to emulate a "select list".

5.8 Graphics

For these the use of the **xygraph** datatype is recommended. An example of its use would be:

```
xygraph: graph [ req: Y multi: 4 ]
```

which says you want something that will allow you to display 4 graphs on the same plot.

5.9 Performing calculations in ACD

Sometimes you'll want, for example, a maximum value set to the sequence length plus or minus a certain value. For calculations the @() syntax is used. Here is an example:

```
int: window [ req: Y max: @($(sequence.length)-30) ]
```

This sets the maximum value to be 30 residues less than the sequence length. Addition, subtraction, multiplication and division are allowed and all operations can be nested.

5.10 Variables

Sometimes a calculation can get a bit messy. For this, and other reasons, the ACD syntax allows you to use variables. A short example will help here:

```
var: mulexp: [ "@($(sequence.length)*2)" ]
```

```
int: bridge [ def: 100 max: @($(mulexp)-30) ]
```

5.11 Conditionals

Our brief tour of "common ACD for the programmer" ends with a look at conditional operations. A very common requirement is to prompt for a text output file only if the user hasn't selected a plotting option. Let's assume the default is to prompt for a text output file and the user needs to type **-plot** on the command line to force a plot. The first part is easy:

```
bool: plot [ opt: N def: N ]
```

Here is one way to control the prompting:

```
outfile: outf [ req: @(!$(plot)) nullok: Y]
```

The negation operator (!) is effectively a calculation so the @ syntax has to be used. What is the **nullok** term for? Well, files always need a name and nothing is usually not a valid name. In this case it has to be. To stop ACD processing complaining the nullok term has to be added.

The ternary operator is often used to, for example, set a default value to differently depending whether the sequence is a protein or DNA one. Here is an example:

```
def: "@($(sequence.protein) ? 30 : 7)"
```

Which will set the default to 30 if the sequence is a protein otherwise it will be set to 7. The ternary operation is another calculation hence the @ syntax again.

6. PROGRAMMING PHILOSOPHY OF EMBOSS

A remark made by Richard Stallman, the author of EMACS, sums up the most important idea.

"Any program using arbitrary limits is fundamentally bugged"

If anyone submitted a program for inclusion in EMBOSS that had such a line as:

```
char sequence[10000];
```

within it then the main development team would need a lie down to recover from the horror. The second notable feature of EMBOSS programming is that it borrows concepts from C++. It is, however, written in the C programming language. The reason for this is mainly because when the EMBOSS project began there was no ANSI C++ standard. The library does look rather like C++. It uses objects and includes constructors, iterators, destructors and even incorporates the idea of data hiding.

Every object in EMBOSS is dynamically allocated. The programmer should not dig into the library objects themselves, that is purely the domain if the library functions themselves. You will mainly be dealing with string objects and sequence objects. There are many other objects in EMBOSS though and we'll meet them later.

Saying that you shouldn't look at objects is like a red rag to a bull as far as C programmers are concerned, myself included. So, lets have a look at the string object both as an example of how EMBOSS manages to emulate C++ and to introduce object nomenclature. The string object is one of the most simple ones and internally its definition is:

```
typedef struct AjSStr
{
        ajint Len;
        ajint Res;
        ajint Use;
        char *Ptr;
} AjOStr, *AjPStr;
```

The **object** is the **AjPStr** definition. As C programmers will see, this is a pointer to a structure. All objects are referred to with pointers. For the AJAX pointers these start with "**AjP**". This nomenclature is maintained for all objects so its easy to guess that a single sequence object is an **AjPSeq.** Back to the AjPStr object though. The **char *Ptr;** is just a standard C pointer which holds a character string and the **ajint Len;** is its length. The character string may or may not be null terminated and this has both gotchas and benefits. For example, the library functions for printing AjPStr objects look at the length field for how many characters tp print; they won't stop at the first NULL if there is one. The **ajint Res;** item internally lets the library know how much reserved dynamic memory is associated with the object. This will obviously be at least equal to Len but often more. *Res* is and should be outside your direct control. If you use a library call to add anything to the string then, if it'll fit within the memory given by *Res* then the operation is performed immediately; if the memory required is larger than *Res* then more memory is allocated and the *Res* item is updated. A little more memory than required is usually allocated. That just leaves the **ajint Use;** item to describe. It is a usage pointer. You will sometimes want to have two objects pointing to exactly the same data. In that case it is pointless duplicating the string. Instead, the usage counter is incremented. When destroying a string object usage counter is first decremented. Only if the usage counter is zero will the object be deleted. Of course noone can prevent you from accessing the internals directly, all we can advise is that **if you intend altering the contents of an object then safety is guaranteed if you use the library functions for the purpose in hand**. If you don't use the library functions and dabble without fully understanding the library then its either segmentation fault or bus error time.

After that look at the internals I'll now introduce an ajax library function that will catenate two string objects. That library function has the prototype:

*AjBool ajStrApp(AjPStr *str, AjPStr str2);*

The funtion will append str2 to str leaving the result in str. The question is, since the objects are pointers anyway why is the first argument a pointer to a pointer? You should be able to work that out given the explanation above. Suffice it to say that if use of a function can possibly mean more memory has to be allocated for an object then a pointer to that object must be passed. The string that is being added to the end of the first one is not going to be altered in any way therefore the object alone can be passed. In short:

If a function can change how much memory is allocated to an object then a pointer to that object must be passed.

IF THERE IS ONE RULE TO REMEMBER FROM READING THIS TEXT THEN PLEASE REMEMBER THAT ONE

The library functions are well documented as to whether an object or pointer to an object is required but they know what they're doing. Hopefully you do too now.

6.1 And the rock cried out "No hiding place!"

We can now move onto the subject of data hiding. A reasonable working definition of data hiding is that the data held within that object are private. Only functions which are "friends" of the object are allowed to see the data. Any functions which are going to alter the data should not alter the original, rather they should get their own copy of the data and alter that. EMBOSS provides these ideas.

However, the concept of a **char *** character pointer is so fundamental to C programmers that the developers of EMBOSS thought it would be a real hassle not to provide access to the raw data. Angry C programmers were not the only reason for providing such access, a major consideration was to make porting of molbiol code out in the big wide world easier. A function you'll see a lot in EMBOSS programs is:

*char *ajStrStr(AjPStr str);*

This returns a pointer to the contents of a character string held within an AjPStr object. It is just so unbelievably useful to be able to get this sometimes. That's not to say that EMBOSS library functions are slow though, just that getting a pointer might help you sometimes. Once you have the pointer you can manipulate the internal data contents of an object directly but on your own head be it. We do not guarantee that internal representations will stay the same!

So, which do you use, the full data hiding capabilities of EMBOSS or the more direct "get the pointer" approach? If you are a purist then this is a rhetorical question. If you use common sense it is perfectly acceptable to mix the two and, after all, you do document your code don't you? I make the following recommendation. **There is no real stigma in getting the char* pointer of a string object BUT, whenever the size of the data in an object can increase then use the library functions!** The rule of no arbitrarily sized arrays must never be broken but the data hiding rule can be bent if it makes things more efficient or more intuitive. Use good judgement, the choice is yours. As you program more in EMBOSS you'll find yourself using the library functions in preference to the standard C library. A lot of code will continue to use pointers though, we are dealing with genome sized sequences and the lower the computational overhead the better.

For most objects there is a so-called *cast* function provided that lets you get at the data directly. For example the **AjPInt** object for dynamic integer arrays has an associated casting function called *ajIntInt*.

6.2 Constructing and Destructing.

It is good and recommended programming practice to construct objects before you use them. You might even say its essential but there you'd be partially wrong. Take the following code segment:

```
AjPStr str=NULL;
        str = ajStrNew();
        ajStrAssC(&str, "This is a string");
        ajStrDel(&str);
```

First a string object pointer called *str* is declared. It is initialised to point to null. That itself is good practice. To create the object a *constructor* function is called. That is the line containing *ajStrNew()*. After that another library function is called which copies (assigns) a character string to the string object. Some work is done after this and then, as the string is no longer required the object is deleted and thereby its associated memory is freed. That is the proper way to write the code. Here is the wrong way:

```
AjPStr str=NULL;
            ajStrAssC(&str, "This is another string");
            ajStrDel(&str);
```

The interesting thing is that this will still work. Whenever they can be the library routines tend to be smart. If a NULL pointer is passed to them they'll recognise the fact and construct a string for you. It can get you out of trouble, it can also cause trouble. So, please say what you mean when writing code. **Explicitly construct use and deconstruct**. There is no advantage to not using the constructor (that just means the assignment function has to call it internally anyway) , it is a good reminder to you that memory has been allocated and will most likely cause you to spot that it needs deleting when you've finished with it. In short, good programming practice helps stop applications leaking memory like a sieve.

7. A SIMPLE EMBOSS PROGRAM

As a bit of light relief lets put a bit of all this into action. This will lead to a description of the structure of an EMBOSS program. Here is an application that will read in a sequence and tell you the GC fraction. We'll call this program **gcf.**

The first thing to do is to create an ACD file. We require a sequence and an output file for the results. As they are both essential we'll make them parameters

```
appl: gcf [
      doc: "Work out GC fraction"
      groups: "DNA: sequence composition"
]

 sequence: green [ param:Y type DNA ]
 outfile: boggo [ param: Y ]
```

And that's it for the ACD. Call the file **gcf.acd** and put it in the emboss/acd directory. Now for the application itself.

```
#include "emboss.h"

int main(int argc, char **argv)
{
   AjPSeq seq=NULL;
   AjPStr str=NULL;
   AjPFile outf=NULL;
   char *p;
   ajint  len;
```

```
    ajint  count=0;
    ajint  begin;
    ajint  end;
    ajint  c=0;


    embInit("gcf",argc,argv);


    seq  = ajAcdGetSeq("green");
    outf = ajAcdGetOutfile("boggo");

    str = ajStrNew();

    begin = ajSeqBegin(seq);
    end   = ajSeqEnd(seq);

    p = ajSeqChar(seq);
    ajStrAssSubC(&str,p,--begin,--end);
    ajStrToUpper(&str);


    p = ajStrStr(str); /* see section 10.2.6 for the "proper" data hiding method */
    len = ajStrLen(str);

    while(*p)
    {
        c = *p++;
        if(c=='G' || c=='C')
            ++count;
    }

    ajFmtPrintF(outf,"GC Fraction = %f\n", (float)((float)count/(float)len));

    ajStrDel(&str);

    ajExit();
    return 0;
}
```

Call this file **gcf.c** and put it in the **/emboss** directory. Now edit **emboss/Makefile.am** and add **gcf** to the **bin_PROGRAMS** line and add a line saying **gcf_SOURCES = gcf.c** . Now type **make**. Try it on a DNA sequence and check that it works.

7.1 The include directive

The compiler directive **#include "emboss.h"** imports the entire EMBOSS interface i.e. makes all the EMBOSS library calls available to you. This must be included at the start of every EMBOSS program.

7.2 Importing the command line

The command line must be available to the application so the **main** function must include it. This is done in the parameter list using **int main(int argc, char **argv)**

7.3 Declarations

Here we meet the objects for the first time. We said in the ACD that a sequence is required so we need a sequence object (**AjPSeq**). An output file is required as well so a file object is needed (**AjPFile**). Why an object for a file? Its so that EMBOSS can get or put input and output from anywhere using the same system. A file could be local or it could be an internet connection. A string object (**AjPStr**) is also needed for some processing. Note their initialisation to NULL, not really required but good practice.

The remaining declarations are plain ANSI C and should be familiar.

7.4 Processing the ACD

The next line, or one like it, appears in all EMBOSS programs. **embInit("gcf",argc,argv);** This one line hides wealth of activity. It reads in local database definitions, finds the right ACD file to use (from the **"gcf"** parameter) , processes the command line (it uses argc and argv from main) and, by the time the call returns it has read in the sequence and put it somewhere in memory and has also opened the output file. Remember I've said before that all of the user input processing is done before the application starts with a vengeance, well that one line does it.

N.B.: If you are doing graphics then use **ajGraphInit("prog",argc,argv)** instead.

7.5 Retrieving the ACD values

The ACD has placed all the values we want somewhere in memory. We now need to retrieve them. This is done by the **ajAcdGet** family of functions. Taking the sequence as an example, note that the sequence is referred to by the *name* it was given in the ACD file and not by the *datatype*. I gave the sequence such a strange name just to make this point clear. It would have been less obvious if the ACD had declared **sequence: sequence [param:Y type: DNA]** as it should have done. The output file pointer is received similarly.

Saying that the call has "put the values somewhere in memory" is just another way of saying that the ajAcdGet functions are all constructors. There is therefore no need to explicitly create a new object, in fact it should be avoided as memory would be leaked.

7.6 Constructors

It is a good habit to put all the constructors together at the beginning of the application. They are therefore declared immediately after the ajAcdGet calls. In this case the only object that has not yet been created is the string object. This is done by the **str = ajStrNew();** call.

7.7 The body of the application

How you program the body of the application depends on style. This example shows a reasonable mixture of EMBOSS code and standard C. There are points to note though. The ajAcdGetSeq call will always return the whole sequence. The user might have specified a start and end position on the command line though (using the **-sbegin** and/or **-send** built-in qualifiers). A substring of the sequence must therefore be processed. That's why the string object is needed. I've written this example long-windedly to make the process clear. First, the begin and end values are received. These will be between 1 and "n" where n is the length of the sequence; that means, for C purposes, the returned begin and end values will need decrementing. Next a char* pointer to the sequence is returned. Finally the substring is assigned to the string object. The rest should be clear, the ajFmtPrintF call is the EMBOSS equivalent of fprintf but uses the file object.

7.8 Destructing

We are clean programmers, right? So any memory we consume we're going to tidy up at the end and not rely on the operating system to clear up our mess. Destruct any objects you've created at the end of the program **with the exception of any ACD-constructed objects**, the latter should be one of the jobs of the **ajExit();** call. In this case only the string object needs deleting.

7.9 Exit gracefully

main was declared as an int and it is good practice to return zero on success. That is done at the end with the return statement.

8. THE LIBRARIES AND PROGRAMMING

8.1 Introduction

It is not really the scope of this guide to provide you with a full list of all the library functions. The tropical rain forests are one consideration but tedium is the main one. So, given that there is currently about a million lines of code then where does the programmer start? The best solution is for you to download the function and datatype listings (see later), use the online indexing of the source code, and then use this guide which will give examples of how to use the functions.

8.2 The Function Listings

I strongly recommend that you go to the web page http://www.uk.embnet.org/Software/EMBOSS/Doc/ where you will find two important pages. These are **Ajax Library Documentation** and **Nucleus Library Documentation**. They are both important but you will be using AJAX most of the time so I'll use the former page as an example, they both have the same features. On that page you will see the library split into logical sections such as *strings, lists, tables, arrays, memory* etc. These sections follow the way the code is supplied in the distribution so, for example, for the string functions there are the two files **ajax/ajstr.c** and **ajax/ajstr.h**.

These sections have associated **function** and **datatype** links. Go to each of these sections and save them out as postscript and print them (or use some other appropriate method for printing.) Then, what you probably don't want to do is read them from start to finish. You can if you like but the best thing is to **use them as a reference manual!** On these pages, which are generated automatically and are therefore up-to-date the functions, where appropriate are split into logical sections concerning constructors, destructors, assignments, operators, casts etc. Have a flick through to get the general idea. You'll hardly ever need to look at the datatypes unless you intend writing a library function. For applications it is the functions that are important.

The source code itself is pretty much self-documenting and one useful thing you can do is to look at the **ajax/*.h** files which often list the functions alphabetically and give a short description.

One of the quickest ways to find something or see if it exists (which it probably will) is to use the online indexed facility.

8.3 The Online Indexed Library

The libraries are indexed using a piece of software called SRS. This enables you to browse the functions or search for one. Go to the **Ajax Library Documentation** page on the webserver (see 8.2). At the top of the page you'll see two databases highlighted. These are **EFUNC** and **EDATA**. The EFUNC database holds information about the functions; the EDATA database holds information about the datatypes.

Taking the simple program of section 7 as an example, suppose you'd forgotten what the ajStrAssSubC function was called but you knew that it dealt with substrings and char arrays and that it did an assignment. **Click on EFUNC**. From there **click on Search** at the top right of the page. On the form that is presented type:

substring & cop

and then **click on Submit Query**. You'll see two functions are found *ajStrAssSub* and *ajStrAssSubC*. Take a look at both of them. You'll see that even the source code is indexed as well as the description line. Clicking on a function another calls will take you to that function.

8.5 Nomenclature

All the ajax functions start with ***aj***. These letters are followed by a short indicator of what general section they belong to so ***ajStr*** functions deal with strings and ***ajSeq*** functions deal with sequences.

Then follows a short description of what they do so ***ajStrAss*** is an assignment function, in C terms it is a form of strcpy. If the last letter of the name is a capital ***C*** then that function deals with char* strings.

The NUCLEUS library has similar nomenclature except that the function names start with ***emb*** standing for EMBOSS.

Now you know how to find the functions the next sections show you how to use them.

9. TRUE OR FALSE

EMBOSS has its own names for true and false values. These are values an **AjBool** datatype can hold. The values are:

ajTrue

ajFalse

10. STRINGS

10.1 Datatype, constructors and destructors

The main datatype is the **AjPStr** and there are two constructors. Here are code segments for them both.

```
AjPStr str=NULL;
        str = ajStrNew();
```

```
AjPStr str=NULL;
str = ajStrNewC("Hello World");
```

The first just allocates a new string object, the second does the same but initialises it with a given character string. There is one destructor.

```
        ajStrDel(&str);
```

10.2 Useful, commonly used, string functions.

There are tens and tens of these functions. They are the workhorses of the ajax library. Just take a look at your printed function library and you'll see the ajstr section is rather thick. Here are some of the very commonly used functions with some helpful hints.

10.2.1 Assignment (copying) and Appending (catenating)

```
        AjBool ajStrAss(AjPStr  *string1, AjPStr  string2);

        AjBool ajStrAssC(AjPStr *string1, char *string2);
```

These functions both return an AjBool which says whether or not more memory had to be allocated to the string. That isn't usually that useful so the return value is ignored. They are therefore usually called like:

```
        ajStrAss(&string1, string2);

        ajStrAssC(&string1, "Hello world");
```

but if you are programming properly you should explicitly say you want to ignore the return value by writing:

(void) ajStrAss(&string1, string2);

(void) ajStrAssC(&string1, "Hello World");

We tend to insist on that in library functions as it is proper ANSI C. For applications we are more generous but occasionally go through code and tidy it up.

Here is a useful tip:

You will frequently find constructs such as:

ajStrAssC(&string1,ajStrStr(string2));

in some programs. Why? The answer is that a plain assignment using string objects might just increment a string usage counter but **using character assignments guarantees a discrete string object with a usage count of 1.** Sometimes you can't do without a unique string! In the above the AjPStr string2 has its character pointer derived by the ajStrStr call and that pointer is used for the assignment.

For assignment of substrings use:

*ajStrAssSub(AjPStr *string1, AjPStr string2, ajint start, ajint end);*

*ajStrAssSubC(AjPstr *string1, char *string2, ajint start, ajint end);*

There are three ways of catenating with a string object:

*ajStrApp(AjPStr *string1, AjPStr string2);*

*ajStrAppC(AjPStr *string1, char *string2);*

*ajStrAppK(AjPStr *string1, char c);*

These are the equivalents of the C strcat function.

10.2.2 Case changes

These functions will change the case of a string object.

*ajStrToUpper(AjPStr *string);*

*ajStrToLower(AjPStr *string);*

They can be indispensible as sequence reading, quite rightly, doesn't change the case of a string but some databases have their sequences in lower case and others in upper case.

10.2.3 String Comparison

There are many of these functions. Again I'll limit this to the common ones.

These are the equivalents to the C strcmp function. They will return true if the strings are of equal length and exactly match.

AjBool ajStrMatch(AjPStr string1, AjPStr string2);

*AjBool ajStrMatchC(AjPStr string1, char *string2);*

They have two equivalents which are case insensitive, these being:

AjBool ajStrMatchCase(AjPStr string1, AjPStr string2);

*AjBool ajStrMatchCaseC(AjPStr string1, char *string2);*

Wild card matching is also provided by the functions:

AjBool ajStrMatchWild(AjPStr string1, AjPStr string2);

*AjBool ajStrMatchWildC(AjPStr string1, char *string2);*

The equivalent functions to the C library call strncmp which matches N characters are:

ajint ajStrNCmpO(AjPStr string1, AjPStr string2, ajint n);

*ajint ajStrNCmpC(AjPStr string1, char *string2, ajint n);*

but note that they return 0 (ajFalse) if the strings match. They are mainly used by the library itself.

Other matching routines allow you to test prefixes, suffixes etc.

10.2.4 String Length

That of course had to be in the library. And is:

ajint ajStrLen(AjPStr string);

Unlike the C function strlen this is a very speedy function as the string object already contains the length of the string.

10.2.5 Tokenising a string

AJAX has its own tokenisation functions. These include a constructor and a destructor. As an example, to tokenise the string **"lets:tokenise:this"** into its three colon-delimited components a code segment example would be:

```
AjPStr string=NULL;
AjPStr result=NULL;
AjPStrTok token=NULL

        result = ajStrNew();
        string = ajStrNewC("lets:tokenise:this");
        token = ajStrTokenInit(string, ":");

        ajStrToken(&result, &token, ":");        /* result =lets */
        ajStrToken(&result, &token, ":");        /* result = tokenise*/
        ajStrToken(&result, &token, ":");        /* result = this */

        ajStrTokenClear(&token);
        ajStrDel(&result);
```

You can see that this is a bit like the C function strtok.

The three ajStrToken() calls return the three elements one at a time. The program would probably want to do something with them but this is just an example code segment. The **AjPStrTok** is the string tokenisation object, **ajStrTokenInit** is its constructor and **ajStrTokenClear** is its destructor.

Handy hint:

The function **ajStrTokenCount** will tell you how many tokens a string will produce.

Handy hint 2:

Tokenisation is a valuable technique for reading in data. It can be used, with other functions, to emulate the C function sscanf.

10.2.6 String Iterators

String iterators allow you to step through the contents of a string one position at a time. These iterators have their own constructors and destructor. The datatype is **AjIStr**, the constructor is **ajStrIter** and the destructor is **ajStrIterFree**. Its use is best illustrated by showing how the simple GC content program in section 7 should have been written to avoid violating data hiding principles.

This is the modified program:

```
#include "emboss.h"

int main(int argc, char **argv)
{
    AjIStr iter;
    AjPSeq seq=NULL;
    AjPStr str;
    AjPFile outf=NULL;
    ajint  count=0;
    ajint  begin;
    ajint  end;
    ajint  len;
    char c='\0';


   embInit("wibble",argc,argv);


    seq  = ajAcdGetSeq("green");
    outf = ajAcdGetOutfile("boggo");

    str = ajStrNew();

    begin = ajSeqBegin(seq);
    end   = ajSeqEnd(seq);

    ajStrAssSubC(&str,ajSeqChar(seq),--begin,--end);
    ajStrToUpper(&str);


    len = ajStrLen(str);
```

```
    iter = ajStrIter(str);
    while(!ajStrIterDone(iter))
    {
        c = ajStrIterGetK(iter);
        if(c=='G' || c=='C')
            ++count;
        ajStrIterNext(iter);
    }

    ajFmtPrintF(outf,"GC Fraction = %f\n",(float)((float)count/(float)len));

    ajStrIterFree(&iter);
    ajStrDel(&str);


    ajExit();

    return 0;
}
```

Note that there isn't a char* pointer in sight. The *iter* iterator is constructed using the AjPStr *str*. The *ajStrIterDone* returns ajTrue if there are no more characters left in the string. The *ajStrIterGetK* function returns the current character in the string object and the *ajStrIterNext* function moves the iterator on to the next character.

10.2.7 Too many to mention

The above shows the main features of using strings. There are loads of string functions left unmentioned but these are easily perused online. String functions exist for trimming, inserting, reversing, etc. Look at the function documentation and browse the EFUNC and EDATA databases!

11. LISTS

I'll say it again Arbitrary array sizes are forbidden in EMBOSS programs. Hopefully you'll be so sick of me saying that soon, or already, that it'll be burned into your long term memory. A typical position with databases is that you don't know how many entries there are until you've reached the end; or that you cannot know how many hits you're going to get using a given query. How can you cater for that without using arbitrary arrays? The answer is, of course, lists. EMBOSS lists can be used to implement both FIFO or LIFO stacks. Also, like string objects, they have their own iterator.

Lists are kept intentionally general in EMBOSS, they have to deal with many different types of objects. That is why you'll see they use **void*** address pointers. It's the ultimate in non-specific pointers. There is one exception to this rule though. As string objects are so common they have their own list operations. I'll describe them first. Please read 11.1 before skipping to general lists as the principles are the same.

11.1 String Lists

The datatype for a string list object is the **AjPList**, this datatype is also used for all the other list operations. The constructor for string lists is the function *ajListStrNew* but there are two destructors namely *ajListstrFree and ajListstrDel*. The difference is that the former will delete all the strings in the list plus the list itself, the latter will not delete the strings but will delete the list. A sample code segment would be:

```
AjPList list=NULL;

        list = ajListstrNew();
        ajListstrFree(&list);
```

11.1.1 LIFO lists

LIFO (last in first out) lists can be regarded as the default for EMBOSS as far as function naming is concerned. You put something onto a string list using the **ajListstrPush** function. You get something off the list by using the **ajListstrPop** function. Here is another rule to remember. **It is only the address of an object that gets pushed onto a list**. There is absolutely no copying of objects. Here is a common error.

```
AjPList list=NULL;
ajint i;
AjPStr str=NULL;

        str = ajStrNew();
        list = ajListstrNew();

        for(i=0;i<10;++i)
        {
                ajFmtPrintS(&str,"%d",i);
                ajListstrPush(list, str);
        }
```

The ajFmtPrintS function will print 0->9 to the string object. Its like sprintf in C. So, what does that code segment do? Well, if you're lucky, when you pop things off the list you'll find all the strings contain "9", if you're unlucky it'll crash (not very likely with this example but if the print statement caused *str* to be reallocated the accessing what was returned most likely would cause a crash). Why is it doing that? Yes, because only the address of the string is being pushed. The expected behaviour can be obtained by creating a new string each time around the loop. This code is correct:

```
AjPList list=NULL;
ajint i;
AjPStr str=NULL;

list = ajListstrNew();

for(i=0;i<10;++i)
{
        str = ajStrNew();
        ajFmtPrintS(&str,"%d",i);
        ajListstrPush(list, str);
}
```

To pop these items off the list we can use ajListstrPop. Something like the following would complete the program:

```
AjPList list=NULL;
ajint i;
AjPStr str=NULL;
AjPStr tmp=NULL;

list = ajListstrNew();

for(i=0;i<10;++i)
{
        str = ajStrNew();
        ajFmtPrintS(&str,"%d",i);
        ajListstrPush(list, str);
}

        for(i=0;i<10;++i)
        {
                ajListstrPop(list,&tmp);
                ? print out the string or do something else with it ?
                ajStrDel(&tmp);
        }

        ajListstrDel(&list);
```

There are two points of interest in the above. One is that I needn't have used *tmp*, I could have reused *str* instead (why?) but it arguably made the code easier to read. The second is that I could replace the second *for* loop with a *while* loop since the ajListstrPop function returns ajTrue if there was something on the list. So:

```
while(ajListstrPop(list,&str))
{
        ? print out the string or do something else with it ?
        ajStrDel(&tmp);
}
```

would do the same thing. Note the need to delete the strings when they're finished with.

11.1.2 FIFO Lists

FIFO (first in first out) lists are easy to implement. The code is precisely the same as that for LIFO lists with one exception. The function **ajListstrPushApp** is used instead of *ajListstrPush*. The former function appends an entry onto a list. This will mean the first item to be put on the list will be the first to be popped.

11.1.3 Reversing the order of a list

Easy. To reverse the order of entries in the list in section 11.1.1 you only have to type:

```
ajListReverse(list);
```

Note that pushing items onto a list and then reversing the list is another way to make a FIFO list.

11.1.4 Sorting a string list

Sorting a list is a bit trickier to understand if you haven't used routines like *quicksort* before. The prototype of the list sorting function is:

ajListSort(AjPList list, ajint (*compar) (const void *, const void *));

What the function needs is another function which will return 0 if two strings are the same and a positive number if the second should sort before the first or a negative number if the first should sort before the second. For string lists such a routine is already provided in the library and is called **ajStrCmp**. So, to sort the list in 11.1.1 (which is already sorted but what the heck) you'd use:

ajListSort(list,ajStrCmp);

11.1.5 Getting the number of entries in a list

Use the **ajListLength** function

11.1.6 List Iteration

Lists can be iterated through much as string objects can. Iterating through a list is useful as the list can be scanned through more than once without having to pop values off all the time. The syntax is pretty much the same as for strings. The list iterator datatype is the **AjIList**. Here is a way that 11.1.1 could be rewritten using an iterator.

```
AjPList list=NULL;
AjIList iter=NULL;
ajint i;
AjPStr str=NULL;
AjPStr tmp=NULL;

list = ajListstrNew();

for(i=0;i<10;++i)
{
        str = ajStrNew();
        ajFmtPrintS(&str,"%d",i);
        ajListstrPush(list, str);
}

        iter = ajListIter(list);

        while(ajListIterMore(iter))
        {
                tmp = (AjPStr) ajListIterNext(iter);
                ? print out the string or do something else with it ?
        }
```

```
ajListIterFree(iter);
ajListstrFree(&list);
```

The ajListIterMore tests to see if the iteration can continue. Note that the ajListIterNext function returns a void* pointer so a cast has to be used to tell the compiler that its really an AjPStr. I've used the ajListStrFree function this time for a bit of variety. This means the ajStrDel function calls are unnecessary.

List iteration has other advantages. There are library functions, *ajListstrRemove* and *ajListstrInsert*, which can add or remove items in the middle of a list. They are effectively pops and pushes anywhere in the list. This can only be done using iterators.

11.1.7 Turning Lists into arrays

It is sometimes more convenient to work with an array than with a list, once the list has been created. EMBOSS provides functions for this. If we were to apply the following code segment to the list from 11.1.1:

```
AjPStr *array;
ajint len;

        len = ajListstrToArray(list,&array);
```

Will set len to 10 and make *array* an array of ten AjPStr objects which can be referred to as array[0] to array[9]. The list is left unchanged.

Please note **the above cannot be written as**:

```
AjPStr **array;
ajint len;

        len = ajListstrToArray(list,array);
```

which will compile but crash when run, for rather obvious reasons given a little thought!

We would, of course, delete the array with code like:

```
for(i=0;i<len;++i)
        ajStrDel(&array[i]);
AJFREE(array);
```

11.2 More General Lists

Some of the functions are precisely the same. The equivalents of the ones that aren't have the same function names without the three characters **str** therefore *ajListStrPush* becomes just **ajListPush**. The second difference is that there is only one destructor **ajListDel** which only deletes the list and not the objects themselves. The other major difference, as I've said, is that you must use casts as the general functions use void* pointers. Let me illustrate this using the same examples but this time having lists of AjPCod codon objects.

11.2.1 The general LIFO list

The example from 11.1.1 becomes:

```
AjPList list=NULL;
ajint i;
AjPCod cod=NULL;
AjPCod tmp=NULL;

list = ajListNew();

for(i=0;i<10;++i)
{
        cod = ajCodNew();
        ? do something to the codon object and then ?
        ajListPush(list, (void *)cod);
}

        while(ajListPop(list,(void **)&tmp))
        {
                ? do something with the codon  object then ?
                ajCodDel(&tmp);
        }

        ajListDel(&list);
```

Note the void* and void** casts.

11.2.2 General FIFO lists

Just the same as for string lists except **ajListPushApp** is used with a (void *) cast.

11.2.3 General List Reverse

Exactly the same as for strings

11.2.4 General List Sorting

This uses the same ajListSort function however we now most probably will have to write our own comparison function and be very careful with casting otherwise the program will not compile. Let us assume we had an object defined as;

```
typedef struct AjSWibble
{
        ajint granny;
        ajint weatherwax;
} AjOWibble, *AjPWibble;
```

and we had a list of AjPWibble objects which we wanted to sort on the basis of the granny field. Our comparison function must return an ajint with 0 if both values are the same or a positive or negative value depending on whether one should sort before the other. Our comparison function would be written like:

```
ajint wibblecompare(const void *a, const void *b)
{
        return (*(AjWibble*)a)->granny - (*(AjWibble*)b)->granny;
}
```

depending on the preferred sort order. And the sort function becomes:

```
ajListSort(list,wibblecompare);
```

11.2.5 Getting the number of entries in a general list

Use the **ajListLength** function i.e. exactly the same as for strings.

11.2.6 List Iteration

The previous example using codon objects becomes:

```
AjPList list=NULL;
AjIList iter=NULL;
ajint i;
AjPCod cod=NULL;
AjPCod tmp=NULL;

list = ajListNew();

for(i=0;i<10;++i)
{
        cod = ajCodNew();
        ? do something ?
        ajListPush(list, (void *)cod);
}

        iter = ajListIter(list);

        while(ajListIterMore(iter))
        {
                tmp = (AjPCod) ajListIterNext(iter);
                ? print out the string or do something else with it ?
        }

        ajListIterFree(iter);

        while(ajListPop(list,(void **)&tmp))
                ajCodDel(&tmp);

        ajListDel(&list);
```

Note the casts AND the extra step of deleting the objects and the list separately at the end.

11.2.7 Turning General Lists into arrays

Like string lists but with casts.

```
AjPCod *array;
ajint len;

        len = ajListToArray(list,(void ***)&array);
```

Will set len to 10 and make *array* an array of ten AjPStr objects which can be referred to as array[0] to array[9]. The list is left unchanged.

11.3 An End To Lists

As with string objects there are lots more things lists can do but they use the same principles as given above. For example, the **ajListMap** function applies another function to every member of the list; the principle here is very similar to that of the sort function except that there is only one parameter in the map function. It can be useful for emptying lists.

12. DYNAMIC NUMERIC ARRAYS. Pints, Doubles and Shorts (and floats and longs)

Much as arbitrary length string arrays are a no-no so are arbitrary numeric arrays. Sometimes you'll know that an integer array will never need to be larger than the some function of another number, such as the length of a sequence. In that case its acceptable to directly AJALLOC such memory. Sometimes you will not know or don't want to know or maybe it would be wasteful to allocate too much for most circumstances. This is where dynamic numeric arrays (DNAs) come in. AJAX allowed you to use one, two or three dimensional arrays of ajints, ajshorts, floats, doubles and ajlongs.

12.1 One dimensional integer arrays

This is the most simple case. The datatype is the **AjPInt**, the constructor is **ajIntNew** and the destructor is **ajIntDel**. Here is a code segment:

```
AjPInt array=NULL;
ajint v;

        array  = ajIntNew();
        ajIntPut(&array, 27, 15);
        v = ajIntGet(array, 27);
        ajIntDel(&array);
```

This example stores the value 15 at position 27 in the equivalent of a one dimensional integer array (ajint*), and then retrieves it. As many values can be put into the array as you like at any position. The code will automatically take care of the memory allocation. N.B. It is an error to access an unfilled element. This can be avoided by conversion to a normal array (see below).

Hint: If you have at least some idea of the size of the array then you can use the **ajIntNewL(n)** constructor instead. This doesn't mean the array is restricted to n values but does mean that the library will have to worry less about memory reallocation. Alternatively, if your code calculates the maximum position first and fills that value then less memory reallocation will need to be done. In short, filling the array from the top is more efficient than filling from the bottom. This is only really a consideration for very large arrays.

At any time you can get the length of the array by using :

```
ajint n;
        n = ajIntLen(array);
```

You may also find it more convenient, after filling an array, to convert the array to a standard C integer array. This can be done as follows:

```
ajint *a;

        a = ajIntInt(array);
```

You can then access the elements using a[0] -> a[n] as usual but remember that the DNA still exists so you may want to delete it.

N.B. All unfilled elements will have a zero value.

12.2 Two dimensional integer arrays

These are very similar to one dimensional arrays. The datatype is the **AjInt2d** and some example code is:

```
AjPInt2d array;
ajint v;

        array  = ajInt2dNew();
        ajInt2dPut(&array, 27,8, 15);
        v = ajInt2dGet(array, 27,8);
        ajInt2dDel(&array);
```

This fills the 8th column of the 27th row with the value 15 and then retrieves it.

Getting the dimensions for multidimensional arrays has a slightly different syntax.

```
ajint rows;
ajint columns;

        ajInt2dLen(array, &rows, &columns);
```

will return the row and column values. Converting to a normal array is just the same:

```
ajint **a;

        a = ajInt2dInt(array);
```

12.3 Three dimensional arrays

These have the same syntax as two dimensional arrays. The datatype is **AjPint3d**.

12.4 Floats, Doubles, Shorts and Longs

All these have their equivalent functions to the integer ones and all have the same syntax as for the integer ones. Note though, that although the datatypes of the values change the row/column/etc positions are still specified as integers.

13. TABLES

Tables in EMBOSS allow you to add key/value pairs and then retrieve an associated value for a given key. As with lists there are two types. One is where the key is a string object, the other where the key is more general. As it is nearly always the case that strings can be used for most keys I'll mainly discuss these. The advantage of using strings as keys is that the library calls are easier to understand. A full discussion of general tables may be given in future versions of this document. General tables have not needed to be used by either the library or any application so far.

For all types of table the datatype is the **AjPTable**. The tables are automatically hashed to provide speedy access to the keys. Like some other functions it is more efficient to have some idea of the size the table will need to be before creating it. It is not critical though and more memory will always be allocated if the table grows beyond the estimate.

Here is some example code using a table, it uses strings for both keys and values:

```
AjPTable table=NULL;
```

```
AjPStr key=NULL;
AjPStr value=NULL;
ajint I;

static char *keys[] =
{
        "Sheridan", "Garibaldi", "Londo", "Vir", NULL
};
static char *values[]=
{
        "One", "Two", "Three", "Four", NULL
};

        table = ajStrTableNew(10);

        i=0;
        while(keys[i])
        {
                key = ajStrNewC(keys[i]);
                value = ajStrNewC(values[i]);

                ajTablePut(table,(const void *)key, (void *)value);

                ++i;
        }

        key = ajStrNewC("Garibaldi");
if(!(value = ajTableGet(table, (const void *)key)))
        ajFatal("Cannot find key");

        ? value is now a string object holding "Two" ?

        ajStrTableFree(&table);
        ajStrDel(&key);
```

So you can see that **ajStrTableNew** is the constructor and **ajStrTableFree** is the destructor, if the string objects are real and not just copies of other string pointers then this function deletes the strings as well. If you just want the table deleted use **ajTableFree** instead. The value 10 is just a guess at the number of values, we know it is really 4 as its hard coded into the character arrays but even examples in this documentation are given in the spirit of the thing.

Just like lists only the addresses are put into the table so new key and value objects need allocating for each addition. The function that adds the pairs is **AjTablePut**. This function is also used for general lists so void casts need to be applied.

The last part of the example allocates one of the strings we know is there to the AjPStr key and then uses **ajTableGet** to return the associated value (in this case "Two"). The ajTableGet function is also used by general lists and so the void cast is required. If a key is given which doesn't exist in the table then NULL is returned.

You can get the length of a table with **entries = ajTableLength(table);**, you can remove key/value pairs, you can apply a function to all the key/value pairs with **ajTableMap** and convert a table to a 2N+1 array (the last value being a delimiter usually NULL), an example of the latter for the above would be:

```
AjPStr *array;

        array = (AjPStr *) ajTableToArray(table,NULL);
```

14. FORMATTED OUTPUT

There are only a handful of routines in ajfmt.c that are used to any degree in EMBOSS applications but they are some of the most important. After all, it is occasionally useful to print out the odd result. I'll lump warnings and errors into this section.

14.1 ajFmtPrintF

This can be regarded as the equivalent of the C routine fprintf. It takes an AJAX file object which has previously been opened as an output stream to somewhere (local file, network etc) and then uses a varargs approach for printing data. File objects are discussed in section 15 but for now assume that we have an AjPFile object called **outf** opened where we want it.

An example is all that is required to explain this function:

```
ajint i=5;
float fl=6.7;
static char *fred="Hello";

        ? assumes open file object outf ?
        ajFmtPrintF(outf,"%s %d %f\n",fred,i,fl);
```

will print out "Hello 5 6.7" followed by a newline.

This function can cope with all the format specifiers of the standard C library and one or two more that are specific to EMBOSS. Rather then having to specify %s and cast a string object to char* you can print a string object directly using **%S** i.e.

```
AjPStr str=NULL;

        str = ajStrNewC("Hello world");
        ? assumes open file object outf ?
        ajFmtPrintF(outf,"%S\n",str);
```

even something like **%60.60S** will work. The other differences are:

- **%S** as above
- **%D** print a date
- **%B** print an AjBool value
- **%s** will accept a NULL pointer and print "<null>"

14.2 ajFmtPrintS

This is just like ajFmtPrintF only it prints to a string object hence:

```
AjPStr str=NULL;

        str = ajStrNew();
        ajFmtPrintS(&str,"%s %d %f","Hello",5,6.7);
```

will leave *str* holding "Hello 5 6.7".

14.3 ajFmtPrintAppS

Like ajFmtPrintS but appends to a string object

14.4 User messages, Warnings, Errors and Fatalities

Messages at various levels of severity can be sent to the user. There are five functions for this.

- **ajUser** used to print messages to the user
- **ajWarn** used for printing warnings
- **ajErr** used for printing errors
- **ajDie** prints non-recoverable errors and aborts the application
- **ajFatal** prints fatal error messages and aborts the application

All these functions have varargs capability like the FmtPrint functions e.g.

```
ajFatal("Cannot open file %S",filename);
```

14.5 Debugging

The **ajDebug** function is like **ajUser** except that it will print messages to the debugging file if the user has specified **-debug** on the command line.

15. INPUT AND OUTPUT FILES

There are many variations on opening input and output files in EMBOSS and so I'll deal with the basics here; it is worth perusing the functions in ajfile.c after you've got the idea. The datatype **AjPFile** is the standard one (although there is another form for buffered files).

15.1 Opening a file for reading

The function **ajFileNewIn** is used. The filename is passed as an AjPStr. Here is a code segment:

```
AjPFile inf=NULL;
AjPStr fn=NULL;

        fn = ajStrNewC("my.file");
        if(!(inf = ajFileNewIn(fn)))
```

```
        ajFatal("Cannot open file %S for reading",fn);
```

15.2 Opening a file for writing

The function **ajFileNewOut** is used.

```
AjPFile outf=NULL;
AjPStr fn=NULL;

    fn = ajStrNewC("my.file");
    if(!(outf = ajFileNewOut(fn)))
        ajFatal("Cannot open file %S for writing",fn);
```

15.3 Closing input and output files

The same function, **ajFileClose**, is used for both e.g.

```
AjPFile outf=NULL;

    ? assumes open file object ?
    ajFileClose(&outf);
```

15.4 Files in the EMBOSS data areas

If a program requires a data file (e.g. a BLOSUM matrix) then it first looks in the user's current directory, a few other directories as documented elsewhere, and finally looks in the EMBOSS data directory. The function to open such a data file for reading which will look in all those locations is **ajFileDataNew**. It has a rather unstandard calling method. Here is an example of its use:

```
AjPFile inf=NULL;
AjPStr str=NULL;

    str = ajStrNewC("BLOSUM100");
    ajFileDataNew(str,&inf);
    if(!inf)
        ajFatal("Cannot open data file %S",str);
```

To open a file for writing in the top level EMBOSS data directories use the **ajFileDataNewWrite** function:

```
AjPFile outf=NULL;
AjPStr str=NULL;

        str = ajStrNewC("BLOSUM666");
        ajFileDataNewWrite(str,&outf);
        if(!outf)
                ajFatal("Cannot open data file %S",str);
```

15.5 Input from other commands

EMBOSS can accept data from external applications. It does this by means of pipes. To use a pipe as input the **ajFileNewInPipe** is used. The filename for this function is a remote command.

15.6 Reading from multiple files

There are several variations on this. For example **ajFileNewInList** will accept a list of filenames and create a file object with the first file open. After reading all of the file a call to **ajFileNext** will close the current file and open the next in the list. When the function returns **ajFalse** there are no more files left to read. An alternative is to use **ajFileNewDW** in combination with **ajFileNext**. You cam give this a directory and a wildcard string; all matching files will be found and the reading then proceeds as for the previous example.

15.7 Buffered Input

Equivalent functions are available for buffered input using the datatype **AjPFileBuff**. To open a file for reading use **ajFileBuffNew**, to close it use **ajFileBuffDel** etc.

```
AjPFileBuff buff=NULL;
AjPStr str=NULL;

        str = ajStrNewC("my.file");
        if(!(buff = ajFileBuffNew(str)))
                ajFatal("Cannot open input file %S",str);
        ? add something here ?
        ajFileBuffDel(&buff);
```

As the name suggests, reading from buffered files can be more efficient as more data is slurped into memory at any one time. You have to be a little careful that you clear buffers etc when you use them.

Not all the file functions can work with buffered files, logical impossibilities for a lot of them. There is normally little overhead in using normal file objects as the operating system normally buffers things anyway.

The function **ajFileBuffLoad** will read all the file into the buffer. Not recommended for large databases and small computers.

15.8 Seeking, Raw Reading, Position and other C-ish equivalents

- **ajFileSeek** is the equivalent of the C fseek function
- **ajFileTell** is the equivalent of the C ftell function
- **ajFileStat** is the equivalent of the C stat function

15.9 Handy File functions

- **ajFileName** will return the filename of a file object as a char* pointer
- **ajFileStdin, ajFileStderr, ajFileStdout** these test to see if file objects point to a particular stream and return ajTrue if they do.

16. READING FROM FILES

16.1 Reading a line from a standard file

Use the function **ajFileReadLine**. This has the advantage that it will strip trailing newlines.

Assuming an open input file object *inf*:

```
AjPStr str=NULL;

        str = ajStrNew();

        while(ajFileReadLine(inf,&str))
        {
```

```
        ? process the successive lines ?
    }

    ajFileClose(&inf);
```

16.2 Reading a line from a buffered file

Use the function **ajFileBuffGet**. Its use is the same.

```
AjPStr str=NULL;
AjPFileBuff inf=NULL;

    str = ajStrNew("my.file");
    if(!(inf=ajFileBuffNew(str)))
        ajFatal("Cannot open file %S",str);

    while(ajFileBuffGet(inf,&str))
    {
        ? process the successive lines ?
    }


    ajFileBuffDel(&inf);
```

16.3 Reading in binary

Use a normal file object and the function **ajFileRead**, it's the equivalent of the C fread function.

17. SEQUENCES

You might expect this to be a very long section but you'd be wrong. Although sequences are the bread and butter of the molecular biologist most of the sequence functions in the library are concerned with reading them in transparently to the user. Once you've got the sequence it is effectively a string with a name, an accession number etc. Most of the work done by any application will first get the sequence into an AjPStr object and work on it there. The datatype is the **AjPSeq**. It does have a constructor called **ajSeqNew** but you will hardly ever need to use it unless you want to construct a sequence from scratch. Your sequences will mainly come from ACD and will therefore have been constructed for you.

17.1 Getting multiple sequences

You have seen that to get a single sequence you define a line in ACD like:

sequence: sequence [param: Y type: protein]

and then retrieve it with **ajAcdGetSeq** and the **AjPSeq** datatype. We haven't yet seen how to get multiple sequences. This is done with a line in ACD such as the following:

seqall: sequences [param:Y type: protein]

Then, in the code you use the **AjPSeqall** datatype. Here is an example:

```
AjPSeqall seqs=NULL;
AjPSeq    seq = NULL;

          seqs = ajAcdGetSeqall("sequences");

          while(ajSeqallNext(seqs, &seq))
          {
                    ? do something with the sequence seq
          }
```

The **ajSeqAllNext** function loads the next sequence in the AjPSeq object each time around the loop. If you want to associate sequences as a set then use **seqset** in ACD and use the **ajSeqSet** family of functions. You'd probably do this for multiple alignments.

17.2 Getting information from a sequence

· **ajSeqGetName** get the name. This is a pointer to the internal AjPStr

· **ajSeqName** get the name. This is a pointer to the internal char*

- **ajSeqGetDesc** get the description. This is a pointer to the internal AjPStr
- **ajSeqGetAcc** get the accession number. This is a pointer to the internal AjPStr

17.3 Sequence begin and end points

If you specified a single sequence in ACD then use **ajSeqBegin** and **ajSeqEnd**. If you've used seqall in ACD then use **ajSeqallBegin** and **ajSeqallEnd**, or if seqset has been used then use **ajSeqsetBegin** and **ajSeqsetEnd**.

17.4 Getting a string object copy of a sequence

```
str = ajSeqStrCopy(seq);
```

17.5 Reversing and complementing

- **ajSeqReverse** reverse and complement a sequence
- **ajSeqCompOnly** just complement a sequence
- **ajSeqRevOnly** just reverse a sequence

Note that you can reverse complement a STRING object using **ajSeqReverseStr** or just complement it using **ajSeqCompOnlyStr**. Simple reversal would be achieved by **ajStrRev**. With the first two of these functions be sure to pass a nucleic acid sequence string, there can be no checking.

17.6 Upper and Lower Case

- **ajSeqToUpper**
- **ajSeqToLower**

17.7 Sequence output

The sequence output datatype is the **AjPSeqout**. This datatype is mainly used after manual construction of a (set of) sequence object(s). This is rare. More frequently the string objects are used for output.

18. ACD RETRIEVAL FUNCTIONS

18.1 The Simple Functions

- **ajAcdGetBool**
- **ajAcdGetCodon**
- **ajAcdGetFloat**
- **ajAcdGetInt**
- **ajAcdGetInfile**
- **ajAcdGetOutfile**
- **ajAcdGetSeq**
- **ajAcdGetSeqall**
- **ajAcdGetSeqset**
- **ajAcdGetSeqout**
- **ajAcdGetSeqoutall**
- **ajAcdGetSeqoutset**
- **ajAcdGetString**

18.1 List and Select

The **ajAcdGetList** and **ajAcdGetSelect** calls return an array of string objects. The value of the last element is a NULL pointer.

```
AjPStr *array=NULL;

        array = ajAcdGetList("list");
```

19. REGULAR EXPRESSIONS

The regular expression routines used at the lowest level in EMBOSS are the Henry Spencer ones. These appear in the ajax directory as the hs* files. An AJAX interface has been made for them. The regular expression datatype is the **AjPRegexp**. A regular expression must first be compiled before being used to scan a target string object. Compilation is done using the **ajRegComp** or **ajRegCompC** functions, the scanning by the **ajRegExec** function. The compilation functions are constructors. The destructor for regular expressions is the **ajRegFree** function.

After a string has been scanned then any (sub)matches can be returned using the **ajRegSubI** function and the remainder of the string that did not match by the **ajRegPost** function.

Here is an example:

```
AjPRegexp exp=NULL;
AjPStr str=NULL;
AjPStr id=NULL;
AjPStr remain=NULL;

        str = ajStrNewC(">KABAT wibble This is some more text");

        id = ajStrNew();
        remain = ajStrNew();

        exp = ajRegCompC("^>[A-Za-z0-9_-]+[ \t]+([A-Za-z0-9_-]+)"); /* note the space before \t */

        if(!ajRegExec(exp,str))
                ajFatal("No match found");

        ajRegSubI(exp,1,&id);
        ajRegPost(exp,&remain);

        ajRegFree(&exp);
```

A regular expression is compiled by ajRegCompC. The compiled expression *exp* is used to scan the string *str*. A match is found and the first substring (the bracketed "()" bit of the regular expression) is returned into the id string by ajRegSubI so the id string object now contains "wibble". After the ajRegPost function the remain string object contains "This is some more text". The regular expression is destructed by ajRegFree.

20. MEMORY ALLOCATION

Here are some useful memory allocation macros:

- **AJALLOC(nbytes)** equivalent of malloc
- **AJALLOC0(nbytes)** a calloc of nbytes
- **AJCALLOC(count,nbytes)** a malloc of count lots of nbytes
- **AJCALLOC0(count,nbytes)** a calloc of count lots of nbytes
- **AJNEW(p)** a pointer to an object gets an object allocated for it using malloc
- **AJNEW0(p)** a pointer to an object gets an object allocated for it using calloc
- **AJCNEW(p,c)** a pointer to an object gets c objects allocated using malloc
- **AJCNEW0(p,c)** a pointer to an object gets c objects allocated using calloc

For non-C programmers "malloc" allocates memory but the contents are undefined whereas "calloc" allocates memory setting each location to zero.

In some ways a str=ajStrNew() call is almost the same as AJNEW0(str), but not quite, the AJAX library is just a little more clever than that! It’s a close analogy though. The macro is very useful for allocating your own objects.

21. CLOSING REMARKS

I hope this guide is enough for most programmers toget to grips with EMBOSS. If you feel there is something you’d like to see added that is within the scope of the document then please contact me. A separate graphics guide is in the pipeline.