Design and Implementation of a C Standard Library as Foundation for Porting System Software

Felix Grützmacher

October 23, 2003

Abstract

This diploma work deals with the design and implementation of an ANSI C library. The target architecture for this implementation is ECO32, a 32-bit risc processor invented by Prof. Dr. Hellwig Geisse.

This document describes the design and implementation of the library, documenting the behavior and describing the implementation of library functions, especially where either the implementation is interesting in itself or where it deviates from the C standard. The final chapters briefly describe the design of a shell I implemented in the process of testing the library.

Contents

Chapter 1

Introduction

The ANSI C standard defines more than just the language of C with its lexical tokens, grammar, and semantics. In addition, it defines the standard library, a collection of functions allowing application programmers to exploit the services provided by the operating system in a portable way.

The standard library fulfills two major tasks: First, it ensures portability on the source code level by encapsulating system calls in standardized functions, and second, it provides utility functions which most any application is likely to need, such as functions for processing C strings or basic mathematical functions. System programs such as ls or sh should always prefer library functions over system calls to be more easily portable to other platforms.

Many functions which were present in the early versions of Unix, such as malloc(), printf() and strcmp(), have been included into the standard library of ANSI C. In some cases, such as most of the string functions, porting those functions is as easy as changing the syntax of the function definition to ANSI C and writing the appropriate function prototype (this process can be automated). In other cases, subtle differences between traditional C and ANSI C must be taken into consideration.

Since a standard library is linked into almost any application, efficiency must usually be preferred over readability, leading to a compressed, cryptic coding style. So, rather than just use some reference implementation and be done with it, I have chosen to implement most of the standard library myself to gain an understanding of what goes on inside the more complex parts, such as the i/o system and the storage allocator.

The following chapters describe my implementation of an ANSI C compliant standard library and illustrate the major algorithms, techniques, and design decisions. While documenting many library functions for easy reference and to point out certain specifics of this implementation, this document should not be regarded as a replacement for the ANSI C standard.

Chapter 2

The Transition Between C Dialects

2.1 Function prototypes

2.1.1 The motivation for change

A function prototype, also commonly referred to as a forward declaration, serves the purpose of telling the compiler and the programmer how a function should be used. For instance, consider the following function declaration, which is in traditional style C:

```
double sqrt();
```

This carries two pieces of information: First, that there is a function called `sqrt`, and second, that this function will return a value of type `double`. The declaration does not tell what type—and even what number—of arguments the function expects. From the function's name a caller might infer that `sqrt` expects one argument of type `double`, but without either documentation or the source code of `sqrt` one cannot be sure.

An old-style prototype primarily consists of the following:

- A return type,
- the function's name,
- and an empty pair of parentheses.

If a function's return type happens to be `int`, then the prototype can be omitted altogether. In any other case the prototype is mandatory.

Note also that since the `void` keyword was not part of traditional C, it was impossible to define a function without a return value. If a function contained no return statement, or only return statements without return values, then the return value of that function would be undefined, but nevertheless present. So even by looking at the head of the function definition, it was impossible to deduce whether or not a function returned a meaningful value.

Certain Unix system programs, such as the Borne Shell, include a header file containing a line to the effect of:

```
#define void int
```

This made `void` mean the same as `int` to the compiler by means of macro substitution, and even though a void function still had a return value, at least the programmer could indicate its insignificance.

As an aside, the preprocessor was sometimes also used to approximate the syntax of other programming languages, particularly Pascal. With preprocessor definitions such as

```
#define if if(
#define then )
#define begin {
#define end }
```

it became possible to write an `if` statement as:

```
if a==b
then begin
  c=b;
  d=a;
end
```

You can imagine that I was, shall we say, slightly confused when I looked for the first time at the source code of the Borne Shell and encountered code like that. This is an example how the preprocessor can be used to perpetuate old habits, all be it at the cost of introducing new bugs.

In short, in traditional C it is impossible to perform any sort of type checking on the arguments to functions, unless the function is defined in the same compilation unit as the function call. This has the obvious disadvantages of, for instance, a float argument being passed to a function expecting an int, but there are more subtle implications as well.

Consider the following function definition:

```
char rot13(c)
char c;
{
  if(c>='a' && c<='z') {
    if(c<'m') c+=13; else c-=13;
  }
  else if(c>='A' && c<='Z') {
    if(c<'M') c+=13; else c-=13;
  }
  return c;
}
```

This function returns a `char` value, so if we want to call it from within another compilation unit, we will have to write a prototype:

```
char rot13();
```

Now, if you coded the function call `rot13('A')`, you would expect that it yields the value `'N'`, and on some machines this will indeed be the case, but on others the function would probably return `'\0'`.

This is because `rot13('A')`, apart from being a function call, is also an expression. The operator is `()` (function call), the operands are `rot13` and `'A'`. Since these operands are obviously of different types, the compiler will generate code for type conversion. In particular, the value `'A'`, a char value, will undergo integer promotion. If we assume the character set of our machine to be ASCII, the value which is actually passed to `rot13` is the int value 65. Thus, though the source code does not contain any apparent mistakes, nevertheless a function is passed a value of a different data type than it expects.

On a little endian machine, it would probably make no difference because the relevant byte would end up where the function expects it (on the stack just above the return address). So if I tested my function on an Intel machine I would never notice the bug. But if I compiled and executed the same code on a big endian system, `rot13` might suddenly fail without apparent reason. A variant of this actually happened to me while trying to port a library function. Such pitfalls provided the motivation for me to decide on implementing at least part of an ANSI C standard library instead of simply trying to port the Unix subroutines.

2.1.2 ANSI C prototypes

With ANSI C, a new form of function prototype was introduced, one in which the pair of parentheses need not be empty, but may instead contain a list of the types of arguments that the function expects. The function `rot13`, given in the previous section to illustrate one of the more subtle pitfalls of the lack of type checking, could be forward-declared in ANSI C as follows:

```
char rot13(char c);
```

The identifier for the formal parameter can even be omitted:

```
char rot13(char);
```

This eliminates the problem described in the previous section, because the compiler now knows that the function expects a char value as its argument.

2.2 More differences

2.2.1 The signed qualifier

In traditional C, the "signed" type qualifier was usually not available because it was considered that all integral types would be signed by default. The exception to the rule was the "char" data type which was signed in some implementations and unsigned in others. To safeguard against the problems related to unsigned-preserving integer promotion (see previous section), chars were often cast to int in complex expressions.

2.2.2 const

The "const" qualifier was not available in traditional C, leading to excessive use of preprocessor macros to achieve the effect of a named constant value. As any C programmer will find, preprocessor macros are error-prone because a

great deal of care must be taken concerning the correct use of parentheses. The rule of thumb—which I have followed most of the time while implementing the library—is that an additional pair of parentheses should always be used when in doubt.

2.2.3 Initialization of automatic arrays

In traditional C, it was not possible to directly initialize an automatic array.[1] For instance, the following code snippet was not allowed:

```
f() {
  int a[] = {2, 3, 5, 7};
}
```

To avoid manual assignment to the elements of an automatic array, quite a few programmers made it common practice to use a global array instead, for the only reason that this could be initialized. In this way, variables became global to the entire program although their use was often restricted to just one function. On some implementations it was sufficient to define the array as static. This, however, resulted in an unnecessary increase in the size of the application's data segment.

2.2.4 enum

In traditional C, the availability of enumeration types varied between implementations, although its increasing popularity finally caused it to be included in the ANSI C standard. On those implementations where it was present, the enum keyword was used mainly as a means of creating constants; on other implementations one had to fall back on preprocessor macros—with all their inherent disadvantages.

[1] An automatic variable is one which is defined local to a function, and which is also not static.

Chapter 3

Error handling

3.1 Detecting errors

Error handling in ANSI C is based on two mechanisms: The return values of library functions, and the global variable *errno* which is of type int and is declared in the header file errno.h.

When a C library function detects that it is unable to carry out its task, it sets the errno variable to some value describing the error condition, and it returns a value indicating an error (for instance, most of the I/O functions indicate an error by returning the constant EOF). On the other hand, a function which succeeds does not reset errno to zero, so a non-zero value of errno does not indicate that the last library function call was the one which produced the error. While reading ANSI C source code, I often encounter the idiom of setting errno to zero, then calling a library function, then checking whether errno is still zero. While this works as one would expect, it is more efficient to check the return value, and access errno only when necessary.

3.2 Retrieving more information about an error

More information about the nature of an error can be retrieved by comparing the value of errno to the valid error constants. These constants are defined as preprocessor macros in errno.h, and are distinguished by the fact that their names start with the letter "E".

The current implementation defines 34 error constants. The first 33 of them are all identical to the ones returned by Unix system calls, and their documentation may be looked up in the Unix Programmer's Manual. The library introduces an additional constant ERANGE which indicates that an argument or return value is outside the valid range. For instance, the function strtol() converts the string representation of a number to a value of type "long". This function sets errno to ERANGE if the number contained in the string is too large or too small to be expressed as a "long" value.

All error constants have error messages associated with them. To retrieve a pointer to the error message for a given error constant, the function *strerror* (defined in string.h) may be used. The current implementation returns a pointer to the string "Unknown error code" when passed an argument which is not one

of the error constants. The exception is the value zero which results in the string "No error".

The library provides a convenient shortcut for printing an error message to the "stderr" stream. In the header stdio.h, the function *perror* is defined so that

```
perror(str);
```

is exactly equivalent to:

```
fprintf(stderr, "%s: %s\n", str, strerror(errno));
```

The functions setjmp() and longjmp(), described in chapter 8, make it possible to approximate more sophisticated error handling methods, such as the "exception-throwing" mechanism of C++. In fact, the first C++ compilers were basically ANSI C compilers with an additional, quite complex preprocessor pass which transformed a C++ program into an equivalent ANSI C program which was then compiled as usual. Where the C++ code made use of exceptions, the resulting ANSI C code would usually call setjmp() and longjmp() to achieve the same behaviour.

3.3 Implementation

The variable errno is implemented as a global variable exported by the assembly module "syscall.s" which encapsulates all Unix system calls. The error constants are defined, as preprocessor macros, in errno.h. When adding more error codes, be sure to increase the preprocessor constant `_MAX_ERRNO` so that it yields the value of the greatest error constant.

Finally, the file errno.c defines the array `_ERRMSG` which contains the corresponding error message for each error constant.

3.4 Deviations from ANSI C

The current implementation of the error handling mechanism complies fully with the C standard.

Chapter 4

Standard Input/Output

4.1 Usage

4.1.1 Overview

All but the most trivial applications will need to interact with their environment by means of input/output operations. Even the "Hello World" applications of introductory programming classes need to rely on an underlying layer of code for carrying out the actual output. This chapter deals with the usage and implementation of the input/output part of the C standard library.

The data types, functions, and variables for the I/O functionality are declared in the header *stdio.h*. Unless otherwise noted, the functions described in this chapter are declared there, and their implementation can be found in the file *stdio.c* in the lib/source directory.

The functionality of stdio.h revolves around a simple concept called a stream, which may be defined as a source or sink of data. In the C standard library, the atomic unit of data is a character (which takes up 8 bits on most architectures, including ECO32). A stream might be attached to a terminal, a file, a pipe, a printer or a network connection, but the C programmer may use the same set of I/O functions for all these different stream variants, i.e. streams are treated in a uniform way regardless of their physical implementations.

This design goal leads to the fact that the functionality of stdio.h is undoubtedly the most complex part of the C standard library. Its implementation takes up approximately one third of the entire library code, and at the time of writing comprises about 1200 lines of code (including inline documentation and blank lines). It is also harder to debug than the rest of the library because one cannot rely on stdio.h for producing debug messages.

In the implementation section of this chapter, I have chosen to adopt a fairly low-level approach, including many technical details. The aim of this is not to emphasize their importance but to illustrate certain programming techniques, and also to point out certain pitfalls of the C language (more on this in chapter 2).

4.1.2 Streams

Representing Streams

Streams in C are represented by objects of data type *FILE* which in most implementations (including mine) is defined as a struct. The actual members of this structure are irrelevant to users because most of the time they will be operating on pointers to FILE structures rather than on the structures themselves. Most functions declared in stdio.h thus expect an argument of type *FILE **.

Pre-defined Streams

In addition to any streams explicitly opened by the application, the C standard library defines the following three streams:

- stdin (input): The standard input stream. Most of the time, when a shell is used to launch a program, the control terminal of the shell also becomes that program's input stream unless the user chooses to redirect it to a file or pipe. Filter programs such as *sort* or *wc* should always request their input from stdin.
- stdout (output): The standard output stream (analogous to stdin but for output).
- stderr: The standard error stream. To prevent error messages from becoming mixed up with the regular output of the program, a separate output stream is maintained for them, so that, for instance, the regular output may be redirected to a file while error messages still go to the user's terminal. This stream should be used for debugging and error messages only.

4.1.3 Character-level I/O

To read or write a single character from or to a stream, an application uses the functions *getc* and *putc*. If the operation was successful, both of these functions return the character which was written or read, otherwise they return *EOF* (a special constant defined in stdio.h, which is guaranteed not to compare equal to any character).

When getc() returns EOF, this is because of one of two reasons:

1. The end of the input stream was reached, or
2. an error occurred while reading from the input stream, such as the error produced by reading from a damaged disk sector.

The macros *ferror* and *feof* can be used on a stream to determine which of those two was the reason for the return value of EOF. If the reason was an error then the variable errno will have been set accordingly. (My implementation does not assign any values to errno but merely retains the ones assigned by the Unix kernel.)

A stream's error flag may be cleared with the *clrerr* macro, so that the next call to ferror for that stream will return zero. After an application has

attempted to correct an I/O error, it should use clrerr to clear the error flag, then retry the operation which produced the error.

Since most of the time characters will be read from stdin or written to stdout, the library defines the macro getchar() to be equivalent to getc(stdin), and the macro putchar(c) to be equivalent to putc(c, stdout).

The function "ungetc" exists to put a character that has been read into the stream once more, so that the next reading operation performed on that stream will return that character.

This is a feature which is used by most lexical scanners. Consider, for instance, the following short function for reading a number from stdin:

```
int number(void) {
  int c;
  int res=0;
  while((c=getc(stdin))>=0 && c<=9)
    res = res*10 +c-'0';
  if(c!=EOF)
    ungetc(c, stdin);
  return res;
}
```

This function only knows where the number ends if it has read one character "too much". To make that additional character available to subsequent reading operations on that stream, the function calls "ungetc" to put it back.

The ANSI standard states that it must be possible to "ungetc" at least one character. Putting more than one character back is optional, and applications should not rely on it. My implementation always allows for at least one character, even when the stream is unbuffered.

4.1.4 String-level I/O

The functions *fgets* and *fputs* are provided for I/O operations on zero-terminated C strings. (In this and the following chapters, unless otherwise noted, the word "string" will always refer to a zero-terminated C string.)

The fputs() function writes a string to a stream, not including its termination character. Contrary to puts (see below), this function does not append a newline character to the string.

The fgets() function reads a maximum of $n - 1$ characters from a stream, stopping when it encounters a newline character. In this case, the newline will be placed into the buffer. (By n is meant the second argument to fgets.)

There are two additional functions, puts() and gets(), which are different from fputs() and fgets() in the following ways:

1. They operate only on the two streams stdin and stdout.
2. puts() appends a newline character at the end of the string.
3. gets() does not allow for the specification of the size of the receiving buffer. The consequence of this is that code using gets() is always unsafe because of potential buffer overruns. In other words, gets() should *not* be used, and its implementation in the ECO32 library exists only for the sake of completeness.

puts() and fputs() return EOF on error, any other value on successful completion. gets() and fgets() return the NULL pointer on error or end of stream, otherwise they return a pointer to the receiving buffer.

For reading or writing a block of memory which is not a C string, the functions *fread* and *fwrite* are provided. fread() does not stop reading upon encountering a newline character, and fwrite() continues writing even when encountering a zero byte. Both functions return the number of objects successfully read or written, an object being defined as a block whose size, in bytes, is given by the *size* argument.

4.1.5 Formatted I/O

This section documents the *printf* and *scanf* families of functions. These provide the functionality to scan a stream for formatted input and to print formatted data such as rows of tables.

scanf, fscanf, sscanf

All of these functions take at least one argument which is a string called the format specifier. The arguments following the format specifier are pointers through which the scanned values are stored. The scanf functions return the number of objects successfully scanned, or EOF if an error occurred.

The *scanf* function reads its input from stdin, the *fscanf* function reads from the stream given as the first argument, and the *sscanf* function reads from a C string.

The rest of this section documents the structure of the format specifier understood by the ECO32 library. This is only a subset of the syntax specified by ANSI C because the ECO32 architecture, at the time of writing, does not have a convenient representation for floating point numbers.

The specifier, or format string, consists of whitespace characters, conversion specifiers, and other characters.

A whitespace character means that scanf (or the function in question) will skip any sequence of whitespace characters in the input stream.

Any other character which is not a conversion specifier means that the next character from the input stream must match this one exactly. If the character in the format string and the next character from the input stream do not compare equal, scanf stops at this point.

A conversion specifier begins with a percent character (%) and then contains, in that order:

1. An optional asterisc (*). If present, it tells scanf not to store the result of the conversion through the next pointer argument.

2. An optional decimal integer which gives the field width, i.e. the maximum number of characters from the input stream that should be used for the conversion. Any leading whitespace in the stream does not count towards that maximum.

3. An optional “h” or “l”. The “l” flag must be specified if and only if the corresponding pointer argument points to a “long int”, and the “h” flag must be specified if and only if the corresponding pointer argument points to a “short int”. It is an error to include both.

4. The type specifier, which is not optional. The following specifiers are recognized:

Specifier	Type pointed to by next pointer argument
c, s, [...]	char
d, i, o, u, x, X	int, short, or long
p	pointer
n	int

The specifiers d, i, o, u, x and X are used for scanning integers. The d specifier scans for a signed decimal, o, u, and x scan for unsigned octal, decimal and hexadecimal, respectively. X also scans for hexadecimal but expects the alphabetic digits ("A"-"F") to be capitalized.

The p specifier scans for a pointer value. In my implementation it is equivalent to x.

The c and s specifiers both scan for a sequence of characters. c scans for exactly *width* characters, defaulting to one character if no width was given. s scans for a maximum of *width* characters, and if no width was given it is assumed to be infinite. Contrary to c, s stops upon encountering a whitespace character, and it also null-terminates the result, making it a C string.

The n specifier does not read any characters from the input stream. It stores, through the next pointer argument, the number of successful conversions scanf has performed so far. In my implementation, the n specifier itself counts as one conversion (although nothing is really converted).

A special case of the format specifier is the sequence %%, meaning the next character from the input stream must be a percent sign. If it isn't, then scanf aborts.

printf, fprintf, sprintf

All of these functions expect at least one argument, which is a format string similar to that expected by the scanf functions. The remaining arguments following the format string contain the data to convert.

The format string is comprised of simple characters and conversion specifiers. A simple character, when encountered, is simply printed as-is. A conversion specifier instructs printf to convert and print its next argument. A conversion specifier is composed of the following, in that order:

1. A percent sign
2. An optional sequence of the following flags:
   - "#": Instructs printf to prepend 0x or 0X to a hexadecimal, and 0 to an octal integer.
   - "0": Left-pad with zeros instead of spaces.
   - "+": Print a plus sign for a positive integers.
   - "-": The output is left-adjusted. This flag overrides the "0" flag described above.

- " " (blank): Use a blank as sign for positive integers. This is useful to ensure that in a table column, the least significant digits of numbers are properly aligned.

3. An optional decimal integer which gives the field width, i.e. the minimum number of characters printed by this conversion. Instead of a decimal an asterisc (*) can be given, meaning that printf will take its next argument value as the field width. If this is negative, then its absolute value is used as field width, and the behavior of printf changes as if the "-" flag had been specified.

4. An optional precision value, preceded by a dot, giving the number of significant characters printed by this conversion (e.g. the number of digits). The precision field may be a decimal integer, an asterisc (with the same semantics as in the field width), or nothing at all (in which case a precision of 0 is assumed).

5. An optional "l" or "h" flag, with semantics analogous to those of the same flags in the scanf format string.

6. The type specifier, which is not optional. The following type specifiers are understood:

   - "%": A percent character is printed, and no argument is used.
   - "d": The argument is printed as a decimal integer.
   - "i": Exactly equivalent to "d".
   - "o": The argument is printed as an octal integer.
   - "u": The argument is printed as an unsigned decimal integer.
   - "x": The argument is printed as an unsigned hexadecimal integer, using lower case letters a-f.
   - "X": The same as "x" but using upper case letters A-F.
   - "c": The argument, which is assumed to be of type int, is printed as a character.
   - "s": Prints out the C string pointed to by the argument.

The printf functions return the number of bytes printed, or EOF if an error occurred.

4.1.6 File Handling

Opening a File

The *fopen* function is provided for opening a file. If successful, it returns a pointer to the newly created stream, otherwise it returns the NULL pointer.

fopen() takes two arguments, the first of which is the filename and the second of which is a mode specifier. The list below gives all the mode specifiers understood by the ECO32 library, along with their respective semantics:

- r, rb, rt: The file is opened for reading only, and the position in the file will be set to the beginning.

- w, wb, wt: The file is opened for writing only, and the position in the file will be set to the beginning. If the file already exists, it will be truncated to zero length.
- r+, r+b, r+t, rb+, rt+: The file will be opened for reading and writing, and the position in the file will be set to the beginning.
- w+, w+b, w+t, wb+, wt+: The file will be opened for reading and writing, and the position in the file will be set to the beginning. If the file already exists, it will be truncated to zero length.
- a, ab, at: The file is opened for appending at the end.
- a, a+, a+b, a+t, ab+, at+: The file is opened for appending at the end, and also for reading.

Closing a file

The *fclose* function closes a file stream. If a stream is not closed explicitly by means of this function, it will be closed at program termination (see chapter 12).

Changing the file position

The function *fseekfseek* may be used to set the position in a file where the next I/O operation will occur. fseek() takes three arguments: The file stream, the offset, and one of three pre-defined constants telling fseek() how the offset is to be interpreted. A value of `seek_set` means the offset is relative to the beginning, a value of `seek_cur` means it is relative to the current position, and a value of `seek_end` means it is relative to the end of the file.

fseek() returns the new position in the file, relative to the beginning, or EOF if an error occurred.

Temporary files

A temporary file, to the C standard library, is a file opened for reading and writing which is automatically deleted when closed. The *tmpfile* function returns a pointer to such a temporary file stream, or NULL if the file could not be opened.

To obtain a unique name for the temporary file, tmpfile() makes a call to the tmpnam() function which returns such a filename. tmpnam() may be called directly, although it usually only makes sense in the context of tmpfile().

4.2 Implementation

This section gives some implementation details for the functionality provided by stdio.h.

4.2.1 Buffering basics

Besides portability, a major advantage of the stdio functions over their analogous system calls is the fact that the stream functions are buffered.

Every stream can have a buffer associated with it. When reading from such a buffered stream for the first time, the buffer is filled with characters from the actual underlying file descriptor. Subsequent calls to reading functions will return characters from that buffer, or refill the buffer in case it runs out of characters.

Similarly, when writing to a buffered stream, the characters are not directly written to the underlying file descriptor, but stored in that stream's buffer. Only if this buffer is full will its contents be written to the underlying file descriptor.

The purpose of this approach is, of course, to save on expensive system calls and even more expensive disk access. The price to pay is a marked increase of complexity in the implementation, and also the fact that the file on disk may not always be consistent with the abstract stream. So if two processes share the same file, changes originating from one process may not immediately be visible to the other.

By default, the library will always try to allocate a buffer for a newly created stream, and only if the system runs out of memory will the library resort to unbuffered I/O. However, there are two exceptions to this general rule:

1. stderr is never buffered.

2. stdout is unbuffered if it is connected to a terminal. To determine whether or not this is the case, the library issues the system call "sgtty" for the file descriptor of stdout. This system call is only valid for terminals, so if it fails, this would indicate that stdout is connected to something other than a terminal, e.g. a file or pipe.

4.2.2 The FILE structure

The following is the definition of the FILE data type, taken directly from stdio.h:

```
/* The FILE structure */
typedef struct {
  int fd;
  char *buf;
  char *pos;
  int cnt;
  int flags;
} FILE;
```

This structure is fairly self-explanatory:

"fd" is the file descriptor as returned by the "open" system call, and is used for subsequent operations on that file.

"buf" points to the I/O buffer for that stream. Note that the same buffer is used for both reading and writing. This presents no problem because a read operation must not be followed by a write operation without an intervening call to one of the functions "fflush" or "fseek". It is the responsibility of these functions to tidy up the buffer.

"pos" points to the current position in the buffer. If this buffer is currently used for writing, this is the address where the next character is placed by one of the write functions. If it is used for reading, it points to the next character which a read operation will return.

The meaning of "cnt" also depends on whether the buffer is currently used for reading or writing. When reading, cnt specifies the number of characters remaining in the buffer. When writing, it specifies the number of remaining free slots. This value is used by "getc" and "putc" to determine if the buffer is empty or full, respectively.

Finally, "flags" is a bitmask of several boolean flags giving additional information about the stream. Some of these flags are:

- `_READ`: The stream is open for reading
- `_WRITE`: The stream is open for writing
- `_READ_WRITE`: The stream is open for both reading and writing. Additionally, the `_READ` or `_WRITE` flag may be set to indicate the state of the buffer.
- `_EOF`: The application has tried to read beyond the end of the file
- `_ERROR`: An error occurred while reading from or writing to the stream. This is most likely some physical I/O error encountered by the operating system. The variable errno will have been set accordingly.
- `_UNBUF`: This stream is unbuffered, i.e. any reading or writing operation on that stream will result in an immediate system call. An example of an unbuffered stream is standard output when it's going to a terminal, as would be the case in an interactive application. If this were buffered, then not all the output from an application would immediately be visible to the user. One example of a stream which is never buffered is stderr. (I made this decision based on the assumption that error messages would always be of immediate interest to the user, even when they are redirected to a file.)

All the FILE structures are kept in the `_files` array, which is defined as follows:

```
FILE _files[_MAX_FILES] = {
  {0, _stdinbuf, _stdinbuf, 0, _READ},
  {1, NULL, NULL, 0, _WRITE},
  {2, NULL, NULL, 0, _UNBUF|_WRITE}
};
```

With the `_files` array set up in this way, it is now possible to define the three standard streams:

```
#define stdin (&_files[0])
#define stdout (&_files[1])
#define stderr (&_files[2])
```

4.2.3 getc, putc, ungetc

At the core of buffered I/O, we have the two functions `getc` and `putc` for reading or writing a single character. The idea is that if we have functions for buffered I/O of single characters, all the other I/O functions can be implemented based on those.

I have implemented both `getc` and `putc` as preprocessor macros in `stdio.h`. If `getc` is performed on a stream with a non-empty buffer, or if `putc` is performed on a stream with a buffer that is not full, then the action is carried out by the macro alone. Otherwise, `getc` and `putc` will delegate their calls to the helper functions `_bufread` and `_bufwrite`, respectively. These functions will refill, flush, or allocate the buffer as needed. This would have been tedious to implement as preprocessor macro.

There are also the two functions `fputc` and `fgetc` available, which are semantically equivalent to `putc` and `getc` but are implemented as functions instead of macros. This makes no difference most of the time, except if their arguments have side effects. To illustrate, consider the following two definitions:

```
/* First, a function */
int square(int x) {
  return x*x;
}
/* and second, a preprocessor macro */
#define SQUARE(x) ((x)*(x))
```

The expressions `square(y)` and `SQUARE(y)` would seem to be equivalent, but aren't. For instance, if `v` is an integer variable with the value of 5, then the expression `square(v++)` yields 25 and leaves `v` at 6, whereas the expression `SQUARE(v++)`, interestingly, yields 30 and leaves `v` at 7. This is because the preprocessor substitutes `SQUARE(v)` by:

```
((v++)*(v++))
```

So, if `fp` were a pointer into an array of streams, then `fgetc(fp++)` would work as expected, but `getc(fp++)` would fail miserably with my implementation of stdio.h. This is ANSI C compliant.

For more on such intricacies, see chapter 2 on page 5.

The ungetc() function, which makes it possible to re-insert a character into an input stream, is also implemented as a preprocessor macro. It simply places the character into that stream's buffer[1].

4.2.4 High level I/O

All of the remaining I/O functions, such as printf and scanf, fgets and fputs, etc. perform their I/O operations by means of the three low-level I/O functions, getc, putc and ungetc. In light of this it becomes obvious why the latter three had to be implemented as preprocessor macros rather than functions: This way the stdio library does not become cluttered with lots of inefficient function calls.

[1] Note that this works even for unbuffered streams because, internally, they have a buffer of size 1.

4.2.5 Temporary files

This section deals with the implementation of the functions tmpnam() and tmpfile().

The tmpnam() function creates a name for a temporary file, i.e. the name must be unique on a system-wide basis, and it must easily be recognizable as a temporary filename.

In this implementation, the name of a temporary file consists of the following:

1. The process id (pid) of the process creating the name,
2. A serial number between 00000 and 99999, and
3. the suffix ".tmp".

So a process may have 100,000 temporary files opened simultaneously before this implementation will run into trouble.

The tmpfile() function does little more than obtain a filename (by calling tmpnam), open that file (with a mode argument of "w+b"), then unlink it from the file system. This last step makes sure the file will be automatically deleted by the operating system once the application closes it. So theoretically, only if the system crashes will temporary files need to be cleaned up manually.

4.2.6 Conclusion

I conclude this section on the implementation of stdio, in the hope that it will be helpful in reading and extending its source code more easily.

The source code includes extensive inline documentation explaining those implementation details which might appear cryptic at first glance. In those cases where the information in this document clashes with that in the inline documentation, the latter should be regarded as superceding the former.

4.3 Deviations from the ANSI C standard

At the time of writing, my implementation of stdio.h still lacks the function setvbuf(). The remaining functions have demonstrated ANSI-compliant behavior during testing. Note that scanf() and printf() (and their relatives) do not include the functionality for printing/scanning floating point numbers.

Chapter 5

String and memory functions

This chapter documents the functions provided by the header string.h, giving implementation details where the implementation is interesting enough to justify it.

5.1 String functions

The functions declared in string.h mostly fall into two distinct categories: Those whose names begin with "str" are string functions; they operate on C strings, i.e. zero-terminated character arrays. Those functions whose names start with "mem" are memory functions; they operate on blocks of memory, regardless of their structure.

The following string functions are available:

- `strcpy(dst, src)`: Copies a string from src to dst, stopping at the termination character which is also copied. The function returns dst. The memory regions for source and destination should not overlap.
- `strncpy(dst, src, n)`: Copies at most `n` characters from src to dst, stopping after a zero character has been copied. At this point, if less than `n` characters have been copied, zero characters are written to `dst` until a total of `n` characters have been written. Note that if the length of src is greater or equal to `n`, dst will not be zero-terminated. strncpy() does not work if the memory regions for src and dst overlap.
- `strcat(dst, src)`: Appends src to dst. This is exactly equivalent to `strcpy(dst,strlen(dst), src)`, but it is more efficient to call strcat().
- `strncat(dst, src, n)`: Appends at most n characters from src to dst, stopping if a zero character has been copied. strncat() makes sure that dst will always be zero-terminated. The function returns dst.
- `strcmp(s1, s2)`: Lexicographically compares the two strings s1 and s2, returning zero if they are equal, a negative value if s1 is less than s2, and a positive value otherwise. If one of the strings is shorter than the other,

and the latter starts with the former, then the latter is considered greater. Note that this function is useful as an argument to qsort() or bsearch(), to sort or search an array of strings.

- `strncmp(s1, s2, n)`: Has the same semantics as strcmp(), but pays attention only to the first n characters of both strings.

- `strchr(s, c)`: Returns a pointer to the first occurrence of the character c in the string s, or a NULL pointer if s does not contain c. If `'\0'` is passed as argument c, a pointer to the end of the string (the termination character) is returned.

- `strrchr(s, c)`: Returns a pointer to the last occurrence of the character c in the string s. The NULL pointer is returned if s does not contain c. If `'\0'` is passed as argument c, the NULL pointer is returned.

- `strspn(s1, s2)`: Returns the length of the prefix of s1 which consists only of characters contained in s2. A common use for this function is to strip whitespace from the beginning of a string. If s2 points to the empty string, zero is returned.

- `strcspn(s1, s2)`: Returns the length of the prefix of s1 which consists only of characters not in s2. A common use is to strip trailing whitespace from a string. If s2 points to the empty string, the length of s1 is returned.

- `strpbrk(s1, s2)`: Returns a pointer to the first occurrence of a character in s1 which is also contained in s2. If none of the characters in s2 are contained in s1, the return value is the NULL pointer. The termination characters are not considered part of the strings.

- `strstr(s1, s2)`: Returns a pointer to the first occurrence of s2 in s1, i.e. performs string searching. If s2 is not a substring of s1, the NULL pointer is returned. A special case is that s1 is returned if s2 points to the empty string.

  This function is implemented as an iteration over s1. If, during this iteration, the first character of s2 is found, then strncmp() is called to determine if a match was found. If so, the current position in s1 is returned, otherwise the loop continues.

- `strlen(s)`: Returns the length of the string s, which is defined as the number of characters before the termination character.

- `strerror(n)`: Returns a pointer to an appropriate error message corresponding to the error code n (see chapter 3 for more details).

- `strtok(s1, s2)`: Tokenizes a string. The string to be tokenized should be passed as argument s1; s2 should contain the separator characters, i.e. the characters which separate tokens in s1. If s1 contains anything other than separator characters, a pointer to the first token is returned. If the NULL pointer is passed as s1 to subsequent calls to strtok(), the function will return a pointer to the next token as long as one is available. When strtok() runs out of tokens, it returns the NULL pointer. The separator string s2 may be different for each call to strtok(), but it may not be

NULL. Note that strtok() "destroys" the string it tokenizes by writing termination characters at the end of each token it finds.

The implementation of strtok() may serve as a good example for the use of strspn() and strcspn().

5.2 Memory functions

These functions provide efficient methods for working with memory blocks, which may or may not be C strings. The zero character which is used to terminate a C string has no special significance to these functions.

- `memcpy(dst, src, n)`: Copies n bytes from src to dst, returning dst. memcpy() will not have the desired effect if the memory regions for source and destination overlap.
- `memmove(dst, src, n)`: This function has the same semantics as memcpy(), but also works for overlapping memory areas.

  The implementation works by comparing the two pointers dst and src. If dst is less than src, copying takes place "from left to right", i.e. in ascending order of addresses. If dst is greater than src, copying is done in reverse order. This way it is ensured that memmove() does not overwrite memory regions it hasn't read yet.
- `memcmp(s1, s2, n)`: Compares two memory regions of n bytes which may overlap. The return value is exactly the same as that of strcmp(), except for the fact that memcmp() does not stop comparing when it encounters a zero character.
- `memchr(s, c, n)`: Searches the memory region defined by s and n for the byte c, returning a pointer to the first occurrence. A NULL pointer is returned if c cannot be found.
- `memset(s, c, n)`: Starting with the byte pointed to by s, memset() sets n contiguous bytes to c, then returns s.

Chapter 6

Sorting an array

6.1 Motivation

This chapter is a singularity in that it concerns itself with just a single function, namely the *qsort* function which employs the Quicksort algorithm to sort the elements of an array. There are two reasons for this somewhat lengthy observation of qsort(): First, the source code might appear cryptic to the uninitiated, and second, it illustrates the special care which must be taken when implementing library functions.

A library routine exists, by definition, to be reused in a wide variety of applications, so its efficiency and correctness, or lack thereof, have a much greater impact than would be the case with any other function. The description of the Quicksort algorithm is deceptively simple, yet it is extremely easy to come up with an implementation which is almost, but not quite, correct. Such an implementation would work as expected in many cases, then suddenly fail when confronted with a special case, such as an array whose last element is the smallest. A naive Quicksort implementation also tends to consume much more memory than is actually necessary.

6.2 The algorithm

The basic Quicksort algorithm was invented in 1960 by C. A. R. Hoare, and it still ranks among the best general-purpose sorting algorithms known today. "General-purpose" means that the algorithm performs well in most cases, although better algorithms exist for certain special cases. For instance, if an array is already sorted except for two adjacent elements which need to be exchanged, even Bubble-sort performs better than certain Quicksort implementations. However, when the nature of the array, particularly the distribution of key values, is not known in advance, Quicksort has been demonstrated to be a reasonable choice.

Like most recursive algorithms, Quicksort is based upon the principle of "divide and conquer", i.e. it solves its problem by considering it as a composition of smaller sub-problems which are more easily handled.

To sort an array `A`, Quicksort performs the following steps:

1. Partition `A` so that the element `A[i]` ends up in its final place. All the elements to the left of `A[i]` are less than or equal to `A[i]`, and all the elements to the right of `A[i]` are greater than or equal to `A[i]`. We call `A[i]` the partitioning element or pivot element.

2. Independently of each other, sort the two sub-arrays thus created.

Admittedly, the above description leaves a lot to be desired, particularly when it comes to the partitioning process in step 1. One common way of partitioning is accomplished by maintaining two pointers, `i` and `j`, so that `i` starts off pointing to the first element, and `j` starts off pointing to the last element of the array. The following steps are then performed:

1. Select a pivot element.

2. Advance `i` to the right until it points to an element which is greater than or equal to the pivot.

3. Advance `j` to the left until it points to an element which is less than or equal to the pivot.

4. If `i<j` still holds, then the two elements pointed to by `i` and `j` are obviously in the wrong order, so exchange them.

5. If the pointers cross, the partitioning process is almost complete. Otherwise, go back to step 2.

6. Finally, exchange the element pointed to by `i` (or, alternatively, `j`) with the partitioning element.

This description leaves us with but one arbitrary factor: the method of choosing a partitioning element.

The following example illustrates the partitioning process for an array of 6 elements. The first element of the array has been arbitrarily chosen as the pivot element.

State	`A[0]`	`A[1]`	`A[2]`	`A[3]`	`A[4]`	`A[5]`	Remarks
1	4	2	5	6	3	1	Pivot is 4
			i			j	
2	4	2	1	6	3	5	
				i	j		
3	4	2	1	3	6	5	
				j	i		The pointers cross
4	3	2	1	4	6	5	Exchanging

With state 4, the goal of the partitioning process has been achieved. `A[3]` is in its final place in the array, all elements to the left of it are less than or equal to it, and all elements to the right of it are greater than or equal to it. The next step would be to sort the two sub-arrays to the left and right of `A[3]`, and the array `A`, in its entirety, would be sorted.

6.3 Implementation

6.3.1 The first attempt

I started out with a naive Quicksort implementation which was well-suited to small arrays but very demanding on memory for larger ones. For purposes of reference, this implementation may be found in the file lib/source/qsort.c.old.

In this implementation, most of the work is done by the function `rec`. It partitions the array using the method described above, choosing the first element of the array as the pivot element. It then calls itself recursively to sort the two sub-arrays created by the partitioning process.

The inefficiency of this implementation becomes apparent when one considers the two recursive calls for the sub-arrays. For every recursive call, the following information is pushed onto the stack:

- The return address
- The size of an array element ("size" argument)
- A pointer to the comparison function ("cmp" argument)
- Two pointers (the local variables i and j)
- Two more pointers (the arguments f and l)

Of the list above, only the last entry contains information relevant to the rest of the sorting process. The following section introduces an implementation of qsort() which drastically reduces the demand on memory by no longer containing recursive calls.

6.3.2 Eliminating recursive calls

The explicit stack

The naive implementation described in the previous section makes implicit use of the program's runtime stack to store information. The disadvantage of this approach is that usually more information is stored than is actually necessary. So instead of using the implicit stack by means of nested function calls, the current implementation of qsort() uses an explicit stack, and saves only the information which is relevant to the problem at hand. This explicit stack is used for "remembering" those sub-arrays which are still in need of sorting.

The idea is to place the previous implementation inside a while loop. Whenever a partitioning process is complete, its resulting sub-arrays are pushed onto the stack. Following this, the next sub-array is simply popped off the stack, and partitioning continues until the stack is empty, i.e. there is no more work to be done.

The stack is implemented by means of an array and a stack pointer. Information is pushed onto the stack by storing it through the stack pointer, then incrementing the pointer. Information is popped off the stack by first decrementing the pointer, then accessing the data it points to.

Removing tail recursion

The use of an explicit stack already eliminated a lot of unnecessary overhead. More overhead can be removed by reducing the amount to which the stack is used.

A function is called tail-recursive if the last thing it does before returning is to call itself. To allocate a new stack frame for such a function call would be a waste of memory, because the caller will return immediately when the new instance has returned. So instead of allocating a new stack frame, it suffices to overwrite the old arguments with the new ones, then jump back to the beginning of the function. So tail-recursion and iteration are one and the same: The idea is to repeat the same operation for different values, but in the same space. As an aside, in a functional programming language such as LISP, the only way of expressing iteration is by means of tail recursion.

The function rec() in the implementation described above ends with such a tail-recursive call. So, instead of pushing the boundaries of both sub-arrays onto the stack, the current implementation saves just the boundaries of one sub-array; the second sub-array is stored directly in the two variables f and l, so it will be processed immediately in the next iteration.

6.4 Usage

The qsort() function expects the following arguments:

- base: A pointer to the start of the array to sort
- n: The number of elements in the array
- size: The size, in bytes, of one element. qsort() has no other way of determining where an element ends and the next one starts.
- cmp: A pointer to a function which expects two arguments of type `const void *`. qsort() uses this function to compare the key values of two elements. The function should return zero if they are equal, a negative value if the first argument is considered less than the second one, and a positive value if the first argument is considered greater than the second.

A good indication of whether or not the algorithm performs well for some array can be obtained by counting the number of comparisons, i.e. the number of calls to the function passed as cmp. An easy way to accomplish this would be to implement this function so that it increments some global variable each time it is called. There are degenerate cases where the number of comparisons is proportional to n^2 [3] (this may happen when the array in question is almost sorted). If it is known in advance that such a case is to be expected, my advice would be to use an implementation of Shellsort instead.

6.5 Searching an array

The library function *bsearch* performs a binary search on an array. It expects the following arguments:

- key: A pointer to the key to search for
- base: A pointer to the start of the array to search
- n: The number of elements in the array
- size: The size of such an element (note that the key does not need to be of the same size)
- cmp: The comparison function; it expects two arguments of type `const void *`, the first one pointing to the key, the second pointing to an element in the array. The expected return values are the same as with the cmp argument to qsort().

bsearch() returns a pointer to the element if it could be found, otherwise it returns the NULL pointer.

For completeness: Note that the binary search algorithm expects the array to be sorted with respect to the comparison function.

Chapter 7

The random number generator

7.1 Usage

The ANSI C standard library contains a pseudo-random number generator whose interface consists of the two functions *srand* and *rand*, as well as the constant `RAND_MAX`.

Before the generator can be used, it is advisable to set its seed value with the *srand* function. The seed value is an unsigned integer which determines the sequence of numbers generated, so that the same seed will always result in the generation of the same sequence.

After the generator has been seeded, the *rand* function is used to obtain the next random number in the sequence. The random numbers produced by rand() are integers in the range 0 to `RAND_MAX` inclusive.

7.2 Implementation

7.2.1 Linear congruential generator

I have implemented the rand() function is a so-called linear congruential generator (LCG). It is called linear because its equation does not contain exponents, and it is called congruential because all numbers are calculated with respect to a certain modulus m (congruential calculus).

The sequence of pseudorandom numbers x_0, x_1, ... is defined recursively by means of the following equation:

$$x_i = ax_{i-1} + b \pmod{m}$$

Note that a, b, m and x_0 are constants. The value of x_0, i.e. the base case of the recursion, is called the seed, and is the value passed as argument to the srand() function.

Linear congruential generators vary greatly in their properties and in their usefulness for different types of simulations. A detailed account of the mathematics involved is beyond the scope of this document, but in general, the aim

is to choose a, b, and m so that the generator will have the greatest possible period p.

The period p is defined as follows:

$$x_0 = x_p$$

$$\forall n \qquad 0 < n < p \rightarrow x_n \neq x_0$$

In itself, a long period is not enough of an indication that a random generator will be useful in a certain context. Consider, for instance, an application which only uses the least significant bit of the random number produced. Even if $p = m$, i.e. the period equals the modulus and is therefore as long as it can possibly get for a given m, the generated numbers might alternate between being odd and being even, making the sequence of least significant bits extremely predictable.

In general, LCGs should not be used for cryptographic applications. With reasonable parameters, however, they have been demonstrated to be strong enough for most types of simulations. For efficiency reasons I have chosen to implement rand() as an LCG rather than using a slower, non-linear method of generation.

7.2.2 Parameters

My implementation of the LCG uses the following parameters, which are defined as preprocessor macros in stdlib.c:

$$m = 2^3 1 - 1$$

$$a = 7^5 = 16807$$

$$b = 0$$

$$x_0 = 1$$

In [6] this generator is commented upon as follows:

"... based largely on the fact that this generator is a full period generator, this generator has in subsequent years passed all new theoretical tests, and (perhaps more importantly) has accumulated a large amount of successful use."

Chapter 8

setjmp.h, non-local jumps

8.1 Introduction

E. W. Dijkstra, in his famous article "Go To Statements Considered Harmful", has pointed out that the excessive use of jump statements could greatly decrease the readability of a program text because "it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress." In practice, this means that in source code liberally cluttered with goto statements, one cannot follow the evolution of the algorithm in time by following the source code in text space. In my implementation of the ANSI C standard library, I followed Dijkstra's advice not to use goto statements, sometimes resulting in redundant, but hopefully more readable and modifiable, source code.

There are situations, however, where it seems reasonable to immediately "jump out" of a sequence of nested function calls to some well-defined point in the source code. Contemporary programming languages, including C++ and Java, have introduced the concept of an "exception." The idea is based on the observation that most errors, i.e. exceptional program states, cannot be satisfactorily handled at the point in the source code where they occur. So when an exception is "thrown", stack frames are destroyed until an exception handler is found, i.e. a piece of code which was specifically designed to handle that kind of error. Exceptions make it possible to tidy up the source code by separating the "usual" code from the error handling routines. No such luxury is provided in ANSI C, where return values generally have to be checked for error indicators, and regular code and error handling routines usually end up interspersed with each other.

There is, however, a mechanism in ANSI C which provides the functionality of non-local jumps out of a hierarchy of nested function calls. This functionality is provided by the header *setjmp.h*, which is part of the ANSI C standard library.

8.2 Usage

The header setjmp.h is minimalistic in that it declares but one data type and two functions. The following code illustrates their use:

```
#include <setjmp.h>
```

```
#include <stdio.h>

void f(void);
void g(void);
jmp_buf jb;

int main(void) {
  puts("main(): Starting");
  switch(setjmp(&jb)) {
  case 0:
    puts("main(): State saved to jb");
    break;
  default:
    puts("main(): Ended up here after jump");
    return 0;
  }

  f();
  return 0;
}

void f(void) {
  puts("f(): Starting");
  g();
  puts("f(): Ending");
}

void g(void) {
  puts("g(): Starting");
  longjmp(jb, 1);
  puts("g(): Ending");
}
```

Before the function longjmp() may be used to execute an actual jump, the function setjmp() must first be called to save the current processor state to a buffer, i.e. to define the target of the jump. At this time, the return value of setjmp() will always be zero.

Later on, when the non-local jump is to be executed, the application calls the function longjmp() with two arguments: First, a pointer to the buffer containing the saved processor state, and second, an integer value which we will arbitrary call "val". After the jump has taken place, the illusion will be that of setjmp() returning with the value "val" (hence the switch statement in the above example). So, based on the return value of setjmp(), the application determines if it has only just saved its state, or if it is now returning from a successful non-local jump.

In chapter 18 I illustrate a real-life example where setjmp.h is used to approximate the exception mechanism familiar to C++ or Java programmers. For now, suffice it to say that setjmp.h may facilitate certain powerful programming techniques, but, like all goto-like statements, should be used wisely and sparingly.

8.3 Implementation

8.3.1 The buffer type

The data type `jmp_buf` is used to save the information which is necessary to jump back to the point at which the state was saved, i.e. at which setjmp() was called. This type is declared as follows:

```
typedef struct {
  int r29, r31; /* stack pointer and return address */
  int r8, r9, r10, r11, /* variables */
  r12, r13, r14, r15,
  r16, r17, r18, r19,
  r20, r21, r22, r23;
} jmp_buf;
```

8.3.2 setjmp() and longjmp()

The buffer contains fields for certain designated registers, and all that setjmp() has to do is save the values currently stored in those registers to the buffer, and then return zero.

The longjmp() function simply restores the saved values to their original registers, then returns its second argument (val). Since one of the registers affected by the operation is the return address, it appears as if setjmp(), not longjmp(), were returning.

The source code for those functions can be found in the file lib/asm/setjmp.s, in ECO32 assembly language. Apart from the startup code, this is at the moment the only part of the library which uses assembly routines.

8.4 Deviation from ANSI C

This implementation deviates from the ANSI C standard in the following way: The ANSI C standard defines the argument to setjmp(), and also the first argument to longjmp(), to be of type `jmp_buf`. To improve the efficiency of these functions, my implementation uses pointers to `jmp_buf` instead. Other than that, it is ANSI C compliant.

Chapter 9

Variable Argument Lists

9.1 Usage

There are situations when a function needs to be able to deal with an arbitrary number of arguments. Consider, for instance, the following declaration from stdio.h:

```
int printf(char *fmt, ...);
```

The three dots in the above declaration are valid C syntax, meaning that any number of arguments of any type may be provided when printf is called.

If a function is declared in this way, the three dots must always be followed immediately by the closing parenthesis of the argument list. In addition, there must always be at least one named parameter, so the following declaration would not be allowed:

```
void this_is_an_error(...);
```

The ANSI C standard library includes a header called stdarg.h which defines a set of macros related to variable argument lists. The actual implementation of these macros may be very machine-dependent, but as the library encapsulates such dependencies, variadic functions may be written, and called, in a portable way.

Specifically, stdarg.h defines the following macros:

- `va_list`: The type of a variable acting as a pointer into the variable argument list. Every function with variable arguments defines a local variable of this type, and then uses this variable to iterate through the unnamed arguments in sequence.
- `va_start(ap, lastnamed)`: Initialises the argument pointer to point to the first unnamed argument. `ap` must be the name of a variable of type `va_list`. `lastnamed` is the name of the last named parameter of the function. This is necessary for calculating the memory address at which the variable argument list starts.
- `va_arg(ap, type)`: This macro evaluates to the members of the variable argument list in the sequence they were specified in the function call. The caller of the macro must know the type of the argument to be retrieved.

- `va_end()`: This macro should be called by a function after it has finished retrieving arguments from the list, giving the library a chance for some cleaning up. For instance, it might free storage that was allocated by `va_start`.

A function using variable-length argument lists has no safe way of finding out with how many arguments it has been called. Also, the function must deduce the types of the unnamed arguments before retrieving them. printf, for instance, deduces the number and types of the variable arguments by analyzing its first argument, a format string with placeholders for every additional argument. If, however, printf is called with the wrong number of arguments or with an argument of the wrong type, the compiler has no way of detecting the error. So, in general, passing a pointer to an array should be preferred over variable argument lists whenever this is possible.

9.2 A Coding Example

The following function, named `varsum`, returns the sum of its arguments of type int. The last argument must be 0 so the function knows where the list ends.

```
int varsum(int first, ...) {
  int retval=0; /* return value */
  int curr=first; /* current argument */
  va_list ap; /* argument pointer */
  va_start(ap, first);
  while(curr) {
    retval+=curr;
    curr=va_arg(ap, int);
  }
  va_end(ap);
  return retval;
}
```

9.3 Implementation of stdarg.h

On most machines, including the virtual one we are dealing with, a function is called using the following method:

1. The caller pushes the arguments onto the stack, in reverse order of appearance in the function call. It may be necessary to cast the arguments to a different type so they match the function's prototype. (On ECO32, up to four arguments are not actually pushed onto the stack, but are instead kept in four registers designated especially for this purpose. Fortunately, however, the difference is hidden from the C programmer by the compiler's back-end. To all intents and purposes, a C program on ECO32 can treat arguments to functions as if they were living on the stack.)

2. The caller pushes the return address onto the stack. (This step is unnecessary if the function does not itself contain a function call. In this case, the return address is merely kept in a special register.)

3. The caller sets the program counter to the entry point address of the function.

4. The function does whatever is necessary to compute its result.

5. The function places its return value into some register.

6. The function pops the return address from the stack and jumps there, so execution flows back into the caller.

7. The caller cleans up the stack, i.e. it removes the arguments it had previously pushed.

If we assume the stack to be a contiguous region of memory growing downwards as elements are being pushed onto it, then we find that the variable arguments end up occupying the memory addresses immediately following the last named argument.

My implementation of stdarg.h defines the type `va_list` to be equivalent to `char *`. The macro `va_start(ap, lastnamed)` initialises `ap` to point directly after the last named argument. This is accomplished by taking the address of `lastnamed` and adding `_ARGSIZE` to it. This is a special constant, also defined in stdarg.h, which gives the interval between the memory addresses at which arguments to functions are placed. On ECO32, which is a 32-bit architecture, this value is 4.

The `va_arg` macro returns the data located at the address contained in `ap`, converted to the given type. It also increments `ap` by `_ARGSIZE`, so that it now points to the next argument (if present).

My implementation of `va_end()` does nothing at all. Since `va_start` and `va_arg` do not allocate any storage, no cleanup activities are necessary.

9.4 Limitations

The current implementation does not allow structures being passed as anonymous arguments. This should never be an issue, however, because structures are almost always passed "by reference", i.e. by means of a pointer.

9.5 Deviations from the ANSI C standard

The implementation of stdarg.h complies with the ANSI C standard.

Chapter 10

The Dynamic Storage Allocator

10.1 Anatomy of the address space

In the programming model of C, variables fall into three distinct storage classes:

- Static storage: A variable defined outside of any compound statement, i.e. at the same level as function declarations, becomes a static variable. In addition, a variable defined within a function definition is static if the definition is preceded by the *static* keyword. Static storage is allocated at the time of loading and remains allocated until the process terminates. When such variables are explicitly initialized with non-zero values, they will usually end up in the data segment of the executable and, thus, of the process. Uninitialized global or static variable may also be stored in the bss segment. The difference is that the values of variables in the bss segment are not stored in the executable file, so that disk space and loading time are saved.

- Automatic storage: Any variable defined between braces and without the *static* keyword is an automatic variable. ANSI C defines the optional *auto* keyword which can precede a variable definition to emphasize that the variable is automatic. An automatic variable is allocated when execution enters its scope; usually the process of allocation is as simple as decreasing the stack pointer. Every instance of a function gets its own automatic (local) variables, making recursion possible. An automatic variable is freed when its scope is left; it vanishes along with the stack frame in which it lived.

- Dynamic storage: Both static and automatic storage allocation are handled by the compiler and the linker, which implies that the amount of memory needed must be known at compile time. ANSI C ensures this by requiring the size in an array declaration to be a constant expression, i.e. one which can be computed by the compiler. However, when the size of a data structure cannot be determined at compile time, such as the storage

needed to hold the symbol table of a compiler, then memory must be allocated dynamically, and freed explicitly when it is no longer needed. The address area holding such dynamic variables is called the heap, and may be considered an extension of the bss. The top end of the heap is called break, so that any address greater than or equal to the break is not in the address space if it is less than the stack pointer. The library functions for heap management are the subject of this chapter.

10.2 Usage

The header file stdlib.h defines a number of functions to manage dynamic storage. Specifically, *malloc(n)* allocates n bytes (actually char-sized units) of memory, returning a pointer to the newly allocated storage. *realloc(ptr, n)* takes as its first argument a pointer to a region of memory previously allocated with malloc, reallocating it to the new size n bytes. Finally, *free* is used to relinquish dynamic storage, giving it back to the storage allocator as free space which can be allocated by subsequent calls to malloc or realloc.

For the programmer's convenience, the ANSI C library includes the additional function *calloc*, which not only allocates storage but also sets all the bytes in the newly allocated area to zero.

In the event that the system runs out of memory and the desired storage cannot be allocated, malloc, calloc and realloc all return the NULL pointer. In addition, the variable errno will have been set to the constant ENOMEM.

10.3 Implementation

Unix provides the *sbrk* system call which is used by processes to ask the operating system for additional memory. Technically, this system call causes the kernel to try to increase the size of the bss segment of the calling process, returning a pointer to the new area, or NULL if the segment could not be enlarged (which is unlikely to happen). At one time I was tempted to implement malloc as almost synonymous with sbrk, but this approach is inapplicable because system calls are lengthy operations, and memory allocation is so frequent that efficiency must be considered.

My implementation of the storage allocator is based loosely on the one proposed by Kernighan and Ritchie in [2]. I decided to re-implement it to improve my understanding of the details, and also to avoid the cryptic C code that comes with maximum efficiency—after all, the source code is likely to get used as educational material.

The storage allocator maintains a pool, called *arena*, of free memory blocks, each one starting with a header which contains its size (in header-sized chunks) and a pointer to the beginning of the next free block in the list. The blocks are kept sorted in the order of increasing memory address, with the last block pointing back to the first (cyclic linked list).

malloc() works by traversing the freespace list until it finds a block large enough to store the requested number of bytes, i.e. it performs a first fit search on the list. If the match is exact, then the block is simply unlinked from the list and a pointer to the free space is returned. If the block is larger than needed,

only the necessary part is unlinked, the remaining space becoming a free block of its own.

If no match is found, the required storage is rounded up to the next kilobyte (this is arbitrary), and a call to sbrk takes place. If this fails, i.e. the operating system has run out of memory, then malloc gives up and returns NULL.

Preferring a first fit algorithm over a best fit one is a design decision which I believe needs explaining: A best fit algorithm always chooses the smallest block that would still be large enough to store the required number of bytes. Over time this leads to the creation of lots of small blocks (those which malloc "cuts" off the end of the blocks it unlinks). The first fit approach is more likely to leave larger blocks in the list and thus to avoid what is known as memory fragmentation.

The function free() traverses the arena to find the correct place to link the freed block (remember that the list is sorted). If the freed block is directly adjacent to a block already in the list, then the two blocks are merged into one large block by simply unlinking the second and changing the size of the first accordingly.

The function realloc() has to differentiate between two cases.

1. If a block is reallocated to a smaller size, then this block simply has its "tail" removed and linked back into the freespace list by a call to free(). A special case is that `realloc(ptr, 0)` is synonymous with `free(ptr)`.

2. If realloc() is used to enlarge a block, it simply frees the block by calling free(), then calls malloc() to allocate a block of the desired size. Finally, realloc() calls memmove() (defined in string.h) to move the user data from the old block to the new one. (Note that free() and malloc() are optimized in such a way that, whenever possible, the old and the new block will start at the same address. memmove() is "clever" enough not to copy anything if the source and destination pointers are found to be equal, so the efficiency of realloc() is greatly increased for this case.)

Chapter 11

Assertions

11.1 Usage

In software engineering, when one writes a formal specification for an abstract data type, this specification will usually contain preconditions and postconditions for its methods. The idea is that calling a method is reasonable only if its precondition holds, and the implementation is considered correct only if the postcondition holds when the method has returned.

Similarly, an assertion states that at a certain point in the execution of a program, a given expression must yield a true (non-zero) value, otherwise it would make no sense to continue executing.

The header file assert.h allows for such assertions to be inserted into the source code. This header defines the *assert* preprocessor macro which expects an arbitrary expression as argument. Example:

```
assert(i<=j);
```

In case the expression evaluates to a non-zero value, execution continues as usual, otherwise the programm would terminate abnormally, with the following error message:

```
Assertion "i<=j" failed!
File: myfile.c
Line: 42
```

"Abnormal termination" means that no cleanup activities will be carried out, in particular, no I/O buffers will be flushed. This is yet another reason why the stream "stderr" is never buffered.

For a beta version of an application it is perfectly reasonable to abort due to a violated assertion, but not in the final release. Fortunately, when the *NDEBUG* flag is defined, the assert value will no longer have any effect. The fact that assertions can be so easily deactivated (or reactivated if necessary) should encourage programmers to keep the assertions in their code in case they are needed later on for debugging purposes.

11.2 Implementation

"assert" is implemented by means of the following preprocessor macro:

```
#define assert(e) ((e) || _failed_assert(#e, __FILE__, __LINE__))
```

Because of short circuit evaluation, the call to `_failed_assert` takes place only if the expression *e* evaluates to "false".

The function `_failed_assert`, defined in assert.c, prints the appropriate error message, then calls `abort` (defined in stdlib.h) to halt the program.

`_failed_assert` expects three arguments: A string representation of the expression, the name of the source file containing the assertion, and the line on which it occurs. `__FILE__` and `__LINE__` are special macros which the preprocessor substitutes with the file name and line number, respectively. The construct `#e` means "the argument e, expressed as a string". The preprocessor, when parsing this construct, simply takes the value of `e` and places double quotes around it.

When the flag "NDEBUG" is defined, the following definition of assert is used instead of the one described above:

```
#define assert(e) ((void) 0)
```

`((void) 0)` is an expression which yields no value; appending a semicolon makes it an expression statement with no side effects, for which a clever compiler will not produce any machine instructions.

Chapter 12

Program startup and termination

Most C programmers like to think of the main() function as the entry point to their creations. A C program is said to begin execution when the main() function starts executing its first statement, and terminates when main() returns. In truth, the standard library carries out certain preliminary steps when a program starts, and only after those steps have been dealt with is the main() function actually called. Similarly, after main() returns, the library engages in some cleanup activity before the process actually terminates.

This chapter briefly deals with the very first, and very last, activities which take place in the context of a process. The primary focus is on how the ANSI C standard library is involved in those activities.

12.1 Program startup

12.1.1 Process creation

The Unix kernel identifies a process by means of a process id (pid), which is a small integer (between 1 and 30,000, inclusive). Whenever a process P issues a successful *fork* system call, a new process C is created and becomes a *child process* of P, which is said to be its *parent*. Thus a hierarchy of processes is created by a sequence of calls to fork().

The root of this hierarchy is called the initialization process, or init task. It is characterized by three facts:

1. It is not created by another process calling fork(), but by the kernel during booting.
2. Its pid is always 1.
3. It never terminates.

When a process terminates abnormally or by using the *exit* system call, its task state (including the exit status) is stored in memory until its parent retrieves the exit status by means of the system call *wait*. A process which

has terminated, but whose exit status has not yet been retrieved in this way, is sometimes referred to as a "zombie process".

If the parent P of a process C terminates without retrieving the exit status of C, then C will automatically become a child of the init task so the process tree remains intact.

12.1.2 Loading a program

An executable is loaded into the context of a process by means of the *exec* system call or one of its derivatives (e.g. execvp). The data, code and bss segments of the process are created and initialized, then execution continues at the start of the code segment, which is where the C library takes control.

12.1.3 Clearing the bss segment

The bss segment of a process contains blocks started by symbols, hence the abbreviation. In contrast to the data segment, the initial content of the bbs segment is not stored in the executable; instead, the bbs may or may not be initialized by the kernel. In the worst case, it might still contain data left over from the operation of another process. I have heard of at least one case in which a password was compromised with the help of an application which would allocate a large array in the bss, then simply print out its contents.

To eliminate this problem, and also to auto-initialize global variables to zero, the library starts off by overwriting the bss with zero bytes.

12.1.4 Calling main() with command line arguments

The main() function expects two arguments: An integer value *argc*, and an array of strings *argv*, with the following semantics: The array argv contains the strings passed as command line arguments, and argc contains their count.

The following assembly code calls main() with command line arguments (no line numbers in the original source code):

```
1: ldw $4,$29,0
2: add $5,$29,4
3: jal main
```

To understand the above code, it is necessary to know that `$29` is used as the stack pointer and that the registers `$4` to `$7` are, by convention, used for passing arguments to a function.

The parameter argc has been placed by the kernel at the top of the stack, and is loaded into `$4` by line 1. Above argc on the stack, the kernel has placed the pointers to the actual argument strings, so in line 2 we set `$5` to `$29+4`. This is a pointer to a pointer to char, and is exactly what main() expects to find.

Finally, in line 3 above, the main() function is called. jal means "jump and link". This instruction places the return address into a designated register (`$31`), then sets the program counter to the start of main().

12.2 Program termination

12.2.1 Abnormal termination

The kernel may notify a process of certain events by sending it a so-called "signal".[1] A process uses the *signal* system call to define handler functions for those signals which it is prepared to receive. If a process is sent a signal which it does not handle, a default handler is called. This default handler may simply ignore the signal and restore control to the program where it left off, or it may terminate the process. The latter case, i.e. process termination due to an uncaught signal, is called abnormal termination.

In this case, the library has no chance of interfering with the flow of events. In particular, it has no possibility of flushing its I/O buffers, resulting in a higher or lesser degree of data loss depending on the circumstances.

12.2.2 Normal termination

A process may terminate normally in one of three ways:

1. By returning from main().
2. By calling the library function *exit*.
3. By calling the library function `_exit` (notice the underscore).

As we shall see, the first two cases are equivalent. This is because of the following assembly code (which immediately follows the code shown in the previous section):

```
add $4,$2,$0
jal exit
```

In this way, a return from main() will always get routed to a call to exit(). Note that in the first line, the return value from main (`$2`) is stored in `$4` (where exit() expects its argument to be).

The third case above, i.e. the call to `_exit`, results in an exit system call. No cleanup activities are executed, no buffers are flushed, and the process terminates immediately (with the exit status given as the argument to `_exit`).

12.2.3 The exit() function

This function is responsible for carrying out all the cleanup tasks necessary prior to process termination. In the current implementation, exit() does the following:

1. It executes all functions registered with atexit(), in reverse order of registration (see below).
2. It calls the internal library function `_iocleanup()` which closes all streams, thereby flushing all pending buffers.
3. Finally, it issues the exit system call, resulting in process termination.

[1] Sometimes referred to as "software interrupt"

The *atexit()* function provides a way to register a number of functions for execution when the program terminates. The functions will be executed in reverse order of their registration. The atexit() function expects a function pointer as its only argument, and returns zero on successful registration, non-zero otherwise. The constant `_ATEXIT_MAX`, defined in stdlib.h, gives the maximum number of functions which may be registered in this way.

Registration is implemented by means of an array of function pointers which is called `atexit_funcs` and is static to one of the library modules. The int variable `atexit_num` holds the number of functions currently registered.

Chapter 13

Introducing the shell

13.1 Motivation

At some time during testing of the standard library, it became apparent to me that my work would be greatly facilitated by a *shell*, i.e. an application which could be used to launch foreground and background processes, and to make decisions based on their exit status.

I began with a minimal variant which simply carried out the following:

1. Read the name of an executable from standard input,
2. start this executable in a sub-process,
3. display its exit status,
4. and finally, jump back to step 1.

However, the shell quickly developed into a small sub-project of its own, and so this and the following chapters are primarily devoted to it.

I decided to call the shell "MINSH", an acronym which may be interpreted as standing for "minimal shell", or, recursively, for "MINSH is not SH". While it is indeed far from being SH (which refers to the Borne Shell), I have tried to structure it in such a way as to make it easy to extend, during future work on the ECO32 project, into a full-featured command interpreter comparable to BASH or the C Shell.

13.2 Shell structure

Software engineering tells us that if we are in possession of the exact description of a process, the nouns contained in that description might give us a clue as to what kinds of classes the system might include. For instance, when looking at the description of the game of chess, we may infer that we need classes for the different kinds of pieces, for the board, and for the players. This would certainly be a reasonable start for the object-oriented design of a chess program.

While the nouns in a description indicate the kinds of objects involved, the verbs will usually indicate the kinds of processes. For instance, consider the following description of how a shell works:

- The shell scans its input for tokens (i.e. file names, operators and keywords)
- It parses the resulting list of tokens for commands.
- It transforms the commands into a machine-readable internal representation.
- Finally, it executes the commands thus transformed.

Interestingly enough, after some designing and experimenting the shell ended up having a modular structure which corresponds exactly with the verbs in the above description. The following modules now make up the shell:

- scanner.c: The lexical scanner
- parser.c: The parser which checks the input for syntactic correctness and transforms it into an internal representation
- internal.c: Contains utility functions for operating on internal representations
- execute.c: Reads the internal representation and executes it

In addition, the following two helper modules became necessary:

- error.c: The module responsible for error handling, error messages etc.
- main.c: The module containing the main() function, which does little more than initialize a few global variables and call the parser.

The following chapters describe the shell in more detail, each chapter concerning itself with another of the shell's modules.

Chapter 14

Shell grammar

Before going into any detail concerning the shell's implementation, it is necessary to give an exact specification of the language interpreted by the shell, along with its semantics. This chapter starts with a specification of the terminal symbols (tokens) recognized by the scanner. Following this, a contextfree grammar for the shell language is introduced and then iteratively refined until it becomes possible to implement a recursive descent parser for it.

14.1 Overview

There are few areas of computer science which have been researched more extensively than compiler engineering, and attempting to do more than scratch the surface of this area would result in a book of its own. So this section restricts itself to giving just some basic definitions of terms used throughout the remainder of this chapter. The so-called Dragon Book [4] gives most of the algorithms in detail.

The problem of describing a machine-readable language is usually subdivided into three levels:

1. The alphabet: This is the set of symbols the language consists of at its lowest level. For instance, the alphabet of the English language would consist of lower case letters, upper case letters, numbers, and punctuation symbols (such as comma or semicolon). The alphabet of the shell is the ASCII character set.

2. Tokens: A token, also referred to as a terminal or lexeme, is a small, meaningful sequence of characters in the alphabet. To continue our running examples, the tokens of the English language are its words and punctuation marks, and examples of shell tokens would be a filename or the keyword "then". The meaning of a token may depend on its context. The process of searching the input stream for tokens is called scanning.

3. Sentence: Sentences are meaningful sequences of tokens. A grammar is used to describe exactly which sequences of tokens constitute sentences and which do not.

14.2 Grammars

From a mathematical point of view, a language is nothing more than a set of valid sentences, and a grammar is one way of describing such a set. A grammar consists of the following:

- A set of terminal symbols (T)
- A set of nonterminal symbols (N), with one of them designated as the start symbol S
- A set of production rules (R).

A production rule is written as an arrow to the left and to the right of which are sequences of terminal and nonterminal symbols. While deriving a sentence, the sequence to the left of the arrow may be replaced by the sequence to the right.

To derive a sentence from a grammar, one begins with the start symbol, then applies the production rules until only terminal symbols remain. The language then consists of all the sentences which may be derived in this way.

Finally, a contextfree grammar is defined as a grammar in which the left side of each production rule consists of exactly one nonterminal symbol. It has been proven that there are languages which cannot be described by means of a contextfree grammar, but those languages are the subject of research and are irrelevant to practical applications. As it is, a contextfree grammar suffices to describe most any programming language, including, for instance, C++ and Perl.

14.3 Shell tokens

The shell, in its present state, recognizes the following tokens:

- Word: This is a sequence of characters which has no special meaning to the shell. It might be a filename or an argument to a command. Certain characters, called metacharacters, are not allowed within words unless the word is enclosed in double quotes.
- Pipe: This is the `|` operator (vertical bar) which means that the output of a command should be redirected to become the input of another.
- Background: This is the `&` operator (ampersand) which means that a command should be executed in the background, i.e. its termination should not be waited for.
- Or: This is the `||` operator which means that a command should be executed if and only if another command failed.
- And: The `&&` operator means that a command should be executed if and only if another command succeeded.
- End: A token which marks the end of a command (this is either a semicolon or a newline character).

- Eof: The end of the input stream being scanned.
- Keyword: One of the words “if”, “then”, “do”, “fi”, “else”
- Braces (`{` and `}`): These are used to express nested lists.
- Parentheses (“(” and “)”): A list enclosed in parentheses is executed in the background.

14.4 Contextfree grammar

14.4.1 First attempt

Now that the alphabet and the lexemes have been defined, it is time to give a first contextfree grammar for the shell language, liberally interspersed with explanations concerning its semantics. By convention, I use capital letters for terminal symbols and lower case letters for nonterminals.

1	list	→	pipeline
2	list	→	list END list
3	list	→	list BACKGROUND list
4	list	→	list AND list
5	list	→	list OR list

At the top level, the shell expects its input to be a list of pipelines to execute (i.e. “list” is the start symbol). The pipelines are separated by operators which denote the control flow. A semicolon or newline character causes sequential execution of the list elements it separates. The `&&` operator causes its right side to be executed if and only if the left side succeeds, i.e. returns an exit status of zero. Similarly, the `||` operator causes its right side to be executed if and only if its left side fails, i.e. returns a non-zero exit status. Finally, the `&` operator causes its left side to be executed in the background, i.e. in a sub-process which is not waited for. The exit status of such a background execution is defined to be always zero.

The next level is the pipeline:

6	pipeline	→	command
7	pipeline	→	pipeline PIPE pipeline

A pipeline is a sequence of one or more commands, separated by the pipe operator (`|`). When a pipeline is executed, pipes are created to connect the commands so that the output of the first one is used as the input for the second, whose output in turn is used as input for the third etc.

So a shell script consists of pipelines which in turn consist of commands. The anatomy of a command is as follows:

8	command	→	simple-command
9	command	→	if-command
10	command	→	LEFTPARENTHESIS list RIGHTPARENTHESIS
11	command	→	LEFTBRACE list RIGHTBRACE
12	simple-command	→	WORD
13	simple-command	→	WORD args
14	args	→	WORD
15	args	→	WORD args
16	if-command	→	IF list op THEN list op FI
17	if-command	→	IF list op THEN list op ELSE list op FI
18	op	→	END
19	op	→	BACKGROUND

This grammar defines four types of commands:

A simple command is the instruction to execute a program with certain arguments. The first WORD of the command is the filename of the executable, the remaining WORDs are passed as arguments to that executable when it is loaded. By convention, the shell passes the name of the executable as first argument (so an application may easily determine its own location in the file system).

A list enclosed in braces is simply a sub-list which is executed in the same shell process. This provides a way to express nested lists, or to pipe the output produced by one list into another.

A list enclosed in parentheses is a sub-list which is executed in the background.

Finally, an if statement allows for conditional execution of lists. The list following the keyword "then" is executed if and only if the list following the keyword "if" succeeds, otherwise the "else" clause is executed (if present).

14.4.2 Refinements

From a theoretical point of view, the grammar given above is a complete description of the shell language. All valid shell scripts, and nothing else, can be derived from it. However, there are still two practical problems with this grammar, the first of which is left-recursion, and the second is operator precedence.

Left-recursion

A left-recursive rule is one whose right side begins with its left side. When the left-recursion is removed from a grammar, it generally becomes much easier to implement a parser for it. In particular, a recursive descent parser (see below) cannot be implemented for a grammar which still contains left-recursion.

There exists an algorithm to remove left-recursion from a grammar. Basically, the right sides of the rules in question are split apart into a "beginning" and a "rest" (which might be candidates for new nonterminal symbols). When the rules for those new nonterminals turn out to be left-recursive in themselves,

the same process of splitting is repeated until the left-recursion has been removed.

As an example, consider the first five rules in the grammar given in the previous question. Since these are the only rules for the "list" nonterminal, it becomes obvious that a list must always begin with a pipeline, and we are left with the task of specifying the several ways in which it may continue.

There, then, is the full shell grammar after applying the transformation:

1	list	→	sub-list list-rest
2	list-rest	→	END sub-list list-rest
3	list-rest	→	END sub-list list-rest
4	list-rest	→	EMPTY
5	sub-list	→	pipeline sub-list-rest
6	sub-list-rest	→	AND pipeline sub-list-rest
7	sub-list-rest	→	OR pipeline sub-list-rest
8	sub-list-rest	→	EMPTY
9	pipeline	→	command pipeline-rest
10	pipeline-rest	→	PIPE pipeline
11	pipeline-rest	→	EMPTY
12	command	→	simple-command
13	command	→	if-command
14	command	→	LEFTPARENTHESIS list RIGHTPARENTHESIS
15	command	→	LEFTBRACE list RIGHTBRACE
16	simple-command	→	WORD args
17	args	→	WORD args
18	args	→	EMPTY
19	if-command	→	IF list op THEN list op optional-else FI
20	optional-else	→	ELSE list op
21	optional-else	→	EMPTY
22	op	→	END
23	op	→	BACKGROUND

Determining operator precedence

When comparing the two grammars listed above, you will find that I have "cheated" by doing a little more than removing left-recursions. In particular, I have introduced a new nonterminal, sub-list, together with its companion sub-list-rest. The reason for this move was to give the two list operators `||` and `&&` a higher precedence than the operators `;` and `&`. For instance:

```
cmd1 && cmd2 ; cmd3 && cmd4
```

Will now be correctly interpreted as:

- Execute cmd1.
- If cmd1 succeeded, execute cmd2.
- In any case, execute cmd3.
- If it succeeded, execute cmd4.

In general, operators are given higher precedence by introducing a new non-terminal which produces expressions containing only those operators, and isolate them from the rest of the grammar.

Chapter 15

Shell scanner

15.1 Purpose

The scanner is that part of the shell which operates directly on the input stream, hiding its details to the higher-level modules of the shell. It scans the stream for tokens, returning their types and, if appropriate, allowing access to the actual text string which constitutes the token.

15.2 Interface

- `GetToken(void)`: Scans for the next token from the stream, skipping whitespace if present. Returns the token type, which is one of the constants defined in scanner.h whose names begin with "TT". In addition, this type value is also assigned to the global variable `token_type`, and, if appropriate, a pointer to the text constituting the token is assigned to the global variable `token_text`.

- `SetInputStream(fp)`: By default, the scanner reads from standard input, but this may be changed using this function. SetInputStream() should be called prior to the first call to GetToken().

- `SetScannerContext(c)`: Sometimes a token needs to be interpreted differently depending on its context as determined by the parser. In the current implementation there are two contexts: If the argument to SetScannerContext() is non-zero, the scanner will not recognize keywords but will return `TT_WORD` instead. An argument of zero will turn keyword recognition back on. The parser makes use of this feature so that, for instance, the command "`rm fi`" becomes possible (`fi` is passed as argument to `rm`).

- `ShowPrompt()`: Displays a prompt if the input stream is connected to a terminal, and does nothing otherwise. It is usually not necessary to call this function from outside the scanner.

- `SkipLine()`: Causes the scanner to skip the rest of the current input line. This may be useful when recovering from a parse error.

Chapter 16

Shell parser

16.1 Purpose

The parser repeatedly calls the GetToken() function provided by the scanner to retrieve tokens from the shell's input stream. It parses the sequence of tokens according to the contextfree grammar given in chapter 14. The result of the parsing process is an internal representation of the user input which may be efficiently executed by the execution module.

Note that the shell may be viewed as consisting of a compiler and an interpreter, the compiler consisting of the scanner and parser modules, the interpreter represented by the execution module.

16.2 Interface and implementation

I have implemented the parser as a recursive descent parser, i.e. one which constructs the parse tree from the top down, beginning with the start symbol. The initial implementation contained a function for every nonterminal in the contextfree grammar of chapter 14. In the current implementation, some of these functions (for instance the one for list-rest) have been optimized away for efficiency reasons. Also, some of the parser functions were initially tail-recursive, and in the current implementation those tail-recursive calls have been replaced with iterations.

The interface consists of the following functions:

- `ParseList`, `ParseSublist`, `ParsePipeline`, `ParseCommand`, `ParseSimple`: These functions parse the corresponding nonterminals. All of them take a pointer to `internal_t` as argument and store their result through this pointer (see below for details on internal representation).
- `Parse()`: This is the driver loop for the parser, which is called from the shell's main module.

Chapter 17

Internal representation of shell commands

17.1 What is the internal representation?

The internal representation is sort of an intermediary code which the parser generates, and which is fed as input into the execution module which then carries out the instructions it contains. There are at least two good reasons for this approach:

1. When executing a loop, it would be a waste of time to have the shell parse certain lists of commands over and over again. Instead, the shell simply keeps a pointer to the internal representation of the list.
2. Certain shell constructs need to be parsed completely before the shell can begin executing them. In such cases, the internal representation provides a way for the shell to "remember" what it has already parsed.

Furthermore, future versions of the shell might implement a serialization mechanism for the internal representation, so that entire shell scripts may be precompiled, saved to a file on disk, and executed more efficiently later. Since the internal representation does not contain any caclic structures, it would be relatively easy to design and implement such a mechanism.

17.2 Specification of the internal representation

17.2.1 The structure

The data type for storing the internal representation is called `internal_t` and is defined in internal.h, as follows:

```
typedef struct {
  void *p1, *p2, *p3;
  int i1, i2, i3;
  int type;
}
internal_t;
```

The semantics of the three pointers and the three integers depend on the value of "type", which is one of the constants defined in internal.h whose names begin with IT (for internal type). The following subsections describe each of the types currently supported. Note that these types correspond to some of the nonterminals in the shell grammar, so the internal representation resembles a parse tree.

17.2.2 Simple command

A simple command is represented by a node of type `IT_SIMPLE`. The `p1` member points to the name of the executable file to launch, and `p2` points to the array of command-line arguments to be passed to the program.

17.2.3 Pipeline

A pipeline is represented by a node of type `IT_PIPELINE`. `p1` points to the representation of the first command in the pipeline, and `p2` points to the rest of the pipeline (which is also a pipeline, or the NULL pointer if this node represents the last command).

17.2.4 List

A list is represented by a node of type `IT_LIST`. The `p1` member points to the first element of the list, the `p2` member points to the rest (which is itself a list, or the NULL pointer if this node represents the last element). In addition, `i1` specifies what type of list this is, meaningful types being the following constants:

- `LT_SEQ`: A sequence. The first element is executed and waited for, then the rest is executed.
- `LT_BG`: A background sequence. The first element is executed in the background, then the rest is executed.
- `LT_AND`: A conjunction. The first element is executed, and the rest is executed if and only if the first element returned an exit status of zero.
- `LT_OR`: A disjunction. Analogous to `LT_AND`, but the rest is executed if and only if the first element returns a nonzero exit status.

17.2.5 If statement

An if statement is represented by a node of type `IT_IF`. `p1` points to the condition, `p2` points to the consequence, and `p3` points to the alternative (if an else clause was present in the if statement). Otherwise, `p3` is the NULL pointer.

17.3 internal.c

The file internal.c defines utility functions useful for operating on internal representations. The following functions are defined:

- `FreeInternal(p)`: When the parser constructs the internal representation, it calls the malloc() library function to allocate memory for its nodes. FreeInternal() frees the memory thus allocated, by recursing down the tree structure and calling free() for its various pointers.
- `PrintInternal(p)`: This prints out a textual description of the internal representation. I implemented this one mainly for debugging purposes, to determine if the parser was working as expected.

Chapter 18

Shell error handling

18.1 Purpose

I conclude this brief excursion into shell design with a very brief description of the shell's error handling mechanisms, which are implemented in errors.c. The purpose of this module is to keep all error-related functions in one place, so that the shell's reactions to exceptional cases may be changed without having to alter every module.

18.2 Interface and implementation

The interface consists of the following two functions:

- `MemoryError()`: This function is called by the parser when the system runs out of memory, i.e. the malloc() library function returned NULL. In this case there is no reasonable alternative but to print out an error message and exit the current shell instance. This is due to the fact that the shell allocates memory as sparingly as possible, i.e. all the allocated memory is indeed needed, so the shell cannot solve memory shortage by freeing unused memory blocks.
- `ParseError(msg)`: This is called by the parser when it has encountered a syntax error, such as a closing brace without a corresponding opening one. The function prints an appropriate error message, calls the SkipLine() function to cause the scanner to skip ahead to the next line, then returns the parser to its initial state. Basically, the parser "forgets" what it was doing and restarts its parsing process (see below for how this is accomplished). The string msg is printed out with `%s` replaced by a textual description of the current token type. A reasonable value for msg would be:

  ```
  "Found %s where fi was expected."
  ```

The function ParseError() is an interesting example of when setjmp() and longjmp() (see chapter 8) come in handy. Without these functions the recursive descent parser would have no way of jumping out of its hierarchy of nested

function calls back into the main driving loop. (The call to setjmp() takes place in the Parser() function in parser.c, and the corresponding longjmp() is located in errors.c.)

Index

Bibliography

[1] Helmut Herold: Linux- Unix- Systemprogrammierung, Second edition, ISBN: 3827315123

[2] Brian W. Kernighan, David Ritchie: C Programming Language, Second edition, Prentice Hall PTR, 1988, ISBN: 0131103628

[3] Robert Sedgewick, Philippe Flajolet, Peter Gordon: An Introduction to the Analysis of Algorithms, Addison-Wesley Professional, 1996, ISBN: 020140009X

[4] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers, Addison-Wesley, 1986, ISBN: 0201100886

[5] P. J. Plauger, Jim Brodie: Standard C (http://www-ccs.ucsd.edu/c/)

[6] Random Generators Homepage, mathematics department, University of Salzburg (http://random.mat.sbg.ac.at/)