Quick ISO C

Reference Manual

Sandro Sigala

To my family

Contents

List of Figures

List of Tables

Preface

References to the ISO C Normative Addendum 1 document are marked by a marginal boxed note at the left of the even pages, and at the right of the odd pages. The same thing applies for references to the ISO C Technical Corrigendum 1 and 2 documents.

⇨NA

⇨TC

This is the format used for the entries documented in the **ISO C Library** chapter, at page 11:

Prototype

`#include <stdlib.h>`	*the header to include*
`int abs(int x);`	*the prototype of the function*

Description

This paragraph describes the function.

This function returns the absolute value of the argument integer `x`. If the argument value is positive, `abs` returns it unmodified, otherwise if the value is negative, `abs` returns it with the positive sign.

Implementation

This paragraph illustrates a working implementation of the function, in this example the `abs` *function. The implementation examples are provided for the portable functions.*

```
#include <stdlib.h>

int abs(int x)
{
    return (x < 0) ? -x : x;
}
```

Example

This paragraph illustrates an example of usage of the function.

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

int main(void)
{
    printf("abs(%d) -> %d\n", 0, abs(0));
    printf("abs(%d) -> %d\n", 1, abs(1));
    printf("abs(%d) -> %d\n", -1, abs(-1));

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.10.6.1 *the reference to the standard document*

Entries in the ISO C Library chapter are ordered by type: first there are the constant macros, then the `typedef`-defined types, then the structure or union tags, and then the functions. Each type of entry is written in alphabetical order, excluding the functions that are ordered by behavior.

All the examples included in this book where tested with the following products:

- Free Software Foundation's GNU CC compiler [SF+98];
- UK Defence Evaluation and Research Agency TenDRA compiler [EA98];
- Christopher W. Fraser and David R. Hanson retargetable C compiler [FH95, FH98].

This book was typeset with the excellent LaTeX document preparation system. All the graphics where done with `xfig` and `gnuplot`.

Chapter 1

The ISO C Language

1.1 A bit of history

This section describes in summary the history of the C programming language. While reading the following text, please take a look at the figure below for a graphical representation of the events.

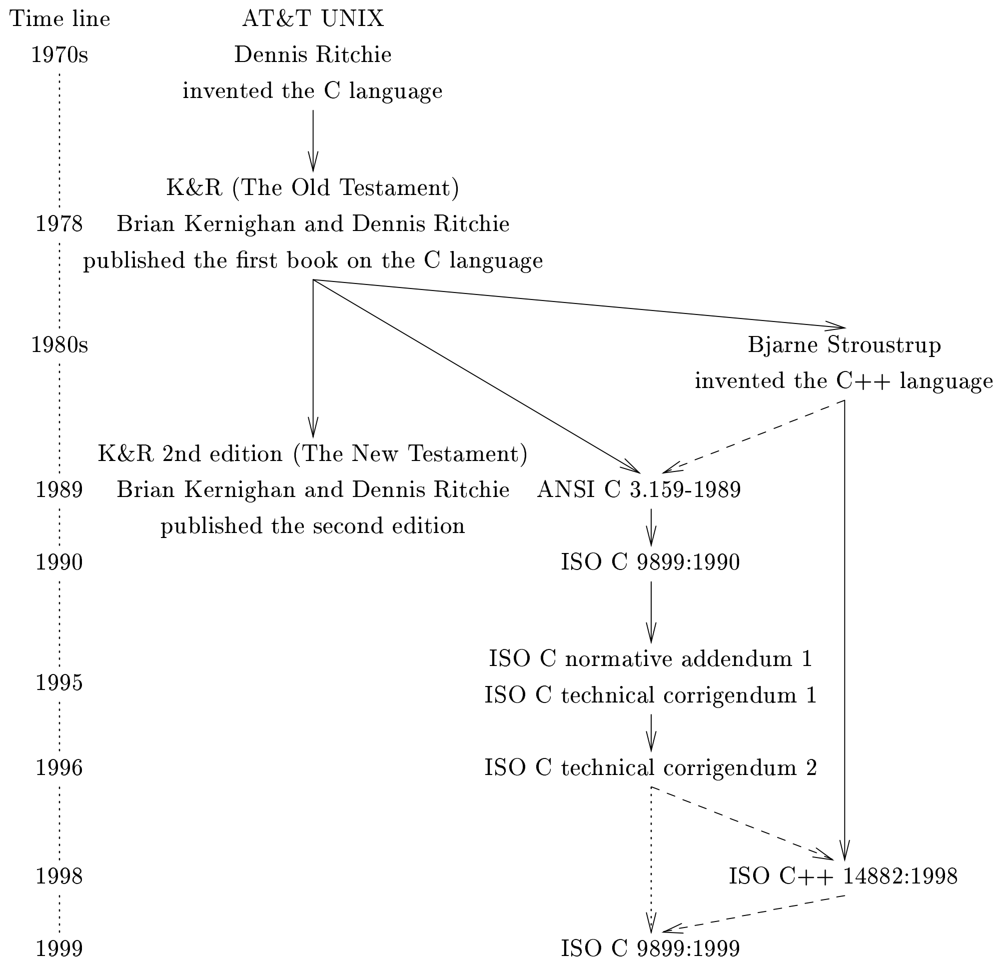

Figure 1.1: The C language history.

The C programming language was first invented by Dennis M. Ritchie in the 1972, while working at the AT&T Bell Laboratories. It was implemented under UNIX on a PDP-11 machine, during the development of release III of the operating system. It directly derives from the B programming language, developed by Ken Thompson for the first UNIX operating system, in the 1970. The B language derives itself from the BCPL language, developed by Martin Richards.

The first public book on C, *The C Programming Language* [KR78], was written by Dennis Ritchie and Brian Kernighan in 1978. During the 1980s, Bjarne Stroustrup, a computer scientist working for AT&T, modified the C language syntax getting some ideas from strong object-oriented languages like Simula and Smalltalk; the C++ language was born [Str97]. From the year of the publication of the first book on C, also called the Old Testament, other implementations of the language were written and C developers grew constantly. It was clear that a standardization was necessary to define the precise language to be used by the various mostly incompatible implementations of the language. In the 1983, the X3J11 committee was established at the American National Standards Institute (ANSI) for writing a definition of the C language not ambiguous and not machine dependent.

The standard was approved only six years later, in 1989, as the ANSI C 3.159-1989 standard [ANS89] (also known as C89). Some months before the approbation of the standard, the second edition of *The C Programming Language* [KR88] (also known as the New Testament) was published.

In the 1990, the International Standards Institute (ISO), as the result of the work of the JTC1/SC22/WG14 working group, ratified the ANSI standard producing the ISO C 9899:1990 document [ISO90] (also known as C90), introducing only minor modifications, like changing *sections* into *clauses* and renumbering the sections.

In the 1995, the ISO C standard was first modified by the ISO C Normative Addendum 1, addressing internationalization issues providing a complete support for wide characters and multibyte strings, then by the ISO C Technical Corrigendum 1 and 2, addressing technical inconsistencies and difficulties in interpretation of the standard. ⇨NA ⇨TC

In the 1998, the work of the ANSI X3J16 committee and the ISO JTC1/SC22/WG21 working group culminated in the ISO C++ 14882:1998 standard (see §3.2 on page 114).

In the 1999, the work of the ISO JTC1/SC22/WG14 working group culminated in the ISO C 9899:1999 document (see §3.1 on page 110).

1.1.1 The ISO C Normative Addendum 1

The ISO C Normative Addendum, approved in 1995, tries to complete the language support for internationalization, adding several functions to the library and support for respelling of tokens. ⇨NA

When the draft ANSI C standard was being considered by the ISO working group for adoption as international standard in 1990, there were several objections because it didn't address internationalization issues. The ISO working group then decided to provide the basis only for internationalization, and work would be carried out separately to provide a complete support for wide characters and multibyte strings.

Alternative token respellings (see §1.4.2 on page 9) and the `<iso646.h>` header (see §2.6 on page 32) were added to permit writing C code in national variants of the ISO 646 character set that don't support some of the C language symbols. Two new headers were added to the standard library, `<wchar.h>` and `<wctype.h>` (see §2.18 on page 100 and §2.19 on page 103, respectively), adding a complete support to wide characters and multibyte strings.

1.1.2 The ISO C Technical Corrigendum 1 and 2

⇨TC The ISO C Technical Corrigendum 1 and 2 are two documents that introduce modifications to the ISO C standard. They were reviewed and voted on by the ISO working group members after the JTC1/SC22/WG14 committee of ISO have received a number of *Defect Reports* from users that found technical inconsistencies and difficulties in interpretation of the standard. The Technical Corrigendum 2, approved in 1995, introduces only minor modifications to the standard, and should be of interest mostly to compiler and test suite vendors, while the Technical Corrigendum 1, approved in 1996, should be of interest also to C programmers.

1.2 The standard translation phases

A standard C translation is composed by eight phases (implementations must behave as if these separate phases occur), as summarized by the figure below:

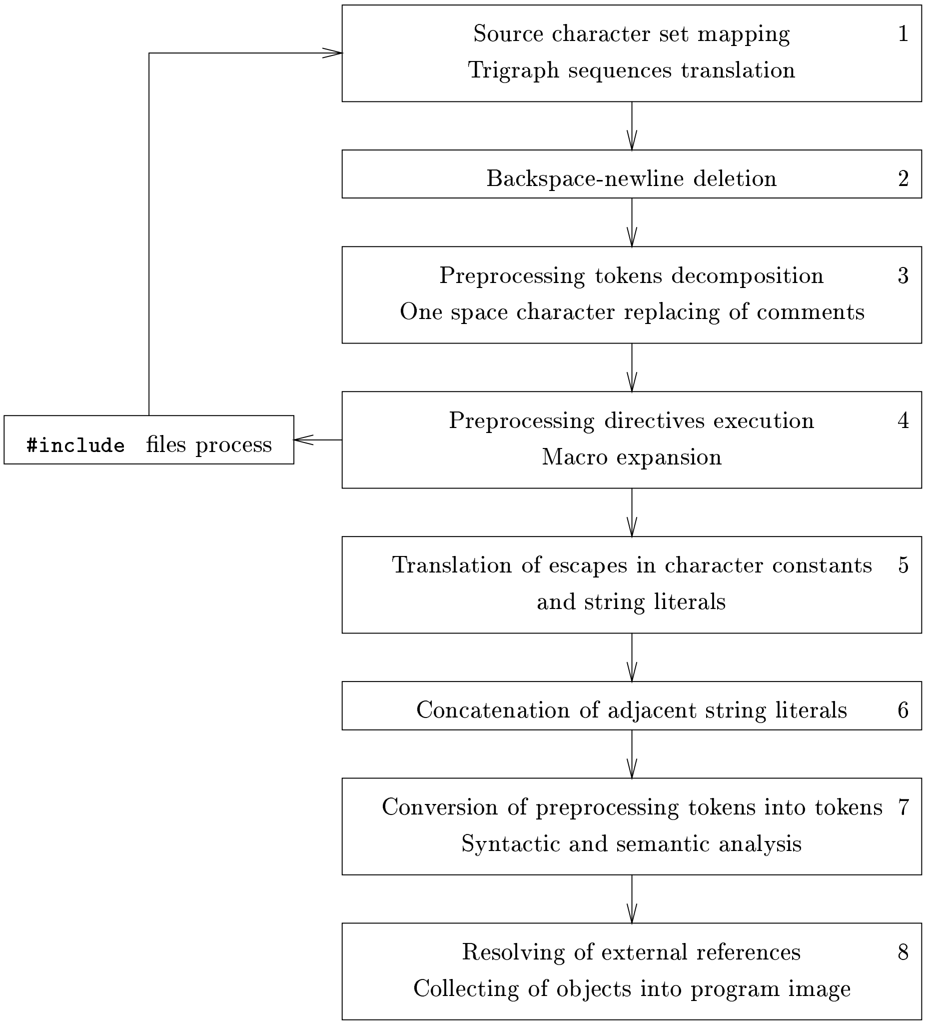

Figure 1.2: The translation phases.

1. The physical source file characters are mapped to the source character set if necessary. Trigraph sequences are replaced by corresponding single-character internal representations (like `??-` to `~`).
2. Each instance of a backslash character followed by new-line character is deleted, joining the two physical lines into a single logical line.
3. The source file is decomposed into preprocessing tokens and sequences of white-space characters. Each comment is replaced by one space character.
4. The preprocessing directives are executed and macro invocations are expanded. A `#include` preprocessing directive causes the specified source file to be processed from phase 1 through phase 4, recursively (see the figure).
5. Escape sequences in character constants and string literals are converted to the effective character values.
6. Adjacent string literals are concatenated and adjacent wide string literals are concatenated.
7. Each preprocessing token is converted into a token. Syntactic and semantic analysis occur.
8. All external object references are resolved and the external components (relocatable objects and libraries) are linked with the translation output into a single program image.

1.3 The character set

The C language uses a set of symbols for representing programs. This includes the visible printing characters, listed in the table below, plus the space character, and control

<table>
<tr><td>26 uppercase letters of the English alphabet</td></tr>
<tr><td>A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z</td></tr>
<tr><td>26 lowercase letters of the English alphabet</td></tr>
<tr><td>a b c d e f g h i j k l m
n o p q r s t u v w x y z</td></tr>
<tr><td>10 decimal digits</td></tr>
<tr><td>0 1 2 3 4 5 6 7 8 9</td></tr>
<tr><td>29 graphic characters</td></tr>
<tr><td>! " # % & ' ( ) * + , - . / :
; < = > ? [\ ] ^ _ { | } ~</td></tr>
</table>

Table 1.1: The C language character set.

characters representing horizontal tab, vertical tab, and form feed. In some implementations there shall be control characters representing alert, backspace, carriage return, and new line.

1.3.1 The trigraph sequences

In environments where some of the symbols of the C character set are not available, the trigraph sequences can be used as alternatives to the required symbols. The trigraph sequences are replaced in the first phase of translation (see §1.2 on page 5).

For example, the following code:

```
void f(void)
??<
    char s??(??) = "Huh";
    printf("%s???/n", s);
??>
```

is translated into:

```
void f(void)
{
    char s[] = "Huh";
    printf("%s?\n", s);
}
```

TRIGRAPH	REPLACEMENT
`??=`	`#`
`??(`	`[`
`??/`	`\`
`??)`	`]`
`??'`	`^`
`??<`	`{`
`??!`	`|`
`??>`	`}`
`??-`	`~`

Table 1.2: The trigraph sequences.

Sequences of `?` characters that do not begin one of the trigraphs listed in the above table are not changed.

1.3.2 The escape sequences

In C, the control characters can be embedded in string literals and character constants using the escape sequences. The translation into the equivalent control character values is done in the fifth phase of translation (see §1.2 on page 5).

ESCAPE	DESCRIPTION
`\a` (*alert*)	Produces an audible or visible alert.
`\b` (*backspace*)	Moves the output cursor to the previous column on the current line.
`\f` (*form feed*)	Moves the output cursor to the next logical line.
`\n` (*new line*)	Moves the output cursor to the beginning of the next line.
`\r` (*carriage return*)	Moves the output cursor to the beginning of the current line.
`\t` (*horizontal tab*)	Moves the output cursor to the next horizontal tabulation position on the current line.
`\v` (*vertical tab*)	Moves the output cursor to the next vertical tabulation position.

Table 1.3: The escape sequences.

1.4 The language

1.4.1 The keywords

The C language provides 32 keywords, as written in the table below:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 1.4: The C keywords.

These tokens are reserved for use as keywords, and shall not be used otherwise.

1.4.2 The alternative tokens

The ISO C Normative Addendum 1 (see §1.1.1 on page 3) added six new tokens (often called *digraphs*) that behave identically to six previous existing tokens, except that they are spelled differently, as listed in the table below: ⇨NA

Alternative	Primary
<:	[
:>	]
<%	{
%>	}
%:	#
%:%:	##

Table 1.5: The alternative token representations.

They were added for improving the appearance of C programs written in national variants of the ISO 646 character set where some of the symbols of the C character set are not available.

1.5 To be used

Macro name	Description
`__STDC__`	The value `1`
`__STDC_VERSION__`	The value `199409L`
`__DATE__`	The date of compilation
`__FILE__`	The current file name
`__LINE__`	The current line number

Table 1.6: The preprocessor predefined macros.

Chapter 2

The ISO C Library

2.1 Introduction

The standard C library is composed by 18 headers, as summarized by the table below:

<assert.h>	Debug assertions (§2.2)
<ctype.h>	Character class checking and conversions (§2.3)
<errno.h>	Error checking facilities (§2.4)
<float.h>	Floating point limits (§2.5)
<iso646.h>	Limited character set implementations support (§2.6)
<limits.h>	Implementation limits (§2.7)
<locale.h>	Localization functions (§2.8)
<math.h>	Mathematical functions (§2.9)
<setjmp.h>	Jumps between functions (§2.10)
<signal.h>	Signal handling functions (§2.11)
<stdarg.h>	Macros for accessing variable number arguments (§2.12)
<stddef.h>	Standard types an macros (§2.13)
<stdio.h>	Standard input and output functions (§2.14)
<stdlib.h>	General and miscellaneous functions (§2.15)
<string.h>	String handling functions (§2.16)
<time.h>	Time and date types and functions (§2.17)
<wchar.h>	Wide characters types and conversions (§2.18)
<wctype.h>	Wide characters class checking and conversions (§2.19)

Table 2.1: The standard C library headers.

The <iso646.h> ,<wchar.h>, and <wctype.h> headers where introduced by the ⇨NA ISO C Normative Addendum 1 (see §1.1.1 on page 3) for providing a complete support for wide characters and multibyte strings.

Some functions may be implemented using macro equivalents for speed reasons; several methods exist for calling the effective functions:

- by undefining the macro

```
#include <stdlib.h>
#undef atoi

int i;
const char *str;
/* ... */
i = atoi(str);
```

- by avoiding the macro expansion

```
#include <stdlib.h>

int i;
const char *str;
/* ... */
i = (atoi)(str);
```

- by explicit declaration

```
extern int atoi(const char *s);

int i;
const char *str;
/* ... */
i = atoi(str);
```

- by implicit declaration

```
int i;
const char *str;
/* ... */
i = atoi(str);
```

2.2 The <assert.h> header

The `<assert.h>` header declares the `assert` macro useful in the debugging phase of software development. The behaviour of the `assert` macro changes whether the `NDEBUG` macro is defined before the inclusion of the `<assert.h>` header.

The `<assert.h>` header declares 1 symbol and checks for 1 macro, as summarized by the table below:

Macros	
`NDEBUG`	Exclude the assertion tests (§2.2.1)
`assert`	Specify a run-time debug test (§2.2.2)

Table 2.2: The `<assert.h>` symbols.

2.2.1 NDEBUG

Prototype

```
#define NDEBUG
#include <assert.h>
```

Description

The `NDEBUG` macro is the sole symbol that is not defined by the standard C library, but may be defined by the user before including the `<assert.h>` header. This is useful after the editing-debug phase of the software development, when the user have assured that the various `assert` checks are never false. When the `NDEBUG` macro is defined before the inclusion of the `<assert.h>` header, the `assert` macro is defined to a null value like this:

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

All the debugging `assert` checks are then avoided.

Example

```
#define NDEBUG          /* Debug phase finished. */
#include <assert.h>
#include <stdlib.h>
#include <string.h>

/* See assert() for the definition. */
extern void *xmalloc(size_t size);
```

```
/*
 * Return a duplicate of the argument string, allocating
 * dynamically a memory area.
 */
char *xstrdup(const char *s)
{
    char *ns;

    assert(s != NULL);  /* Catch invalid arguments. */
    ns = (char *)xmalloc(strlen(s) + 1);
    assert(ns != NULL); /* Catch unexpected behaviors. */
    strcpy(ns, s);

    return ns;
}
```

References

ISO C 9899:1990 §7.2, ISO C 9899:1999 §7.2

2.2.2 assert

Prototype

```
#include <assert.h>

void assert(int expression);
```

Description

Prints an error message to standard output and terminates the program by calling the `abort` function if the argument `expression` is false. This only happens when the macro `NDEBUG` is undefined before the inclusion of the `<assert.h>` header, usually during the editing-debug phase of the software development. The `assert` argument should not have side-effects, because the program behavior will be different depending whether `NDEBUG` is defined, as in this example:

```
char *s;
assert((s = (char *)malloc(FILENAME_MAX)) != NULL);
strcpy(s, otherstr);
```

Implementation

Implementation of the `<assert.h>` standard header:

```
#undef assert
```

```
#ifdef NDEBUG
#define assert(e)  ((void)0)
#else
void __assert_error(char *expr, char *filename, unsigned int line);
#define assert(e)  ((void)(e || __assert(#e, __FILE__, __LINE__)))
```

Implementation of the `__assert_error` function referenced by the `assert` macro in the <assert.h> header:

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

void __assert_error(char *expr, char *filename, unsigned int line)
{
    fprintf(stderr, "Assertion failed: %s, file %s, line %d\n",
            expr, filename, line);
    abort();
}
```

Example

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * Return a pointer to an allocated region of memory of the
 * specified size.  Check for unexpected errors.
 */
void *xmalloc(size_t size)
{
    void *p;

    assert(size > 0);
    if ((p = malloc(size)) == NULL) {
        fprintf(stderr, "No more available memory\n");
        exit(1);
    }

    return p;
}
```

References

ISO C 9899:1990 §7.2.1.1, ISO C 9899:1999 §7.2.1.1

2.3 The <ctype.h> header

The `<ctype.h>` header declares several functions useful for checking and mapping characters. The accepted argument of the functions is an `int`, representable as an `unsigned char`, or the value of the macro EOF. All the following functions may be implemented as macros.

In the following sections, the term *printing character* means a character that occupies one character position on the screen; the term *control character* means a character not a printing character.

The `<ctype.h>` header declares 13 symbols, as summarized by the table below:

FUNCTIONS	
`isalnum`	Check if a character is an alphabetic or a digit character (§2.3.1)
`isalpha`	Check if a character is an alphabetic character (§2.3.2)
`iscntrl`	Check if a character is a control character (§2.3.3)
`isdigit`	Check if a character is a digit (§2.3.4)
`isgraph`	Check if a character is a graphic character (§2.3.5)
`islower`	Check if a character is a lowercase character (§2.3.6)
`isprint`	Check if a character is a printable character (§2.3.7)
`ispunct`	Check if a character is a punctuation character (§2.3.8)
`isspace`	Check if a character is a space character (§2.3.9)
`isupper`	Check if a character is an uppercase character (§2.3.10)
`isxdigit`	Check if a character is an hexadecimal digit (§2.3.11)
`tolower`	Convert a character to a lowercase character (§2.3.12)
`toupper`	Convert a character to an uppercase character (§2.3.13)

Table 2.3: The `<ctype.h>` symbols.

2.3.1 `isalnum`

Prototype

```
#include <ctype.h>

int isalnum(int c);
```

Description

Returns a true value for any character for which `isalpha` or `isdigit` is true.

Example

```
#include <ctype.h>
```

```
/*
 * Return a true value if the argument string is a regular C
 * identifier.
 */
int is_identifier(const char *s)
{
    if (!isalpha(*s) && *s != '_')
        return 0;
    for (++s; *s != '\0'; ++s)
        if (!isalnum(*s) && *s != '_')
            return 0;
    return 1;
}
```

References

ISO C 9899:1990 §7.3.1.1

2.3.2 isalpha

Prototype

```
#include <ctype.h>

int isalpha(int c);
```

Description

In the "C" locale, returns a true value for the characters for which `isupper` or `islower` is true. In other implementation-defined set of characters returns true for the characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true.

Example

```
#include <ctype.h>
#include <stdio.h>

/*
 * Return the next word read the standard input.
 */
char *get_word(char *buf)
{
    char *dp = buf;
    int c;

    while (isalpha(c = getchar()))
```

```
        *dp++ = c;
    if (c != EOF)
        ungetc(c, stdin);
    *dp = '\0';

    return buf;
}
```

References

ISO C 9899:1990 §7.3.1.2

2.3.3 iscntrl

Prototype

```
#include <ctype.h>

int iscntrl(int c);
```

Description

Returns a true value if the `c` character is a control character.

Example

```
#include <ctype.h>
#include <stdio.h>

/*
 * Get the next character from the standard input skipping the
 * control characters.
 */
int xgetchar(void)
{
    int c;
    while (iscntrl(c = getchar()))
        ;
    return c;
}
```

References

ISO C 9899:1990 §7.3.1.3

2.3.4 isdigit

Prototype

```
#include <ctype.h>
```

```
int isdigit(int c);
```

Description

Returns a true value if the `c` character is a decimal-digit character.

Example

```
#include <ctype.h>

/*
 * Return a true value if the argument string is a regular
 * decimal number.
 */
int is_number(const char *s)
{
    const char *sp;
    for (sp = s; *sp != '\0'; ++sp)
        if (!isdigit(*sp))
            return 0;
    return sp > s;
}
```

References

ISO C 9899:1990 §7.3.1.4

2.3.5 isgraph

Prototype

```
#include <ctype.h>

int isgraph(int c);
```

Description

Returns a true value if the `c` character is a printing character except the space character (`' '`).

Example

```
#include <ctype.h>

/*
 * Remove the non-graphic characters from the argument string.
 */
char *filter_graphs(char *s)
{
```

```
    char *sp = s;

    while (*sp != '\0')
        if (!isgraph(*sp))
            *sp = *(sp + 1);
        else
            ++sp;

    return s;
}
```

References

ISO C 9899:1990 §7.3.1.5

2.3.6 islower

Prototype

```
#include <ctype.h>

int islower(int c);
```

Description

In the "C" locale, returns a true value for the characters defined as lowercase letters. In other implementation-defined set of characters returns true for the characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true.

Example

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * Count the lowercase, uppercase, and other characters read from
 * standard input.
 */
int main(void)
{
    int lower = 0, upper = 0, other = 0;
    int c;

    while ((c = getchar()) != EOF)
        if (islower(c))
            ++lower;
```

```
        else if (isupper(c))
            ++upper;
        else
            ++other;

    printf("Lower case: %d, upper case: %d, other: %d\n",
           lower, upper, other);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.3.1.6

2.3.7 isprint

Prototype

```
#include <ctype.h>

int isprint(int c);
```

Description

Returns a true value if the `c` character is a printing character including the space character (' ').

Example

```
#include <ctype.h>
#include <stdio.h>

/*
 * Print the argument character if it is a printable character,
 * otherwise its escape representation.
 */
void prchar(int c)
{
    if (isprint(c))
        putchar(c);
    else
        switch(c) {
        case '\a': putchar('\\'); putchar('a'); break;
        case '\b': putchar('\\'); putchar('b'); break;
        case '\f': putchar('\\'); putchar('f'); break;
        case '\n': putchar('\\'); putchar('n'); break;
```

```
        case '\r': putchar('\\'); putchar('r'); break;
        case '\t': putchar('\\'); putchar('t'); break;
        case '\v': putchar('\\'); putchar('v'); break;
        default:
            printf("\\%o", c & 255);
        }
}
```

References

ISO C 9899:1990 §7.3.1.7

2.3.8 ispunct

Prototype

```
#include <ctype.h>

int ispunct(int c);
```

Description

Returns a true value if the `c` character is a printing character neither a space character (`' '`) nor a character for which `isalnum` is true.

Example

```
#include <ctype.h>

/*
 * Replace punctuation characters with spaces.
 */
char *replace_puncts(char *s)
{
    char *sp;
    for (sp = s; *sp != '\0'; ++sp)
        if (ispunct(*sp))
            *sp = ' ';
    return s;
}
```

References

ISO C 9899:1990 §7.3.1.8

2.3.9 isspace

Prototype

```
#include <ctype.h>
```

```
int isspace(int c);
```

Description

In the "C" locale, returns a true value if the `c` character is a standard white-space character (space `' '`, form feed `'\f'`, new line `'\n'`, carriage return `'\r'`, horizontal tab `'\t'`, and vertical tab `'\v'`). In other implementation-defined set of characters returns true for the characters for which `isalnum` is false.

Example

```
#include <ctype.h>
#include <stdio.h>

/*
 * Get the next token separated by white spaces from the
 * standard input.
 */
char *gettoken(char *buf)
{
    char *dp = buf;
    int c;

    while (isspace(c = getchar()))
        ;       /* Skip white spaces. */
    if (c == EOF)
        return NULL;
    do
        *dp++ = c;
    while (!isspace(c = getchar()));
    *dp = '\0';

    return buf;
}
```

References

ISO C 9899:1990 §7.3.1.9

2.3.10 isupper

Prototype

```
#include <ctype.h>

int isupper(int c);
```

Description

In the "C" locale, returns a true value for the characters defined as uppercase letters. In other implementation-defined set of characters returns true for the characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true.

Example

```
#include <ctype.h>

/*
 * Return a true value if the argument string is a word in the
 * [A-Z][A-Za-z]* format, i.e. the first character is uppercase,
 * the others, if exist, uppercase or lowercase.
 */
int is_uword(const char *s)
{
    if (!isupper(*s))
        return 0;
    for (++s; *s != '\0'; ++s)
        if (!isalpha(*s))
            return 0;
    return 1;
}
```

References

ISO C 9899:1990 §7.3.1.10

2.3.11 isxdigit

Prototype

```
#include <ctype.h>

int isxdigit(int c);
```

Description

Returns a true value if the `c` character is an hexadecimal-digit character.

Example

```
#include <ctype.h>

/*
 * Translate the hexadecimal argument string into an integer value.
 */
int hex_to_int(const char *s)
{
```

```
    int i = 0;

    for (; *s != '\0'; ++s) {
        if (!isxdigit(*s))
            return i;
        i *= 16;
        if (isdigit(*s))
            i += *s - '0';
        else
            i += tolower(*s) - 'a' + 10;
    }

    return i;
}
```

References

ISO C 9899:1990 §7.3.1.11

2.3.12 tolower

Prototype

```
#include <ctype.h>

int tolower(int c);
```

Description

Returns the lowercase character corresponding to the character `c`, if the value of `isupper` with the argument `c` is true, otherwise returns the `c` argument unchanged.

Example

```
#include <ctype.h>

/*
 * Compare the two strings ignoring the characters case.
 */
int istrcmp(const char *s1, const char *s2)
{
    const unsigned char *sp1 = (const unsigned char *)s1;
    const unsigned char *sp2 = (const unsigned char *)s2;

    for (; tolower(*sp1) == tolower(*s2); ++sp1, ++sp2)
        if (*sp1 == '\0')
            return 0;
```

```
    return tolower(*sp1) - tolower(*sp2);
}
```

References

ISO C 9899:1990 §7.3.2.1

2.3.13 toupper

Prototype

```
#include <ctype.h>

int toupper(int c);
```

Description

Returns the uppercase character corresponding to the character `c`, if the value of `islower` with the argument `c` is true, otherwise returns the `c` argument unchanged.

Example

```
#include <ctype.h>

/*
 * Capitalize the argument word.
 */
char *capitalize_word(char *s)
{
    char *sp;

    if (*s == '\0')
        return s;
    *s = toupper(*s);
    for (sp = s + 1; *sp != '\0'; ++sp)
            *sp = tolower(*sp);

    return s;
}
```

References

ISO C 9899:1990 §7.3.2.2

2.4 The <errno.h> header

The `<errno.h>` header declares the `errno` symbol, that can be used to determine what went wrong after calling a library function that returned an error value. Three macro constants are also defined to match three common C errors; additional macro definitions, beginning with `E` and a digit or `E` and a uppercase letter, may be also specified by the implementation.

The `<errno.h>` header declares 4 symbols, as summarized by the table below:

Macros	
`EDOM`	*Domain error* constant (§2.4.1)
`EILSEQ`	*Multibyte sequence encoding error* constant (§2.4.2)
`ERANGE`	*Result too large error* constant (§2.4.3)
Lvalues	
`errno`	Last error number (§2.4.4)

Table 2.4: The `<errno.h>` symbols.

2.4.1 EDOM

Prototype

```
#include <errno.h>

#define EDOM    implementation-defined
```

Description

The `EDOM` macro expands to a constant value that is stored in `errno` after a *domain error* occurred. This type of error occurs when a mathematical function argument is outside the domain over which the mathematical function is defined. For example, a domain error occurs for `acos` arguments not in the range $[-1, +1]$.

References

ISO C 9899:1990 §7.1.4

2.4.2 EILSEQ

Prototype

```
#include <errno.h>

#define EILSEQ implementation-defined
```

Description

The `EILSEQ` macro expands to a constant value that is stored in `errno` after a *multibyte encoding error* occurred. This type of error occurs if a character presented to a wide-character input/output function does not correspond to a valid (generalized) multibyte character or if a character sequence does not form a valid (generalized) multibyte character. The `EILSEQ` macro was introduced by the ISO C Normative Addendum 1 (see §1.1.1 on page 3). ⇨NA

References

ISO C 9899:1990 §7.1.4

2.4.3 `ERANGE`

Prototype

```
#include <errno.h>

#define ERANGE implementation-defined
```

Description

The `ERANGE` macro expands to a constant value that is stored in `errno` after a *range error* occurred. This type of error occurs when the result of a mathematical function cannot be represented as a `double` value.

References

ISO C 9899:1990 §7.1.4

2.4.4 `errno`

Prototype

```
#include <errno.h>

#define errno   implementation-defined
or
extern int errno;
```

Description

The `errno` symbol holds the last error number set by several library functions. It may be implemented as a macro or as an identifier with external linkage. Please note that `errno` value is set to zero only at startup, and the user should set it to zero before a library function call, then inspect it before a subsequent library function call.

Example

```
#include <errno.h>
#include <stdio.h>
```

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

int main(void)
{
    FILE *f;

    /* Try to open an unexistent file. */
    if ((f = fopen("noexist.txt", "r")) == NULL) {
            fprintf(stderr, "Cannot open: error number %d: %s\n",
                    errno, strerror(errno));
            return EXIT_FAILURE;
    }
    fclose(f);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.1.4

2.5 The <float.h> header

2.6 The <iso646.h> header

⇨NA The <iso646.h> header was introduced by ISO C Normative Addendum 1 (see §1.1.1 on page 3) in the ISO C standard. It defines 11 macros mostly useful to write in a restricted source character set. The table below contains the macro names and the corresponding values.

<table>
<tr><th>Macro name</th><th>Value</th></tr>
<tr><td>and</td><td>&&</td></tr>
<tr><td>and_eq</td><td>&=</td></tr>
<tr><td>bitand</td><td>&</td></tr>
<tr><td>bitor</td><td>|</td></tr>
<tr><td>compl</td><td>~</td></tr>
<tr><td>not</td><td>!</td></tr>
<tr><td>not_eq</td><td>!=</td></tr>
<tr><td>or</td><td>||</td></tr>
<tr><td>or_eq</td><td>|=</td></tr>
<tr><td>xor</td><td>^</td></tr>
<tr><td>xor_eq</td><td>^=</td></tr>
</table>

Table 2.5: The <iso646.h> symbols.

For example, the previously defined code:

```
#include <ctype.h>

int is_identifier(char *s)
{
    if (!isalpha(*s) && *s != '_')
        return 0;
    for (++s; *s != '\0'; ++s)
        if (!isalnum(*s) && *s != '_')
            return 0;
    return 1;
}
```

in environments with a restricted source character set, where the ! and & characters are not available, can be easily written as:

```
#include <ctype.h>
#include <iso646.h>
```

```
int is_identifier(char *s)
{
    if (not isalpha(*s) and *s not_eq '_')
        return 0;
    for (++s; *s not_eq '\0'; ++s)
        if (not isalnum(*s) and *s not_eq '_')
            return 0;
    return 1;
}
```

2.7 The <limits.h> header

2.8 The `<locale.h>` header

The `<locale.h>` header declares 10 symbols, as summarized by the table below:

MACROS	
`LC_ALL`	All the locale categories (§??)
`LC_COLLATE`	The string functions category (§??)
`LC_CTYPE`	The character classification and conversion category (§??)
`LC_MONETARY`	The formatted monetary information category (§??)
`LC_NUMERIC`	The formatted numeric information category (§??)
`LC_TIME`	The date and time category (§??)
`NULL`	The null pointer constant (§2.13.1)
STRUCTURES	
`lconv`	Structure holding numeric formatting information (§??)
FUNCTIONS	
`setlocale`	Select the locale setting (§??)
`localeconv`	Returns the current locale formatting information (§??)

Table 2.6: The `<locale.h>` symbols.

2.8.1 `NULL`

See §2.13.1 on page 50.

2.9 The <math.h> header

The `<math.h>` header declares 23 symbols, as summarized by the table below:

MACROS	
`HUGE_VAL`	Value returned on overflow (§??)
FUNCTIONS	
`sin`	Compute the trigonometric sine (§2.9.2)
`cos`	Compute the trigonometric cosine (§??)
`tan`	Compute the trigonometric tangent (§??)
`acos`	Compute the trigonometric arc cosine (§??)
`asin`	Compute the trigonometric arc sine (§??)
`atan`	Compute the trigonometric arc tangent (§??)
`atan2`	Compute the trigonometric arc tangent (§??)
`sinh`	Compute the hyperbolic sine (§??)
`cosh`	Compute the hyperbolic cosine (§??)
`tanh`	Compute the hyperbolic tangent (§??)
`exp`	Compute the exponential function (§??)
`log`	Compute the natural logarithm (§??)
`log10`	Compute the base-ten logarithm (§??)
`pow`	Compute the power of a number (§2.9.4)
`sqrt`	Compute the square root (§2.9.3)
`fabs`	Compute the absolute value of a number (§??)
`ceil`	Round upwards to the nearest integer (§??)
`floor`	Round downwards to the nearest integer (§??)
`fmod`	Compute the remainder of a fraction (§??)
`modf`	Break a number into integral and fractional parts (§??)
`ldexp`	Multiply a number by a power of 2 (§??)
`frexp`	Break a number into a normalized fraction and a power of 2 (§??)

Table 2.7: The `<math.h>` symbols.

2.9.1 Randians and degrees

The ISO C Library mathematical functions accept angles expressed in radians.

A full 360 degrees angle is equivalent to 2π radians. Converting between degrees and radians is trivial, using the formulas:

$$\text{rad} = \frac{2\pi}{360}\text{deg}$$

$$\text{deg} = \frac{360}{2\pi}\text{rad}$$

or, expressed in C:

```
#define PI      3.14159265358979323846

double deg2rad(double x)
{
    return PI/180 * x;
}

double rad2deg(double x)
{
    return 180/PI * x;
}
```

2.9.2 sin

Prototype

```
#include <math.h>

double sin(double x);
```

Description

Returns the sine of `x`, where `x` is given in radians. The return value is in the range $[-1, +1]$.

Example

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define PI      3.14159265358979323846
#define G       9.80665

/* Calculate initial speed. */
double initial_speed(double distance, double angle)
{
    return sqrt((.5 * G * distance * distance) /
                (tan(angle) * distance * pow(cos(angle), 2)));
}

/* Calculate fly time. */
```

```
double fly_time(double speed, double angle)
{
    return 2 * speed * sin(angle) / G;
}

/* Calculate maximum height. */
double maximum_height(doble speed, double angle)
{
    return pow(speed * sin(angle), 2) / (2 * G);
}

/*
 * cannon.c: calculate the initial speed, fly time and maximum
 *           height of a cannon ball that must hit a target at the
 *           specified distance.
 * usage: cannon <distance> [angle]
 */
int main(int argc, char **argv)
{
    double distance, angle, speed, time, height;

    if (argc < 2 || argc > 3) {
        fprintf(stderr, "usage: cannon <distance> [angle]\n"
                "\tdistance is expressed in meters\n"
                "\tangle is expressed in degrees "
                "(optional, default is 45)\n");
        return EXIT_FAILURE;
    }

    distance = atof(argv[1]);
    if (argc == 2)      /* No angle specified. */
        angle = 45;
    else
        angle = atof(argv[2]);

    /* Convert angle into radians. */
    angle = PI/180 * angle;
    /* Calculate values. */
    speed = initial_speed(distance, angle);
    time = fly_time(speed, angle);
    height = maximum_height(speed, angle);
```

```
    printf("distance: %.2f m, angle: %.2f rad\n"
           "initial speed: %.2f m/s, "
           "fly time: %.2f s, maximum height: %.2f m\n",
           distance, angle, speed, time, height);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §XXX

2.9.3 sqrt

Prototype

```
#include <math.h>

double sqrt(double x);
```

Description

Returns ...

Example

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define GAMMA   6.67259e-11

/* Calculate the minimum escape speed. */
double required_speed(double radius, double mass)
{
    return sqrt((2 * GAMMA * mass) / radius);
}

/*
 * escape.c: calculate the minimum speed required by a body to
 *           escape from the gravitational attraction of some
 *           known planets.
 */
int main(void)
{
    static const struct {
```

```
        char *name;
        double radius, mass;
    } planets[] = {
        /* Name      Radius   Mass    */
        { "Sun",     6.96e08, 1.99e30 },
        { "Mercury", 2.44e06, 3.30e23 },
        { "Venus",   6.05e06, 4.87e24 },
        { "Earth",   6.37e06, 5.97e24 },
        { "Mars",    3.39e06, 6.42e23 },
        { "Jupiter", 6.98e07, 1.90e27 },
        { "Saturn",  5.82e07, 5.68e26 },
        { "Uranus",  2.37e07, 8.70e25 },
        { "Neptune", 2.24e07, 1.03e26 },
        { "Pluto",   2.05e06, 6.63e23 },
        { "Moon",    1.74e06, 7.35e22 }
    };
    unsigned int i;
    double speed;

    for (i = 0; i < sizeof planets / sizeof planets[0]; ++i) {
        speed = required_speed(planets[i].radius, planets[i].mass);
        printf("Speed to escape from %-8s: %.3f km/h\n",
               planets[i].name, speed * 3.6);
    }

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §XXX

2.9.4 `pow`

Prototype

```
#include <math.h>

double pow(double x, double y);
```

Description

Returns ...

Example

```
#include <math.h>
```

```
/* Calculate the distance between two points. */
double distance(double x0, double y0, double x1, double y1)
{
        return sqrt(pow(x1 - x0, 2) + pow(y1 - y0, 2));
}
```

References

ISO C 9899:1990 §XXX

2.10 The <setjmp.h> header

The <setjmp.h> header declares 3 symbols, as summarized by the table below:

TYPES	
jmp_buf	Type holding environment information (§??)
FUNCTIONS	
setjmp	Save the calling environment (§??)
longjmp	Restore the saved environment (§??)

Table 2.8: The <setjmp.h> symbols.

2.11 The <signal.h> header

The <signal.h> header declares 12 symbols, as summarized by the table below:

MACROS	
SIG_DFL	Default signal handler (§??)
SIG_ERR	Error return value (§??)
SIG_IGN	Ignoring signal handler (§??)
SIGABRT	Abnormal termination signal (§??)
SIGFPE	Erroneous arithmetic operation signal (§??)
SIGILL	Illegal instruction signal (§??)
SIGINT	Interactive attention reception signal (§??)
SIGSEGV	Invalid access to storage signal (§??)
SIGTERM	Program termination request signal (§??)
TYPES	
sig_atomic_t	Type of an object accessible as an atomic entity (§??)
FUNCTIONS	
signal	Select a signal handling function (§??)
raise	Send a signal to the executing program (§??)

Table 2.9: The <signal.h> symbols.

2.12 The <stdarg.h> header

The `<stdarg.h>` header defines a type and three macros useful for accessing the variable arguments of a functions, i.e. the arguments witch type and number cannot be specified at compilation time, but can be determined at run-time. This type of functions can be declared using the ellipsis '`...`' token, as in the `printf` function prototype:

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

The `<stdarg.h>` header declares 4 symbols, as summarized by the table below:

MACROS	
va_start	Initialize the variable arguments parsing data (§2.12.1)
va_arg	Return the next function variable argument (§2.12.2)
va_end	Cleanup the variable arguments parsing data (§2.12.3)
TYPES	
va_list	Store the variable arguments data during parsing (§2.12.4)

Table 2.10: The `<stdarg.h>` symbols.

2.12.1 va_start

Prototype

```
#include <stdarg.h>

void va_start(va_list ap, paramN);
```

Description

The `va_start` macro initializes the variable arguments data used for scanning the list of arguments. The *paramN* parameter is the last parameter in the function argument list; for example in the following function:

```
void f(void *p, int i, ...);
```

where the `i` variable is the last argument, the `va_start` function should be called as:

```
va_list ap;
va_start(ap, i);
```

Example

```
#include <stdarg.h>
#include <stdio.h>

/*
 * Print the arguments plus their type.
 * The first argument ‘fmt’ specifies the type of the other
 * arguments:
 *   c means char
 *   d means int
 *   s means string (char *)
 * For example:
 *    print_types("cds", ’x’, 10, "abc");
 * Prints:
 *    char = ’x’
 *    int = 10
 *    string = "abc"
 */
void print_types(const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    for (; *fmt != ’\0’; ++fmt)
        switch (*fmt) {
        case ’c’:
            printf("char = ’%c’\n", va_arg(ap, char));
            break;
        case ’d’:
            printf("int = %d\n", va_arg(ap, int));
            break;
        case ’s’:
            printf("string = \"%s\"\n", va_arg(ap, char *));
            break;
        }
    va_end(ap);
}
```

References

ISO C 9899:1990 §7.8.1.1

2.12.2 va_arg

Prototype

```
#include <stdarg.h>

type va_arg(va_list ap, type);
```

Description

The `va_arg` macro gets the next argument, of type *type*, from the variable argument list `ap`. The `ap` variable is modified so that the next call returns the next argument. The `ap` parameter should be previously initialized with `va_start` and the *type* parameter should be of the same type as the user passed argument (or at least of the same size).

Example

```
#include <stdarg.h>
#include <stdio.h>

/*
 * Sum the argument integers.
 * For example:
 *     sum_args(4, 5, 3, 2, 1);
 * Returns:
 *     11
 */
int sum_args(int num_args, ...)
{
    va_list ap;
    int sum = 0;

    va_start(ap, num_args);
    while (num_args--)
        sum += va_arg(ap, int);
    va_end(ap);

    return sum;
}
```

References

ISO C 9899:1990 §7.8.1.2

2.12.3 va_end

Prototype

```
#include <stdarg.h>

void va_end(va_list ap);
```

Description

The va_end macro cleans up the variable arguments data used for scanning the list of arguments before returning the function. This should be called every time a va_start call appears in the same function. In some implementations, the va_end macro does nothing.

Example

```
#include <stdarg.h>
#include <stdio.h>

/*
 * Print the NULL terminated list of strings.
 * For example:
 *    print_str_list("Hell", "o Wor", "ld", NULL);
 * Prints:
 *    Hello World
 */
void print_str_list(char *s, ...)
{
    va_list ap;

    va_start(ap, s);
    while (s != NULL) {
        printf("%s", s);
        s = va_arg(ap, char *);
    }
    va_end(ap);
    fputc('\n', stdout);
}
```

References

ISO C 9899:1990 §7.8.1.3

2.12.4 va_list

Prototype

```
#include <stdarg.h>
```

```
typedef implementation-defined va_list;
```

Description

The `va_list` type is used to access the variable arguments of a function. It is required to scan all the arguments using the `va_arg` macro and should be initialized with `va_start` and cleaned up with `va_end` before returning from the function.

Example

```
#include <stdarg.h>
#include <stdio.h>

/*
 * A mini vfprintf() function.
 *
 * Accepts the following printf() formats:
 *   %%  prints %
 *   %c  prints a char
 *   %d  prints an int
 *   %s  prints a string (char *)
 */
void mini_vfprintf(FILE *stream, const char *fmt, va_list ap)
{
    for (; *fmt != '\0'; ++fmt)
        if (*fmt == '%')
            switch (*++fmt) {
            case '%':
                fputc('%', stream);
                break;
            case 'c':
                fprintf(stream, "%c", va_arg(ap, char));
                break;
            case 'd':
                fprintf(stream, "%d", va_arg(ap, int));
                break;
            case 's':
                fprintf(stream, "%s", va_arg(ap, char *));
                break;
            }
        else
            fputc(*fmt, stream);
```

```
}

/*
 * A mini fprintf() function.
 */
void mini_fprintf(FILE *stream, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    mini_vfprintf(stream, fmt, ap);
    va_end(ap);
}

/*
 * A mini printf() function.
 */
void mini_printf(const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    mini_vfprintf(stdout, fmt, ap);
    va_end(ap);
}
```

References

ISO C 9899:1990 §7.8

2.13 The <stddef.h> header

The `<stddef.h>` header declares some generic macros and types, in particular the `NULL` macro and the `size_t` type, that are so commonly used in user C code that are declared in some other standard C headers.

The `<stddef.h>` header declares 5 symbols, as summarized by the table below:

MACROS	
NULL	The null pointer constant (§2.13.1)
offsetof	Calculate the offset of a structure member (§2.13.2)
TYPES	
ptrdiff_t	Hold the difference of two pointers (§2.13.3)
size_t	Hold the result of the size_t operator (§2.13.4)
wchar_t	Hold the wide character values (§2.13.5)

Table 2.11: The `<stddef.h>` symbols.

2.13.1 NULL

Prototype

```
#include <stddef.h>

#define NULL    implementation-defined
```

Description

The `NULL` macro expands to an implementation-defined null pointer constant. It is usually defined as a `void` pointer or a zero value, like:

```
#define NULL   ((void *)0)
or
#define NULL   0
```

The `NULL` pointer constant should not be confused with the string terminating null character `'\0'`, witch has another mean. For example, for finding the end of a string, we use the terminating null character:

```
while (*s != '\0')
    ++s;
```

while for checking for a null pointer, we use the **NULL** constant:

```
char *p;
p = strchr("abcde", 'x');
if (p == NULL)
    printf("Character not found\n");
```

Example

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

/*
 * Get the next token taken from the `sep' separated argument
 * string `s'.
 * Return NULL when there are no more tokens available.
 */
char *next_token(char *buf, const char **s, int sep)
{
    char *dp = buf;

    while (**s == sep)
        ++*s;
    if (**s == '\0')
        return NULL;
    while (**s != sep && **s != '\0')
        *dp++ = *(*s)++;
    *dp = '\0';

    return buf;
}

/*
 * Print the tokens in witch is composed the colon `:' separated
 * argument string.
 */
void print_tokens(const char *s)
{
    char buf[32];
    while (next_token(buf, &s, ':') != NULL)
        printf("token = `%s'\n", buf);
}

int main(void)
```

```
{
    print_tokens("/bin:/sbin:/usr/bin:/usr/sbin:.");
    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.1.6

2.13.2 `offsetof`

Prototype

```
#include <stddef.h>

size_t offsetof(type, member-designator)
```

Description

The `offsetof` macro calculates the offset in bytes of the structure member *member-designator* from the beginning of the structure *type*. The member should not be a bit-field.

Example

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * This example show the alignment of the structure members done by
 * the compiler.
 */

struct s {
    char member1;
    int member2;
};

int main(void)
{
    printf("offsetof(struct s, member1) = %d\n",
           offsetof(struct s, member1));
    printf("offsetof(struct s, member2) = %d\n",
           offsetof(struct s, member2));

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.1.6

2.13.3 ptrdiff_t

Prototype

```
#include <stddef.h>

typedef implementation-defined ptrdiff_t;
```

Description

The `ptrdiff_t` type is the signed integral type of the result of subtracting two pointers.

Example

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * Apply the function ‘func’ to the sequence of ‘T’ objects,
 * starting from ‘begin’, terminating before ‘end’, excluded.
 */
#define APPLY(T, begin, end, func)              \
    do {                                        \
        T *p = begin;                           \
        ptrdiff_t d = end - begin;              \
        if (d == 0)                             \
            break;                              \
        else if (d > 0)                         \
            while (p < end)                     \
                func(*p++);                     \
        else /* d < 0 */                        \
            while (p > end)                     \
                func(*--p);                     \
    } while (0)

int main(void)
{
    char s[] = "Hello world";
    APPLY(char, &s[0], &s[11], putchar);
    putchar('\n');
    APPLY(char, &s[11], &s[0], putchar);
```

```
    putchar('\n');

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.1.6

2.13.4 size_t

Prototype

```
#include <stddef.h>

typedef implementation-defined size_t;
```

Description

The size_t type is the unsigned integral type of the result of the sizeof operator. It is widely used in the standard C library for declaring size parameters, for example:

```
void *memcpy(void *dest, const void *src, size_t n);
size_t strlen(const char *s);
char *strncat(char *dest, const char *src, size_t n);
```

Example

```
#include <stddef.h>

/*
 * Truncate the argument string to maximum 'n' characters.
 */
char *strtrunc(char *s, size_t n)
{
    char *sp;
    for (sp = s; *sp != '\0'; ++sp)
        if (sp - s >= n) {
            *sp = '\0';
            break;
        }
    return s;
}
```

References

ISO C 9899:1990 §7.1.6

2.13.5 `wchar_t`

Prototype

```
#include <stddef.h>

typedef implementation-defined wchar_t;
```

Description

The `wchar_t` type is an integral type capable of holding wide characters values; the terminating null character shall have value zero.

References

ISO C 9899:1990 §7.1.6

2.14 The `<stdio.h>` header

The `<stdio.h>` header declares 60 symbols, as summarized by the table below:

Macros	
`_IOFBF`	The full buffering constant (§??)
`_IOLBF`	The line buffering constant (§??)
`_IONBF`	The no buffering constant (§??)
`BUFSIZ`	The best block size for input/output efficiency (§??)
`EOF`	The end of file status constant (§??)
`FILENAME_MAX`	The maximum file name length (§??)
`FOPEN_MAN`	The maximum number of open files (§??)
`L_tmpnam`	The buffer size for a temporary file name (§??)
`NULL`	The null pointer constant (§2.13.1)
`SEEK_CUR`	The current position seeking relative constant (§??)
`SEEK_END`	The end of file seeking relative constant (§??)
`SEEK_SET`	The beginning of file seeking relative constant (§??)
`TMP_MAX`	The maximum number of generated temporary file names (§??)
Types	
`FILE`	The stream type (§??)
`fpos_t`	The file position type (§??)
`size_t`	The size type (§2.13.4)
Lvalues	
`stderr`	The standard error stream (§??)
`stdin`	The standard input stream (§??)
`stdout`	The standard output stream (§??)
Functions	
`fopen`	Open a stream (§2.14.9)
`freopen`	Close a stream and open another stream (§??)
`tmpfile`	Open a temporary stream (§2.14.5)
`fclose`	Close a stream (§2.14.7)
`setbuf`	Associate a buffer to a stream (§??)
`setvbuf`	Change the buffering method of a stream (§??)
`fflush`	Flush a stream (§2.14.8)
`feof`	Check the end of file status of a stream (§??)
`ferror`	Check the error status of a stream (§??)
`clearerr`	Clear the status indicators of a stream (§??)
	Continued on next page

Table 2.12: The `<stdio.h>` symbols (*continued on next page*).

Continued from previous page	
FUNCTIONS	
ftell	Obtain the current stream position (§??)
fseek	Set the file position (§??)
rewind	Set the file position to the beginning of the file (§??)
fgetpos	Store the current stream position (§??)
fsetpos	Set the stream position (§??)
getchar	Read a character from the standard input (§??)
fgetc	Read a character from a stream (§??)
getc	Read a character from a stream (§??)
ungetc	Pushes a character back into a stream (§??)
fgets	Read a string from a stream (§??)
gets	Read a string from a stream (§??)
fread	Read a block from a stream (§??)
putchar	Write a character into the standard output (§??)
fputc	Write a character into a stream (§??)
putc	Write a character into a stream (§??)
fputs	Write a string into a stream (§??)
puts	Write a string into a stream (§??)
fwrite	Write a block into a stream (§??)
fscanf	Read a formatted input from a stream (§??)
scanf	Read a formatted input from the standard input (§??)
sscanf	Read a formatted input from a string (§??)
fprintf	Write a formatted output into a stream (§??)
vfprintf	Write a formatted output into a stream (§??)
printf	Write a formatted output into the standard output (§??)
vprintf	Write a formatted output into the standard output (§??)
sprintf	Write a formatted output into a string (§??)
vsprintf	Write a formatted output into a string (§??)
perror	Write an error message on the standard error (§??)
remove	Remove a file (§2.14.3)
rename	Rename a file (§2.14.4)
tmpnam	Generate an unique temporary file name (§2.14.6)

Table 2.13: The `<stdio.h>` symbols (*continued from previous page*).

2.14.1 NULL

See §2.13.1 on page 50.

2.14.2 size_t

See §2.13.4 on page 54.

2.14.3 remove

Prototype

```
#include <stdio.h>

int remove(const char *filename);
```

Description

Removes the specified `filename` file and returns zero if the operation succeeds, nonzero if it fails.

Example

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

/*
 * rm.c: remove the specified files.
 * usage: rm <filename>...
 */
int main(int argc, char **argv)
{
    const char *program_name = argv[0];

    if (argc < 2) {
        fprintf(stderr, "usage: %s <filename>...\n", program_name);
        return EXIT_FAILURE;
    }

    while (--argc > 0)
        if (remove(*++argv) != 0) {
            fprintf(stderr, "%s:%s: %s\n", program_name, *argv,
                    strerror(errno));
            return EXIT_FAILURE;
        }

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.9.4.1

2.14.4 rename

Prototype

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
```

Description

Renames the specified `oldname` file to the `newname` name. The behavior is implementation-defined if a file named `newname` already exists before the call of `rename`. Returns zero if the operation succeeds, nonzero if it fails.

Example

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

/*
 * ren.c: rename the specified file.
 * usage: ren <oldname> <newname>
 */
int main(int argc, char **argv)
{
    if (argc != 3) {
        fprintf(stderr, "usage: %s <oldname> <newname>\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    if (rename(argv[1], argv[2]) != 0) {
        fprintf(stderr, "%s:%s: %s\n", argv[0], argv[1],
                strerror(errno));
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.9.4.2

2.14.5 tmpfile

Prototype

```
#include <stdio.h>

FILE *tmpfile(void);
```

Description

Creates a temporary binary file (with `fopen` mode `"wb+"`) that will be automatically removed when it is closed or at program termination. Returns a pointer to a stream of the created file. If the file cannot be created, returns `NULL`.

Example

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

/*
 * tmpcheck.c: tryes to open a user specified number of temporary
 *             files and to write some data into the files.
 * usage: tmpcheck <num>
 */
int main(int argc, char **argv)
{
    FILE **tmpvec;
    int i, num;
    char data[] = "Some data";

    if (argc != 2 || (num = atoi(argv[1])) < 1) {
        fprintf(stderr, "usage: %s <num>\n", argv[0]);
        return EXIT_FAILURE;
    }

    tmpvec = (FILE **)malloc(sizeof(FILE) * num);
    if (tmpvec == NULL) {
        fprintf(stderr, "%s: cannot allocate memory\n", argv[0]);
        return EXIT_FAILURE;
    }

    for (i = 0; i < num; ++i) {
        if ((tmpvec[i] = tmpfile()) == NULL) {
            fprintf(stderr, "%s: %s\n", argv[0], strerror(errno));
```

```
            return EXIT_FAILURE;
        }
        if (fwrite(data, 1, sizeof(data), tmpvec[i])
            != sizeof(data)) {
            fprintf(stderr, "%s: %s\n", argv[0], strerror(errno));
            return EXIT_FAILURE;
        }
    }

    for (i = 0; i < num; ++i)
        fclose(tmpvec[i]);
    free(tmpvec);

    printf("\nSuccess for %d temporary files\n", num);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.9.4.3

2.14.6 tmpnam

Prototype

```
#include <stdio.h>

char *tmpnam(char *s);
```

Description

Generates an unique file name useful to be used for creating temporary files. It generates a different name each time it is called, up to `TMP_MAX` times (if it is called more that `TMP_MAX` times, the behaviour is implementation-defined). If the argument `s` is not `NULL`, the `s` buffer (assumed to have a size of at least `L_tmpnam` characters) is used for storing the generated file name, then the `s` value is returned. If the argument `s` is `NULL`, an internal static buffer is used for storing the generated file name, and a pointer is returned to that buffer. Please note that in the latter case subsequent calls to `tmpnam` may modify the same internal buffer.

Example

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

int main(void)
```

```
{
    char *name;

    name = tmpnam(NULL);
    printf("Generated temporary name is \"%s\"\n", name);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.9.4.4

2.14.7 fclose

Prototype

```
#include <stdio.h>

int fclose(FILE *stream);
```

Description

Flushes the stream pointed to by `stream` and closes the associated stream. Returns zero if the operation succeeds, or `EOF` if any errors were detected.

Example

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

int main(void)
{
    FILE *fout;

    if ((fout = fopen("test.txt", "w")) == NULL) {
        fprintf(stderr, "%s\n", strerror(errno));
        return EXIT_FAILURE;
    }
    fprintf(fout, "Test string\n");
    fclose(fout);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.9.5.1

2.14.8 fflush

Prototype

```
#include <stdio.h>

int fflush(FILE *stream);
```

Description

Flushes the write buffers of the stream pointed to by `stream`, so that any unwritten data is actually written to the file. If the `stream` parameter is `NULL`, the `fflush` function flushes the write buffers of all the open streams.

Example

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

#define MAXLINELEN      1024

int main(void)
{
    char buf[MAXLINELEN];

    printf("What is your name? ");
    fflush(stdout);
    fgets(buf, MAXLINELEN, stdin);
    printf("You wrote %s", buf);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.9.5.2

2.14.9 fopen

Prototype

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
```

Description

Opens the specified file `filename` and returns a stream associated with it. If the open operation fails, returns `NULL`. The `mode` string specifies the operations that might be done on the stream, as summarized by the table below:

MODE	DESCRIPTION
`r`	Open text file for reading
`w`	Truncate to zero length or create text file for writing
`a`	Append; open or create text file for writing at end-of-file
`rb`	Open binary file for reading
`wb`	Truncate to zero length or create binary file for writing
`ab`	Append; open or create binary file for writing at end-of-file
`r+`	Open text file for update (reading and writing)
`w+`	Truncate to zero length or create text file for update, writing at end-of-file
`a+`	Append; open or create text file for update, writing at end-of-file
`r+b` *or* `rb+`	Open binary file for update (reading and writing)
`w+b` *or* `wb+`	Truncate to zero length or create binary file for update, writing at end-of-file
`a+b` *or* `ab+`	Append; open or create binary file for update, writing at end-of-file

Table 2.14: The `fopen` opening modes.

References

ISO C 9899:1990 §7.9.5.2

2.15 The <stdlib.h> header

The `<stdlib.h>` header declares 37 symbols, as summarized by the table below:

MACROS	
`EXIT_FAILURE`	(§??)
`EXIT_SUCCESS`	(§??)
`MB_CUR_MAX`	(§??)
`NULL`	The null pointer constant (§2.13.1)
`RAND_MAX`	(§??)
TYPES	
`div_t`	(§??)
`ldiv_t`	(§??)
`size_t`	(§2.13.4)
`wchar_t`	(§2.13.5)
FUNCTIONS	
`abort`	(§??)
`abs`	(§2.15.4)
`atexit`	(§??)
`atof`	(§??)
`atoi`	(§2.15.5)
`atol`	(§??)
`bsearch`	(§??)
`calloc`	(§??)
	Continued on next page

Table 2.15: The `<stdlib.h>` symbols (*continued on next page*).

Continued from previous page	
FUNCTIONS	
div	(§??)
exit	(§??)
free	(§??)
getenv	(§??)
labs	(§??)
ldiv	(§??)
malloc	(§??)
mblen	(§??)
mbstowcs	(§??)
mbtowc	(§??)
qsort	(§??)
rand	(§??)
realloc	(§??)
srand	(§??)
strtod	(§??)
strtol	(§??)
strtoul	(§??)
system	(§??)
wcstombs	(§??)
wctomb	(§??)

Table 2.16: The `<stdlib.h>` symbols (*continued from previous page*).

2.15.1 `NULL`

See §2.13.1 on page 50.

2.15.2 `size_t`

See §2.13.4 on page 54.

2.15.3 `wchar_t`

See §2.13.5 on page 55.

2.15.4 abs

Prototype

```
#include <stdlib.h>

int abs(int x);
```

Description

This function returns the absolute value of the argument integer `x`. If the argument value is positive, `abs` returns it unmodified, otherwise if the value is negative, `abs` returns it with the positive sign.

Implementation

```
#include <stdlib.h>

int abs(int x)
{
    return (x < 0) ? -x : x;
}
```

Example

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

int main(void)
{
    printf("abs(%d) -> %d\n", 0, abs(0));
    printf("abs(%d) -> %d\n", 1, abs(1));
    printf("abs(%d) -> %d\n", -1, abs(-1));

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.10.6.1

2.15.5 atoi

Prototype

```
#include <stdlib.h>

int atoi(const char *s);
```

Description

Converts the argument string `s` to an integer. Returns the converted value of the input string `s`.

Example

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

/*
 * twotimes.c: print the argument number multiplied by 2.
 * usage: twotimes <number>
 */
int main(int argc, char **argv)
{
    if (argc > 1)
        printf("Two times %s is %d\n", argv[1], atoi(argv[1]) * 2);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.10.1.2

2.16 The <string.h> header

The `<string.h>` header contains functions useful for doing the most common operations on memory areas and strings. The functions that begin with a `mem` prefix work on memory areas (including strings), and require to be specified a length of the memory area. The functions that begin with a `str` prefix work only on null-terminated C strings; those with a `strn` prefix accept a third argument, specifying the length of the string.

The `<string.h>` header declares 24 symbols, as summarized by the table below:

MACROS	
`NULL`	The null pointer constant (§2.13.1)
TYPES	
`size_t`	The object size type (§2.13.4)
FUNCTIONS	
`memchr`	Search for a character in a memory area (§2.16.3)
`memcmp`	Compare two memory areas (§2.16.4)
`memcpy`	Copy a memory area (§2.16.5)
`memmove`	Copy a memory area with overlap checking (§2.16.6)
`memset`	Initialize a memory area with a certain value (§2.16.7)
`strcat`	Append a string (§2.16.8)
`strchr`	Search for a character is a string (§2.16.9)
`strcmp`	Compare two strings (§2.16.10)
`strcoll`	Compare two strings using the current locale settings (§2.16.11)
`strcpy`	Copy a string (§2.16.12)
`strcspn`	Search for a set characters in a string (§2.16.13)
`strerror`	Convert an error value into a printable string (§2.16.14)
`strlen`	Calculate the length of a string (§2.16.15)
`strncat`	Append a string with specified length (§2.16.16)
`strncmp`	Compare two strings with specified length (§2.16.17)
`strncpy`	Copy a string with specified length (§2.16.18)
`strpbrk`	Search for any of a set of characters in a string (§2.16.19)
`strrchr`	Backward search for a character is a string (§2.16.20)
`strspn`	Search for a set characters in a string (§2.16.21)
`strstr`	Search for a substring in a string (§2.16.22)
`strtok`	Extract tokens from a string (§2.16.23)
`strxfrm`	Transform a string using the current locale settings (§2.16.24)

Table 2.17: The `<string.h>` symbols.

2.16.1 NULL

See §2.13.1 on page 50.

2.16.2 size_t

See §2.13.4 on page 54.

2.16.3 memchr

Prototype

```
#include <string.h>

void *memchr(const void *src, int c, size_t n);
```

Description

Finds in the first `n` bytes of the specified buffer `src` the specified character `c`. Returns a pointer to the first character found, otherwise `NULL`.

Implementation

```
#include <string.h>

void *memchr(const void *src, int c, size_t n)
{
    const unsigned char *sp;

    for (sp = src; n > 0; ++sp, --n)
        if (*sp == c)
            return (void *)sp;

    return NULL;
}
```

Example

```
#include <string.h>

/*
 * Search for any character found in ‘set’ in the first ‘n’
 * characters of the ‘s’ string and return a true value if found.
 */
int contains(const char *s, const char *set, size_t n)
{
    for (; *set != '\0'; ++set)
        if (memchr(s, *set, n) != NULL)
```

```
            return 1;
    return 0;
}
```

References

ISO C 9899:1990 §7.11.5.1

2.16.4 memcmp

Prototype

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
```

Description

Compares the first `n` bytes of the two memory areas `s1` and `s2`. It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`.

Implementation

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n)
{
    const unsigned char *sp1, *sp2;

    for (sp1 = s1, sp2 = s2; n > 0; ++sp1, ++sp2, --n)
        if (*sp1 != *sp2)
            return (*sp1 < *sp2) ? -1 : 1;

    return 0;
}
```

Example

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

/*
 * Check if two objects are equal (contain the same data).
 */
#define OBJEQ(obj1, obj2)                       \
        (sizeof(obj1) == sizeof(obj2) &&        \
         memcmp(&(obj1), &(obj2), sizeof(obj1)) == 0)
```

```
int main(void)
{
    int a = 1, b = 1, c = 2;
    int *bp = &b;
    char d = 1;

    printf("%d\n", OBJEQ(a, b));   /* Prints "1". */
    printf("%d\n", OBJEQ(a, c));   /* Prints "0". */
    printf("%d\n", OBJEQ(a, *bp)); /* Prints "1". */
    printf("%d\n", OBJEQ(a, d));   /* Prints "0". */

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.4.1

2.16.5 memcpy

Prototype

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

Description

Copies the data pointed by `src` into the `dest` buffer, overwriting the previous existing data. The size in bytes to be copied is specified by the `n` parameter. The two areas should not overlap. Returns the `dest` parameter.

Implementation

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n)
{
    char *dp = dest;
    const char *sp = src;

    while (n-- > 0)
        *dp++ = *sp++;

    return dest;
}
```

Example

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

/*
 * Duplicate the argument object.
 */
void *objdup(void *ptr, size_t size)
{
    void *nptr;

    if ((nptr = malloc(size)) != NULL)
        memcpy(nptr, ptr, size);

    return nptr;
}

int main(void)
{
    struct s { int a; char c; } obj, *obj2;

    obj.a = 1; obj.c = 'x';
    if ((obj2 = objdup(&obj, sizeof obj)) == NULL) {
        fprintf(stderr, "Cannot allocate memory\n");
        return EXIT_FAILURE;
    }
    if (obj.a != obj2->a || obj.c != obj2->c) {
        free(obj2);
        fprintf(stderr, "Unexpected behavior\n");
        return EXIT_FAILURE;
    }
    free(obj2);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.2.1

2.16.6 memmove

Prototype

```
#include <string.h>

void *memmove(void *dest, const void *src, size_t n);
```

Description

Copies the data pointed by `src` into the `dest` buffer, overwriting the previous existing data. The size in bytes to be copied is specified by the `n` parameter. The two areas can overlap. Returns the `dest` parameter.

Implementation

```
#include <string.h>

void *memmove(void *dest, const void *src, size_t n)
{
    char *dp = dest;
    const char *sp = src;

    if (dest < src)
        while (n-- > 0)
            *dp++ = *sp++;
    else {
        dp += n;
        sp += n;
        while (n-- > 0)
            *--dp = *--sp;
    }

    return dest;
}
```

Example

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

int main(void)
{
    char buf[8];

    strcpy(buf, "abcdef");
    strcpy(buf + 1, buf);       /* Undefined behavior. */
```

```
    strcpy(buf, "abcdef");
    memmove(buf + 1, buf, 7);   /* OK. */
    printf("%s\n", buf);        /* Prints "aabcdef". */

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.2.2

2.16.7 memset

Prototype

```
#include <string.h>

void *memset(void *dest, int c, size_t n);
```

Description

Fills the first `n` bytes of the specified buffer `dest` with the specified character `c`. Returns the `dest` parameter.

For example, in the following code we initialize a part of an array:

```
char buf[9];
memset(buf + 2, 'x', 5);
```

this is the state of the `buf` array before and after the call of `memset` ? means any character):

buf:	?	?	?	?	?	?	?	?	?	before
buf:	?	?	x	x	x	x	x	?	?	after

Implementation

```
#include <string.h>

void *memset(void *dest, int c, size_t n)
{
    unsigned char *dp = dest;

    while (n-- > 0)
```

```
        *dp++ = c;

    return dest;
}
```

Example

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

int main(void)
{
    char buf[8];

    memset(buf, 'x', 7);
    memset(buf + 1, 'y', 5);
    buf[7] = '\0';                /* Add the terminating null. */
    printf("%s\n", buf);          /* Prints "xyyyyyx". */

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.6.1

2.16.8 `strcat`

Prototype

```
#include <string.h>

char *strcat(char *dest, const char *src);
```

Description

Appends the `src` string to the `dest` string overwriting the terminating null character at the end of `dest`, and then adds a terminating null character. The strings should not overlap, and the `dest` buffer size should be greater or equal to `strlen(dest) + strlen(str) + 1`. Returns the `dest` parameter.

For example, in the following code we catenate two strings:

```
char buf[8];
strcpy(buf, "desk");
strcat(buf, "top");
```

this is the state of the `buf` array before and after the call of `strcat` ? means any character):

buf:	d	e	s	k	\0	?	?	?	before

buf:	d	e	s	k	t	o	p	\0	after

Implementation

```
#include <string.h>

char *strcat(char *dest, const char *src)
{
    char *dp = dest;

    while (*dp != '\0')
        ++dp;
    while ((*dp++ = *src++) != '\0')
        ;

    return dest;
}
```

Example

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

int main(void)
{
    char buf[7];

    strcpy(buf, "abc");
    strcat(buf, "def");
    printf("%s\n", buf);         /* Prints "abcdef". */
    printf("%d\n", strlen(buf)); /* Prints "6". */

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.3.1

2.16.9 strchr

Prototype

```
#include <string.h>

char *strchr(const char *str, int c);
```

Description

Finds in the specified string `str`, starting from the beginning, the specified character `c`. Returns a pointer to the first character found, otherwise `NULL`.

Implementation

```
#include <string.h>

char *strchr(const char *str, int c)
{
    for (; *str != '\0'; ++str)
        if (*str == c)
            return (char *)str;

    return NULL;
}
```

Example

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

int main(int argc, char **argv)
{
    char *valid_opts = "abc";
    int c;

    while (--argc > 0 && (*++argv)[0] == '-')
        while ((c = *++argv[0]) != '\0')
            if (strchr(valid_opts, c))
                printf("got valid option `-%c'\n", c);
            else {
                printf("got invalid option `-%c'\n", c);
                return EXIT_FAILURE;
            }

    printf("other arguments: ");
    while (argc--)
```

```
        printf("%s ", *argv++);
    printf("\n");

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.5.2

2.16.10 strcmp

Prototype

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

Description

Compares the two strings `s1` and `s2` character by character up to the terminating null character. It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`.

Implementation

```
#include <string.h>

int strcmp(const char *s1, const char *s2)
{
    const unsigned char *sp1 = (const unsigned char *)s1;
    const unsigned char *sp2 = (const unsigned char *)s2;

    for (; *sp1 == *s2; ++sp1, ++sp2)
        if (*sp1 == '\0')
            return 0;

    return *sp1 - *sp2;
}
```

Example

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

/*
 * cmpstr.c: compare the two argument strings.
 * usage: cmpstr <string1> <string2>
```

```
 */
int main(int argc, char **argv)
{
    int i;

    if (argc != 3) {
        fprintf(stderr, "usage: %s <string1> <string2>\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    i = strcmp(argv[1], argv[2]);
    printf("‘%s’ is %s ‘%s’\n", argv[1],
           ((i > 0) ? "greater than" :
            (i < 0) ? "less than" : "equal to"), argv[2]);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.4.2

2.16.11 strcoll

Prototype

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

Description

Compares the two strings `s1` and `s2` character by character up to the terminating null character. It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. The comparison is based on the current locale settings (set by the `setlocale` function).

Example

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

/*
 * collstr.c: compare the two argument strings.
```

```
 * usage: collstr <string1> <string2>
 */
int main(int argc, char **argv)
{
    int i;

    if (argc != 3) {
        fprintf(stderr, "usage: %s <string1> <string2>\n",
                argv[0]);
        return EXIT_FAILURE;
    }

    setlocale(LC_ALL, "");

    i = strcoll(argv[1], argv[2]);
    printf("`%s' is %s `%s'\n", argv[1],
           ((i > 0) ? "greater than" :
            (i < 0) ? "less than" : "equal to"), argv[2]);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.4.3

2.16.12 strcpy

Prototype

```
char *strcpy(char *dest, const char *src);
```

Description

Copies the specified string `src` including the terminating null character in the buffer `dest` overwriting the previous existing data. The `dest` buffer should be large enough to contain the new string. The two strings should not overlap. Returns the `dest` parameter.

For example, in the following code we copy a string into an array:

```
char buf[12];
strcpy(buf, "hello world");
```

this is the state of the `buf` array before and after the call of `strcpy` `?` means any character):

buf:	?	?	?	?	?	?	?	?	?	?	?	?	before

buf:	h	e	l	l	o		w	o	r	l	d	\0	after

Implementation

```
#include <string.h>

char *strcpy(char *dest, const char *src)
{
    char *dp = dest;

    while ((*dp++ = *src++) != '\0')
        ;

    return dest;
}
```

Example

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

int main(void)
{
    char buf[12];

    strcpy(buf, "Hello world");
    printf("%s\n", buf);        /* Prints "Hello world". */

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.2.3

2.16.13 `strcspn`

Prototype

```
#include <string.h>

size_t strcspn(const char *s, const char *set);
```

Description

Returns the length of the maximum initial segment of the string `s` which consists entirely of characters *not* from the string `set`.

Implementation

```
#include <string.h>

size_t strcspn(const char *s, const char *set)
{
    const char *sp1, *sp2;

    for (sp1 = s; *sp1 != '\0'; ++sp1)
        for (sp2 = set; *sp2 != '\0'; ++sp2)
            if (*sp1 == *sp2)
                return sp1 - s;

    return sp1 - s;
}
```

Example

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

/*
 * Extract the next white space separated tokenfrom the `s' string.
 */
char *tokenize(char *buf, const char **s)
{
    size_t size;

    while (isspace(**s))
        ++*s;
    size = strcspn(*s, "\t\v\f\r\n ");
    if (size < 1)
        return NULL;
    strncpy(buf, *s, size);
    buf[size] = '\0';
    *s += size;

    return buf;
}
```

```
int main(void)
{
    const char *s = "test string\na b\t c ";
    char buf[128];

    while (tokenize(buf, &s) != NULL)
        printf("token = \"%s\"\n", buf);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.5.3

2.16.14 strerror

Prototype

```
#include <string.h>

char *strerror(int errnum);
```

Description

Returns a string describing the error code `errnum`. The returned string cannot be modified.

Example

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

/*
 * fcheck.c: check if the argument file exists.
 * usage: fcheck <filename>
 */
int main(int argc, char **argv)
{
    FILE *f;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <filename>\n", argv[0]);
        return EXIT_FAILURE;
```

```
    }

    if ((f = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "%s:%s: %s\n", argv[0], argv[1],
                strerror(errno));
        return EXIT_FAILURE;
    }
    printf("The ‘%s’ file is readable.\n", argv[1]);
    fclose(f);

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.6.2

2.16.15 strlen

Prototype

```
#include <string.h>

size_t strlen(const char *s);
```

Description

Returns the number of characters of the specified string `s`, up to the terminating null character, excluded.

Implementation

```
#include <string.h>

size_t strlen(const char *s)
{
    const char *p;

    for (p = s; *p != '\0'; ++p)
        ;

    return p - s;
}
```

Example

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

```
/*
 * len.c: print the length in characters of the argument string.
 * usage: len <string>
 */
int main(int argc, char **argv)
{
    if (argc > 1)
        printf("The length of the argument ‘%s’ is %d\n",
               argv[1], strlen(argv[1]));

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.6.3

2.16.16 strncat

Prototype

```
#include <string.h>

char *strncat(char *dest, const char *src, size_t n);
```

Description

Appends the first `n` characters of the `src` string to the `dest` string overwriting the terminating null character at the end of `dest`, and then adds a terminating null character. The strings should not overlap, and the `dest` buffer size should be greater or equal to `strlen(dest) + n + 1`. Returns the `dest` parameter.

For example, in the following code we catenate two strings, specifing a `n` parameter less than the `src` string length:

```
char buf[13];
strcpy(buf, "ray");
strncat(buf, "-tracing", 4);
```

this is the state of the `buf` array before and after the call of `strncat` `?` means any character):

buf:	r	a	y	\0	?	?	?	?	?	?	?	?	?	before

buf:	r	a	y	-	t	r	a	\0	?	?	?	?	?	after

In this second example we catenate two strings, specifing a `n` parameter greater that the `src` string length:

```
char buf[13];
strcpy(buf, "ray");
strncat(buf, "-tracing", 10);
```

this is the state of the `buf` array before and after the call of `strncat`:

buf:	r	a	y	\0	?	?	?	?	?	?	?	?	?	before
buf:	r	a	y	-	t	r	a	c	i	n	g	\0	?	after

Implementation

```
#include <string.h>

char *strncat(char *dest, const char *src, size_t n)
{
    char *dp;

    for (dp = dest; *dp != '\0'; ++dp)
        ;
    while (n-- > 0 && (*dp++ = *src++) != '\0')
        ;
    *dp = '\0';

    return dest;
}
```

Example

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

int main(void)
{
    char buf[7];

    strcpy(buf, "abc");
    strncat(buf, "defghi", 3);
```

```
    printf("%s\n", buf);         /* Prints "abcdef". */
    printf("%d\n", strlen(buf)); /* Prints "6". */

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.3.2

2.16.17 strncmp

Prototype

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);
```

Description

Compares the first `n` characters of the two strings `s1` and `s2` (the comparison is interrupted before the terminating null characters). It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`.

Implementation

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n)
{
    const unsigned char *sp1 = (const unsigned char *)s1;
    const unsigned char *sp2 = (const unsigned char *)s2;

    for (; n > 0; ++sp1, ++sp2, --n)
        if (*sp1 != *sp2)
            return (*sp1 < *sp2) ? -1 : 1;
        else if (*sp1 == '\0')
            return 0;

    return 0;
}
```

Example

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

```
int main(void)
{
    char *s1 = "abc", *s2 = "abcd";
    int retvalue;

    retvalue = strcmp(s1, s2);
    printf("'%s' %s '%s'\n", s1,
           ((retvalue < 0) ? "<" :
            (retvalue > 0) ? ">" : "=="), s2);
    /* Prints "'abc' < 'abcd'". */

    retvalue = strncmp(s1, s2, 3);
    printf("'%s' %s '%s'\n", s1,
           ((retvalue < 0) ? "<" :
            (retvalue > 0) ? ">" : "=="), s2);
    /* Prints "'abc' == 'abcd'". */

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.4.4

2.16.18 strncpy

Prototype

```
#include <string.h>

char *strncpy(char *dest, const char *src, size_t n);
```

Description

Copies the first `n` characters of the specified string `src` in the buffer `dest` overwriting the previous existing data. If there is no terminating null character among the first `n` characters of the string `src`, the `dest` string will not be null-terminated. If the length of the `src` string is less than `n`, the remaining characters of `dest` will be padded with null characters. The `dest` buffer should be large enough to contain the new string. The two strings should not overlap. Returns the `dest` parameter.

For example, in the following code the `n` parameter is less than the size of the `src` string:

```
char buf[12];
strncpy(buf, "a string", 5);
```

this is the state of the buf array before and after the call of strncpy ? means any character):

buf:	?	?	?	?	?	?	?	?	?	?	?	?	before
buf:	a		s	t	r	?	?	?	?	?	?	?	after

In this second example the n parameter is greaten than the size of the src string:

```
char buf[12];
strncpy(buf, "a string", 11);
```

this is the state of the buf array before and after the call of strncpy:

buf:	?	?	?	?	?	?	?	?	?	?	?	?	before
buf:	a		s	t	r	i	n	g	\0	\0	\0	?	after

Implementation

```
#include <string.h>

char *strncpy(char *dest, const char *src, size_t n)
{
    char *dp = dest;

    while (n-- > 0 && (*dp++ = *src++) != '\0')
        ;
    while (n-- > 0)
        *dp++ = '\0';

    return dest;
}
```

Example

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

```
int main(void)
{
    char buf[10];

    strcpy(buf, "abcdefghi");    /* buf = "abcdefghi\0" */
    strncpy(buf, "-+-+-", 5);    /* buf = "-+-+-fghi\0" */
    printf("%s\n", buf);         /* Prints "-+-+-fghi". */
    printf("%d\n", strlen(buf)); /* Prints "9". */

    strcpy(buf, "abcdefghi");    /* buf = "abcdefghi\0" */
    strncpy(buf, "-+-+-", 6);    /* buf = "-+-+-\0ghi\0" */
    printf("%s\n", buf);         /* Prints "-+-+-". */
    printf("%d\n", strlen(buf)); /* Prints "5". */

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.2.4

2.16.19 strpbrk

Prototype

```
#include <string.h>

char *strpbrk(const char *s, const char *set);
```

Description

Returns a pointer to the first occurrence in the string `s` of any character from the string `set`, or `NULL` if no character from `set` occurs in `s`.

Implementation

```
#include <string.h>

char *strpbrk(const char *s, const char *set)
{
    const char *sp1, *sp2;

    for (sp1 = s; *sp1 != '\0'; ++sp1)
        for (sp2 = set; *sp2 != '\0'; ++sp2)
            if (*sp1 == *sp2)
                return (char *)sp1;
```

```
    return NULL;
}
```

Example

```
#include <string.h>

/*
 * Count the punctuation characters ",.;:?!" found in
 * the 's' string.
 */
int count_punct(const char *s)
{
    int count = 0;
    while ((s = strpbrk(s, ",.;:?!")) != NULL)
        ++count, ++s;
    return count;
}
```

References

ISO C 9899:1990 §7.11.5.4

2.16.20 strrchr

Prototype

```
#include <string.h>

char *strrchr(const char *str, int c);
```

Description

Finds in the specified string `str`, starting from the end, the specified character `c`. Returns a pointer to the first character found, otherwise `NULL`.

Implementation

```
#include <string.h>

char *strrchr(const char *str, int c)
{
    const char *p = NULL;

    for (; *str != '\0'; ++str)
        if (*str == c)
            p = str;

    return (char *)p;
```

```
}
```

Example

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

int main(void)
{
    char *s = "Hi Hello", *p;

    p = strchr(s, 'H');
    printf("%s\n", p);          /* Prints "Hi Hello". */
    p = strrchr(s, 'H');
    printf("%s\n", p);          /* Prints "Hello". */

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.5.5

2.16.21 strspn

Prototype

```
#include <string.h>

size_t strspn(const char *s, const char *set);
```

Description

Returns the length of the maximum initial segment of the string s which consists entirely of characters from the string set.

Implementation

```
#include <string.h>

size_t strspn(const char *s, const char *set)
{
    const char *sp1, *sp2;

    for (sp1 = s; *sp1 != '\0'; ++sp1)
        for (sp2 = set; ; ++sp2)
            if (*sp2 == '\0')
                return sp1 - s;
```

```
            else if (*sp1 == *sp2)
                break;

    return sp1 - s;
}
```

Example

```
#include <string.h>

/*
 * Return a true value if the argument string is a regular
 * decimal number.
 */
int is_number(const char *s)
{
    if (strspn(s, "0123456789") == strlen(s))
        return 1;
    return 0;
}
```

References

ISO C 9899:1990 §7.11.5.6

2.16.22 strstr

Prototype

```
#include <string.h>

char *strstr(const char *s, const char *substr);
```

Description

Returns a pointer to the first occurrence of the substring `substr` in the string `s`. The terminating null characters are not compared. If the substring is not found, `strstr` returns `NULL`. If `substr` points to a string with zero length, `strstr` returns the `s` string.

Implementation

```
#include <string.h>

char *strstr(const char *s, const char *substr)
{
    if (*substr == '\0')
        return (char *)s;
    for (; (s = strchr(s, *substr)) != NULL; ++s) {
```

```
        const char *sp1 = s, *sp2 = substr;
        for (;;)
            if (*++sp2 == '\0')
                return (char *)s;
            else if (*++sp1 != *sp2)
                break;
    }

    return NULL;
}
```

Example

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

#define MAXLINELEN      1024

/*
 * grep.c: find the matches of the user specified string in
 *        the standard input.
 * usage: grep <string>
 */
int main(int argc, char **argv)
{
    char buf[MAXLINELEN];
    int lineno = 1;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <string>\n", argv[0]);
        return EXIT_FAILURE;
    }

    while (fgets(buf, MAXLINELEN, stdin) != NULL) {
        if (strstr(buf, argv[1]) != NULL)
            printf("%s:%d: %s", argv[0], lineno, buf);
        ++lineno;
    }

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.5.7

2.16.23 strtok

Prototype

```
#include <string.h>

char *strtok(char *s, const char *delim);
```

Description

Decomposes the `s` string into tokens, witch are sequences of characters not occurring in the string `delim`. The first call to `strtok` should have the `s` parameter set to a valid string pointer. Subsequent calls should have the `s` parameter set to `NULL`; the `delim` parameter may be different for each call. The `strtok` returns a pointer to the next token, or `NULL` if there are no more tokens. Please note that every call modifies the argument string `s` (replaces the delimiter after the token with a terminating null character).

Implementation

```
#include <string.h>

char *strtok(char *s, const char *delim)
{
    const char *sp;
    char *s_begin, *s_end;
    static char *s_save = NULL;

    s_begin = (s != NULL) ? s : s_save;
    if (s_begin == NULL)
        return NULL;

    /* Skip the delimiters. */
    for (;;) {
        if (*s_begin == '\0') {
            s_save = NULL;
            return NULL;
        }
        for (sp = delim; *sp != '\0'; ++sp)
            if (*s_begin == *sp) {
                ++s_begin;
                break;
            }
```

```
        if (*sp == '\0')
            break;
    }

    /* Find the token end. */
    for (s_end = s_begin; *s_end != '\0'; ++s_end) {
        for (sp = delim; *sp != '\0'; ++sp)
            if (*s_end == *sp)
                break;
        if (*sp != '\0')
            break;
    }
    if (*s_end != '\0')
        *s_end++ = '\0';
    s_save = s_end;

    return s_begin;
}
```

Example

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

int main(void)
{
    char buf[] = "dog:cat;mouse/bee:;horse", *p;

    p = strtok(buf, ":;/");
    while (p != NULL) {
        printf("Token \"%s\"\n", p);
        p = strtok(NULL, ":;/");
    }

    return EXIT_SUCCESS;
}
```

References

ISO C 9899:1990 §7.11.5.8

2.16.24 strxfrm

Prototype

```
#include <string.h>

size_t strxfrm(char *dest, const char *src, size_t n);
```

Description

Transforms the `src` string into a form such that the result of `strcmp` on two strings that has been transformed with `strxfrm` is the same as the result of `strcoll` on the two strings before their transformation. The resulting string (of length equal or less that `n` characters, including the terminating null character) is placed into the `dest` array. If `n` is zero, `dest` is permitted to be `NULL`. This might be useful to determine the required `dest` array size:

```
size = strxfrm(NULL, src, 0) + 1;
```

If copying takes place between objects that overlap, the behavior is undefined. The `strxfrm` function returns the exact number of characters placed into the `dest` array.

References

ISO C 9899:1990 §7.11.4.5

2.17 The <time.h> header

The `<time.h>` header declares 15 symbols, as summarized by the table below:

MACROS	
CLOCKS_PER_SEC	The number of clock ticks per second (§??)
NULL	The null pointer constant (§2.13.1)
TYPES	
clock_t	The object size type (§??)
size_t	The object size type (§2.13.4)
time_t	The object size type (§??)
STRUCTURES	
tm	The object size type (§??)
FUNCTIONS	
asctime	(§??)
clock	(§??)
ctime	(§??)
difftime	(§??)
gmtime	(§??)
localtime	(§??)
mktime	(§??)
strftime	(§??)
time	(§??)

Table 2.18: The `<time.h>` symbols.

2.17.1 NULL

See §2.13.1 on page 50.

2.17.2 size_t

See §2.13.4 on page 54.

2.18 The <wchar.h> header

The `<wchar.h>` header declares 60 symbols, as summarized by the table below:

MACROS	
NULL	The null pointer constant (§2.13.1)
WCHAR_MAX	(§??)
WCHAR_MIN	(§??)
WEOF	(§2.18.2)
TYPES	
mbstate_t	(§2.18.3)
size_t	(§2.13.4)
wchar_t	(§2.13.5)
wint_t	(§2.18.6)
FUNCTIONS	
btowc	(§??)
fgetwc	(§??)
fgetws	(§??)
fputwc	(§??)
fputws	(§??)
fwide	(§??)
fwprintf	(§??)
fwscanf	(§??)
getwchar	(§??)
getwc	(§??)
mbrlen	(§??)
mbrtowc	(§??)
mbsinit	(§??)
mbsrtowcs	(§??)
putwchar	(§??)
putwc	(§??)
swprintf	(§??)
swscanf	(§??)
ungetwc	(§??)
vfwprintf	(§??)
	Continued on next page

Table 2.19: The `<wchar.h>` symbols (*continued on next page*).

Continued from previous page	
FUNCTIONS	
`vswprintf`	(§??)
`vwprintf`	(§??)
`wcrtomb`	(§??)
`wcscat`	(§??)
`wcschr`	(§??)
`wcscmp`	(§??)
`wcscoll`	(§??)
`wcscpy`	(§??)
`wcscspn`	(§??)
`wcsftime`	(§??)
`wcslen`	(§??)
`wcsncat`	(§??)
`wcsncmp`	(§??)
`wcsncpy`	(§??)
`wcspbrk`	(§??)
`wcsrchr`	(§??)
`wcsrtombs`	(§??)
`wcsspn`	(§??)
`wcsstr`	(§??)
`wcstod`	(§??)
`wcstok`	(§??)
`wcstol`	(§??)
`wcstoul`	(§??)
`wcsxfrm`	(§??)
`wctob`	(§??)
`wmemchr`	(§??)
`wmemcmp`	(§??)
`wmemcpy`	(§??)
`wmemmove`	(§??)
`wmemset`	(§??)
`wprintf`	(§??)
`wscanf`	(§??)

Table 2.20: The `<wchar.h>` symbols (*continued from previous page*).

2.18.1 NULL

See §2.13.1 on page 50.

2.18.2 WEOF

Prototype

```
#include <wchar.h>

#define WEOF    implementation-defined
```

References

ISO C 9899:1990 §XXX

2.18.3 mbstate_t

Prototype

```
#include <wchar.h>

typedef implementation-defined mbstate_t;
```

References

ISO C 9899:1990 §XXX

2.18.4 size_t

See §2.13.4 on page 54.

2.18.5 wchar_t

See §2.13.5 on page 55.

2.18.6 wint_t

Prototype

```
#include <wchar.h>

typedef implementation-defined wint_t;
```

References

ISO C 9899:1990 §XXX

2.19 The `<wctype.h>` header

The `<wctype.h>` header declares 21 symbols, as summarized by the table below:

MACROS	
`WEOF`	(§2.18.2)
TYPES	
`wctrans_t`	(§2.19.2)
`wctype_t`	(§2.19.3)
`wint_t`	(§2.18.6)
FUNCTIONS	
`iswalnum`	Check if a character is an alphabetic or a digit character (§2.19.5)
`iswalpha`	Check if a character is an alphabetic character (§2.19.6)
`iswcntrl`	Check if a character is a control character (§2.19.7)
`iswctype`	(§??)
`iswdigit`	Check if a character is a digit (§2.19.8)
`iswgraph`	Check if a character is a graphic character (§2.19.9)
`iswlower`	Check if a character is a lowercase character (§2.19.10)
`iswprint`	Check if a character is a printable character (§2.19.11)
`iswpunct`	Check if a character is a punctuation character (§2.19.12)
`iswspace`	Check if a character is a space character (§2.19.13)
`iswupper`	Check if a character is an uppercase character (§2.19.14)
`iswxdigit`	Check if a character is an hexadecimal digit (§2.19.15)
`towctrans`	(§??)
`towlower`	Convert a character to a lowercase character (§2.19.16)
`towupper`	Convert a character to an uppercase character (§2.19.17)
`wctrans`	(§??)
`wctype`	(§??)

Table 2.21: The `<wctype.h>` symbols.

2.19.1 `WEOF`

See §2.18.2 on the preceding page.

2.19.2 `wctrans_t`

Prototype

```
#include <wctype.h>
```

```
typedef implementation-defined wctrans_t;
```

References
ISO C 9899:1990 §XXX

2.19.3 wctype_t

Prototype

```
#include <wctype.h>

typedef implementation-defined wctype_t;
```

References
ISO C 9899:1990 §XXX

2.19.4 wint_t

See §2.18.6 on page 102.

2.19.5 iswalnum

Prototype

```
#include <wctype.h>

int iswalnum(int c);
```

Description
Returns a true value for any character for which `isalpha` or `isdigit` is true.

References
ISO C 9899:1990 §XXX

2.19.6 iswalpha

Prototype

```
#include <wctype.h>

int iswalpha(int c);
```

Description
In the "C" locale, returns a true value for the characters for which `isupper` or `islower` is true. In other implementation-defined set of characters returns true for the characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true.

References

ISO C 9899:1990 §XXX

2.19.7 iswcntrl

Prototype

```
#include <wctype.h>

int iswcntrl(int c);
```

Description

Returns a true value if the `c` character is a control character.

References

ISO C 9899:1990 §XXX

2.19.8 iswdigit

Prototype

```
#include <wctype.h>

int iswdigit(int c);
```

Description

Returns a true value if the `c` character is a decimal-digit character.

References

ISO C 9899:1990 §XXX

2.19.9 iswgraph

Prototype

```
#include <wctype.h>

int iswgraph(int c);
```

Description

Returns a true value if the `c` character is a printing character except the space character (`' '`).

References

ISO C 9899:1990 §XXX

2.19.10 iswlower

Prototype

```
#include <wctype.h>

int iswlower(int c);
```

Description

In the "C" locale, returns a true value for the characters defined as lowercase letters. In other implementation-defined set of characters returns true for the characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true.

References

ISO C 9899:1990 §XXX

2.19.11 iswprint

Prototype

```
#include <wctype.h>

int iswprint(int c);
```

Description

Returns a true value if the `c` character is a printing character including the space character (`' '`).

References

ISO C 9899:1990 §XXX

2.19.12 iswpunct

Prototype

```
#include <wctype.h>

int iswpunct(int c);
```

Description

Returns a true value if the `c` character is a printing character neither a space character (`' '`) nor a character for which `isalnum` is true.

References

ISO C 9899:1990 §XXX

2.19.13 iswspace

Prototype

```
#include <wctype.h>

int iswspace(int c);
```

Description

In the "C" locale, returns a true value if the `c` character is a standard white-space character (space `' '`, form feed `'\f'`, new line `'\n'`, carriage return `'\r'`, horizontal tab `'\t'`, and vertical tab `'\v'`). In other implementation-defined set of characters returns true for the characters for which `isalnum` is false.

References

ISO C 9899:1990 §XXX

2.19.14 iswupper

Prototype

```
#include <wctype.h>

int iswupper(int c);
```

Description

In the "C" locale, returns a true value for the characters defined as uppercase letters. In other implementation-defined set of characters returns true for the characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true.

References

ISO C 9899:1990 §XXX

2.19.15 iswxdigit

Prototype

```
#include <wctype.h>

int iswxdigit(int c);
```

Description

Returns a true value if the `c` character is an hexadecimal-digit character.

References

ISO C 9899:1990 §XXX

2.19.16 `towlower`

Prototype

```
#include <wctype.h>

int tolower(int c);
```

Description

Returns the lowercase character corresponding to the character `c`, if the value of `isupper` with the argument `c` is true, otherwise returns the `c` argument unchanged.

References

ISO C 9899:1990 §XXX

2.19.17 `towupper`

Prototype

```
#include <wctype.h>

int toupper(int c);
```

Description

Returns the uppercase character corresponding to the character `c`, if the value of `islower` with the argument `c` is true, otherwise returns the `c` argument unchanged.

References

ISO C 9899:1990 §XXX

Chapter 3

The Future Directions

3.1 The C9x programming language

3.1.1 The main C9x features

The C9x language provides 36 keywords, as written in the table below:

auto	double	inline	static
break	else	int	struct
case	enum	long	switch
char	extern	register	typedef
complex	float	restrict	union
const	for	return	unsigned
continue	goto	short	void
default	if	signed	volatile
do	imaginary	sizeof	while

Table 3.1: The C9x keywords.

Restricted pointers

An object that is referenced through a restrict-qualified pointer has a special association with that pointer. This association, requires that all references to that object shall use, directly or indirectly, the value of that pointer. The intended use of the `restrict` qualifier (like the `register` storage class introduced in the ISO C standard) is to promote optimization. For example, suppose to write the following code:

```
void set_array(int v[], const int *value, int n)
{
    int i;
    for (i = 0; i < n; ++i)
        v[i] = *value * 2;
}
```

In theory this can be optimized by the compiler (using a hosting technique, see §4.3.1) to the following:

```
void set_array(int v[], const int *value, int n)
{
    int i;
    int temp = *value * 2;
    for (i = 0; i < n; ++i)
```

```
        v[i] = temp;
}
```

However, a developer might call this function in this mode, that is *legal* in ISO C:

```
void f()
{
    int v[50];
    v[25] = 4;
    set_array(v, &v[25], 50);
}
```

During the iteration of `i` from the values `0` to `25` the vector elements `v[0..25]` will assume the value `8`. Since the `v[25]` entry is also an alias of the `value` pointed integer, the later will also assume the value `8`. During the iterator of `i` from the values `26` to `49` the vector elements `v[26..49]` will assume the value `16`.

This is not so if the compiler tries to optimize the `set_array` code, moving the multiplication out of the loop code. For allowing more aggressive optimizations of this type of code, the C9x committee introduced the `restrict` keyword; the previous code can be easily rewritten as:

```
void set_array(int * restrict v, const int *value, int n)
{
    int i;
    for (i = 0; i < n; ++i)
        v[i] = *value * 2;
}
```

if we are certain that the function won't be called as in the previous function `f`.

Variable-length arrays

In the C9x language it is now possible to write array definitions like this:

```
void f(int n)
{
    int v[n * 2];
    f2(v);
}
```

In ISO C this is not possible; we have instead to write something like this:

```
void f(int n)
{
```

```
    int *v;
    v = (int *)malloc(sizeof(int) * n * 2);
    if (v != NULL)
        /* ... */
    f2(v);
    free(v);
}
```

Designed initializers

It is possible to specify the elements of an array in a bracketed initialization:

```
enum {
    ERROR_MATH,
    ERROR_IO
};
const char *error_msgs[] = {
    [ERROR_MATH] = "Mathematical error",
    [ERROR_IO] = "Input/Output error",
};
```

Structure members can be specified in a bracketed initialization:

```
struct node {
    int id;
    struct node *left;
    struct node *right;
};
struct node t = { .id = 1, .left = NULL, .right = NULL };
```

New types

The C9x languase introduces the new `long long int` integer, as twice as wide as `long int`. Library functions are added for supporting this new type, like the `llabs` and `atoll` functions.

Other features

Some features were borrowed from the C++ language. This includes the facility of mixing declarations and code and the the facility of putting declarations into the `for` statement like in the following code:

```
bool rle8_decoder(FILE *in, FILE *out)
```

```
{
    if (in == NULL || out == NULL)
        return false;

    int c; // mixed declaration and code
    while ((c = fgetc(in)) != EOF) {
        int d = fgetc(in);
        if (d == EOF)
            return false;
        for (int i = 0; i < c; ++i) // declaration in for
                putchar(d);
    }

    return true;
}
```

Also the single line comment token "`//`" and the `inline` keyword uselful to hint the compiler about inlining some functions were inspired by C++.

The implicit function declarations and the implicit integers have been removed from the language for doing a more strict type checking.

Standard library

The C9x library provides 23 headers, as summarized in the table below:

<assert.h>	<float.h>	<setjmp.h>	<stdint.h>	<time.h>
<complex.h>	<iso646.h>	<signal.h>	<stdio.h>	<wchar.h>
<ctype.h>	<limits.h>	<stdarg.h>	<stdlib.h>	<wctype.h>
<errno.h>	<locale.h>	<stdbool.h>	<string.h>	
<fenv.h>	<math.h>	<stddef.h>	<tgmath.h>	

Table 3.2: The C9x standard library headers.

3.2 The C++ programming language

Bjarne Stroustrup invented C++ in the 1980s. The C++ language owes much to C, and it is retained as a subset. The other sources of inspiration for C++ are mainly Simula67 [DMN70, DH72] for the class concept, Algol68 [WB74] for the overloading operators, Ada [I+79] for templates and Ada, Clu [L+79] and ML [Wik87] for exception handling. The first implementation of C++ was called “C with Classes” [Str94] and have been in use since 1980. The name C++ was coined by Rick Mascitti in the 1983. The C++ language has been standardized in 1998 as the ISO C++ 14882:1998 standard, as result of the joint work of the ANSI X3J16 committee and the ISO JTC1/SC22/WG21 working group.

3.2.1 The main C++ features

The C++ language provides 74 keywords, as written in the table below:

`and`	`continue`	`goto`	`public`	`try`
`and_eq`	`default`	`if`	`register`	`typedef`
`asm`	`delete`	`inline`	`reinterpret_cast`	`typeid`
`auto`	`do`	`int`	`return`	`typename`
`bitand`	`double`	`long`	`short`	`union`
`bitor`	`dynamic_cast`	`mutable`	`signed`	`unsigned`
`bool`	`else`	`namespace`	`sizeof`	`using`
`break`	`enum`	`new`	`static`	`virtual`
`case`	`explicit`	`not`	`static_cast`	`void`
`catch`	`export`	`not_eq`	`struct`	`volatile`
`char`	`extern`	`operator`	`switch`	`wchar_t`
`class`	`false`	`or`	`template`	`while`
`compl`	`float`	`or_eq`	`this`	`xor`
`const`	`for`	`private`	`throw`	`xor_eq`
`const_cast`	`friend`	`protected`	`true`	

Table 3.3: The C++ keywords.

Object-oriented programming

In C++ an user can create new types that can be used as conveniently as the built-in types, using the class concept. For example, one user can write a `Point` class that records the coordinate position of a point on the screen and provide some useful manipulation functions for the object data:

```
class Point {
    int xpos, ypos;

public:
    Point(int x, int y) : xpos(x), ypos(y) {} // object constructor
    Point(const Point& p) :                   // copy data from
        xpos(p.xpos), ypos(p.ypos) {}         // another object

    int getx() const { return xpos; }         // return the X position
    int gety() const { return ypos; }         // return the Y position
    void addx(int delta) { xpos += delta; }   // add delta to X
    void addy(int delta) { ypos += delta; }   // add delta to Y
    void rotate(int rad, int rxpos = 0, int rypos = 0);
};

// Rotate the point of 'rad' radiants relative to the
// 'rxpos' and 'rypos' coordinates.
void Point::rotate(int rad, int rxpos, int rypos)
{
    // ...
}

// Return the distance between two points
int operator- (const Point& p1, const Point& p2)
{
    int dx = p1.getx() - p2.getx();
    int dy = p1.gety() - p2.gety();
    return sqrt(dx*dx + dy*dy);
}
```

The C++ language borrowed from Simula the concept of class hierarchies. It is possible to derive the characteristics of an object from another object (called *base class*) using the *deriving* technique. For example, we can define a `Means` structure:

```
struct Means {
    string name;                // The name of the means
    string manufacturer;        // The manufacturer of the means
    string color;               // The color of the means
    int kph;                    // Maximum Kilometers Per Hour
};
```

and derive from it the `Car` and `Plane` objects:

```
struct Car : public Means {
    bool have_abs;              // If the car has the ABS installed
    bool have_radio;            // If the car has a radio installed
    int no_of_doors;            // The number of doors of the car
};

struct Plane : public Means {
    bool have_gps;              // If the plane has the GPS installed
    int no_of_engines;          // The number of engines of the plane
};
```

A derived class can itself be a base class (such a set of related classes is traditionally called a *class hierarchy*); multiple derivations are also possible (often called *multiple inheritance*):

```
class Object { /* ... */ };
class HTMLStream { /* ... */ };
class View : public Object { /* ... */ };
class Window : public View { /* ... */ };
class HTMLViewer : public View, public HTMLStream { /* ... */ };
```

The class hierarchy created by these definitions is graphically visible in the figure below:

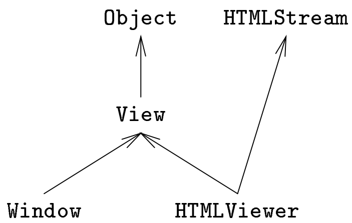

Figure 3.1: The class hierarchy created by the C++ definitions.

Generic programming

The generic programming features of the C++ language are taken partially from the Ada language. Generic programming is implemented in C++ using the *template* mechanism.

Suppose to write, for example, the typical `min` function:

```
int min(int a, int b)
{
    return (a > b) ? b : a;
}
```

Suppose now that you need the same function for a different type, like `double`, instead of `int`. In C you have to write another identical function that applies the same algorithm to the `double` type, and for any other type that is next required (this is easy to do for a `min` function, but think about a complex sorting function):

```
double double_min(double a, double b)
{
    return (a > b) ? b : a;
}
```

In C++ you can simply write this once:

```
template <class T>
T min(T a, T b)
{
    return (a > b) ? b : a;
}
```

Called *function template*, can be read "given a certain type `T`, let's define a function `min` that accepts two parameters, `a` and `b`, either of type `T`". On function invocation, the compiler generates the appropriate version of the function determining the types from the given parameters. While this is a rather simple application of generic programming, the same mechanism can be applied to classes:

```
#include <deque>
#include <iostream>

int main()
{
    std::deque<int> d;

    d.push_back(1);
    d.push_back(2);
    d.push_back(3);
    std::cout << "top element of deque is " << d.back() << '\n';
    d.pop_back();
    std::cout << "top element of deque is " << d.back() << '\n';
    d.pop_back();
```

```
    std::cout << "top element of deque is " << d.back() << '\n';
}
```

This will produce the following output:

```
top element of deque is 3
top element of deque is 2
top element of deque is 1
```

In this C++ example the user declares `d` as a `deque` of integers. A `deque` in C++ is a class template that can be defined for any predefined or user-defined type. The `deque` declaration, found in the `<deque>` standard header, is something like:

```
template <class T, class Allocator = allocator<T> >
class deque {
public:
    // ...
};
```

where `T` is the user specified type, while `Allocator` is the object memory allocator that can be different from the default `allocator` for particular objects.

Overloading

Overloading in C++ permits to write more functions sharing the same same, with different parameters and return values. For example, suppose to write a `print` function that writes the given type to the output plus the name of the type; in C++ we can write something like the following for the `int` and `char *` types:

```
#include <iostream>

void print(int i)
{
    std::cout << "int value = `" << i << "'\n";
}

void print(const char *s)
{
    std::cout << "string value = `" << s << "'\n";
}

int main()
{
    print(10);
```

```
    print("Dennis M. Ritchie");
}
```

This will output the following text:

```
int value = '10'
string value = 'Dennis M. Ritchie'
```

By the way, also statements like this:

```
std::cout << "string value = '" << s << "'\n";
```

are examples of overloading. In the standard headers we can probably find something like the following function that overloads the `<<` operator:

```
ostream& operator<< (ostream& os, const char *s)
{
    ...
    return os;
}
```

The first parameter is the left operand of the expression, while the second parameter is the right operand; the token after the `operator` keyword is the operator to overload.

Namespaces

Namespaces permit grouping declarations that logically belong together according to some criteria. For example, we can place the sorting algorithms into a common namespace `sorting`:

```
namespace sorting {
    template <class Iter> void bubble_sort(Iter begin, Iter end);
    template <class Iter> void shell_sort(Iter begin, Iter end);
    template <class Iter> void quick_sort(Iter begin, Iter end);
};
```

and call the proper functions when needed:

```
void f()
{
    std::vector<int> v;
    ...
    sorting::bubble_sort(v.begin(), v.end());
}
```

In general, namespaces are very useful when using third-parties libraries avoiding name space conflicts. By the way, all the C++ standard library functions are grouped into the `std` namespace.

Exception handling

Exception handling is useful in C++ for avoiding the overhead of checking the error status returned by every single function and for eliminating the need to resolve the error conditions locally. For example, the following C++ code:

```
#include <cctype>
#include <iostream>

struct Conversion_error {
    const char *s;
    Conversion_error(const char *xs) : s(xs) {}
};

int hex_to_dec(char c)
{
    if (!std::isxdigit(c))
        throw Conversion_error("character is not an hex digit");
    if (std::isdigit(c))
        return c - '0';
    return std::tolower(c) - 'a' + 10;
}

int main()
{
    using std::cout;
    using std::cerr;

    try {
        cout << "3 == " << hex_to_dec('3') << '\n';
        cout << "a == " << hex_to_dec('a') << '\n';
        cout << "F == " << hex_to_dec('F') << '\n';
        cout << "j == " << hex_to_dec('j') << '\n';
    } catch (Conversion_error x) {
        cerr << "Caught conversion error: " << x.s << '\n';
    } catch (...) {
        cerr << "Unexpected error\n";
    }
}
```

will produce the following output:

```
3 == 3
a == 10
F == 15
Caught conversion error: character is not an hex digit
```

The error is signaled in `hex_to_dec` using the `throw` keyword after the code in the `try` block has been executed; it is then caught in the first `catch` block. The second `catch` block, with the ellipsis as the argument, means "catch everything else"; it is usually used for catching unexpected errors.

Standard library

The ISO C++ standard library provides the same ISO C library facilities plus a number of new functions providing support for the language features, and a set of generic containers implemented using the template technique. The C++ standard library provides 32 C++ headers, as summarized by table 3.4; the facilities of the standard C library are provided in 18 additional headers, as summarized by table 3.5.

`<algorithm>`	`<iomanip>`	`<list>`	`<ostream>`	`<streambuf>`
`<bitset>`	`<ios>`	`<locale>`	`<queue>`	`<string>`
`<complex>`	`<iosfwd>`	`<map>`	`<set>`	`<typeinfo>`
`<deque>`	`<iostream>`	`<memory>`	`<sstream>`	`<utility>`
`<exception>`	`<istream>`	`<new>`	`<stack>`	`<valarray>`
`<fstream>`	`<iterator>`	`<numeric>`	`<stdexcept>`	`<vector>`
`<functional>`	`<limits>`			

Table 3.4: The C++ standard library headers.

`<cassert>`	`<ciso646>`	`<csetjmp>`	`<cstdio>`	`<ctime>`
`<cctype>`	`<climits>`	`<csignal>`	`<cstdlib>`	`<cwchar>`
`<cerrno>`	`<clocale>`	`<cstdarg>`	`<cstring>`	`<cwctype>`
`<cfloat>`	`<cmath>`	`<cstddef>`		

Table 3.5: The C++ standard C library headers.

The generic containers are designed to be simple and efficient, and each container provides a set of standard operations with standard names and semantics. Iterators are provided for iterating through the containers elements using well defined standards operations.

For example, the following code:

```
#include <iostream>
#include <string>

int main()
{
    using std::string;
    using std::cout;

    string s = "hello world";

    typedef string::const_iterator I;   // Iterate through the elements
    for (I i = s.begin(); i != s.end(); ++i)
        cout << *i;
    cout << '\n';

    typedef string::reverse_iterator R; // Iterate in the reverse order
    for (R i = s.rbegin(); i != s.rend(); ++i)
        cout << *i;
    cout << '\n';
}
```

will produce the following output:

```
hello world
dlrow olleh
```

writing the string from left to right, character by character, and from right to left, using an iterator and a reverse iterator, respectively.

Chapter 4

A Few Words About C Compilers

4.1 The user-visible translation phases

A typical UNIX C translation is composed by four user-visible phases, as summarized by the figure below:

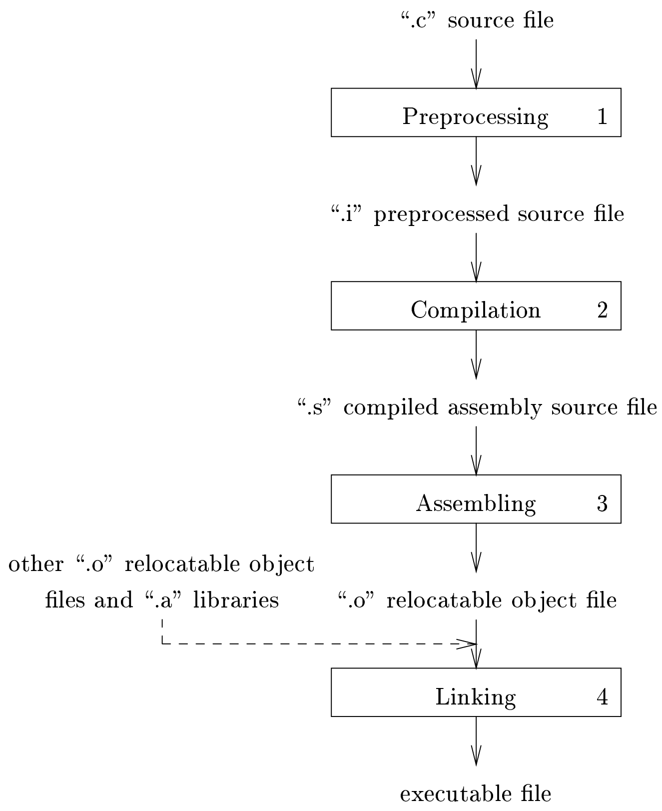

Figure 4.1: The typical user-visible UNIX translation phases.

1. Preprocessing. In the first phase the user C source file (with extension "`.c`") is preprocessed by the C preprocessor[1]. The trigraphs are replaced with the corresponding characters, the backslash-newline sequences are removed, and the comments are replaced with one space character. The macros are expanded,

[1] Usually named `cpp`, under UNIX the C preprocessor is a standalone command line program that can be used for other different preprocessing applications, like as a preprocessor for a Fortran program.

the preprocessor directives are executed and the `#include` directives recursively include the user specified files. At the end of this process a preprocessed "`.i`" file is produced.

2. Compilation. In the second phase the preprocessed "`.i`" file is compiled into the target machine assembly language by the real C compiler, generating a "`.s`" assembly source file.

3. Assembling. In the third phase the compiled "`.s`" file is assembled by the assembler utility, and a relocatable object file "`.o`" is produced.

4. Linking. In the fourth phase the "`.o`" user relocatable object files are linked together with the "`.a`" standard libraries and the user-supplied ones, producing a machine executable file[2].

4.2 The internal compiler translation phases

While the previous described actions are visible to the user, the internal compiler phases are not so simple to identify, as they are visible only to the compiler implementators. A typical internal compiler translation is composed by many complex phases. In a few words, a compiler must recognize the mean of the sequence of character of the input source file, generate the corresponding machine possibly doing a few optimizations to it for speeding up the execution and to reduce the size of the code itself.

For example, the following phases are very common in compilers in general, as summarized by the figure 4.2 on the next page:

1. Lexical analysis. Often known as *scanning*, this phase includes recognizing the mean of the single input characters, and collecting the recognized sequences into tokens. A token is a single language entity, an atomic unit. For example, in the following C input:

   ```
   int vec[3];
   ```

   we can distinguish six tokens: the `int` keyword, the `vec` identifier, the open bracket, the `3` integer, the closing bracket and the semicolon; the whitespaces are not tokens and are ignored by the lexical analyzer. In the following input:

   ```
   int $p;
   ```

[2] Under UNIX an `a.out` file is generated if no name was specified by the user for the output executable.

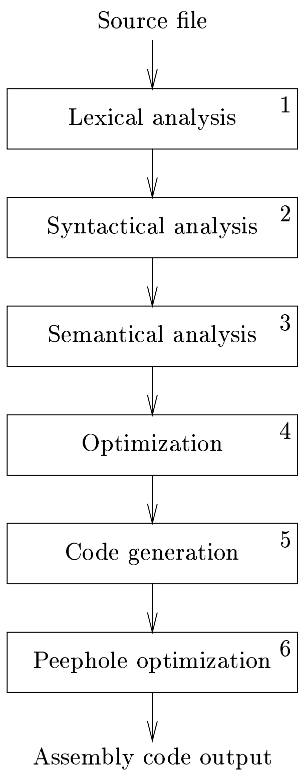

Figure 4.2: An example of internal compiler translation phases.

the lexical analyzer will print a syntax error because the `$` character is not known in the C language as a token.

2. Syntactical analysis. Also known as *parsing*, this phase includes recognizing the mean of a sequence of tokens, in the order they appear, and building the so called *parse tree*. For example, in the following input (appearing in the body of a function):

```
a = b + c;
```

the syntactical analyzer (*parser*) will receive from the lexical analyzer a sequence of six tokens. The parser will then recognize that this sequence is an additive expression of `b` and `c` which resulting value should be stored in `a`; the expression is then terminated correctly by a semicolon. The set of valid token sequences is called grammar. In the following input:

```
p* char;
```

the lexical analyzer will emit no syntax error, because all the four tokens are recognized as correct, while the parser will emit a parsing error because this token sequence has no sense in the C grammar.

Another more complex example may be this:

```
1   void f(void)
2   {
3       int a;
4       a = 1;
5       int b;
6       b = 2;
7   }
```

The parser will emit a parsing error at line 5, because the grammar says that a variable definition is only accepted after the opening brace of a block of code, in this example at line 3 (by the way this code is accepted by a C++ compiler, since the C++ grammar permit such definitions).

3. Semantical analysis. In this phase the semantical analyzer checks for the intrinsic mean of the parse tree, for example the correctness of the types in an expression or if a variable has been declared previously. In the following input:

```
void f(void)
{
```

```
3       int a, c;
4       struct { int x; } b;
5
6       c = a + b;
7       d = 1;
8   }
```

the lexical analyzer will emit no syntax error, nor the syntactic analyzer, while the semantic analyzer will catch two semantic errors. The first error is at line 6, where the `a` and `b` variables in the addition expression have two different incompatible types. The second error is at line 7, where the value 1 is assigned to the undeclared variable `d`.

4. Optimization. In this phase the parse tree, generated by the parser, is searched and modified for optimizing issues, like execution speed and generated code size. See §4.3.1 for more information.
5. Code generation. In this phase the parse tree is translated (known as *emitting*) in the target machine code, statement by statement. Target dependent optimizations can be easily done just before the effective code generation.
6. Peephole optimization. In this low level phase, the generated code is scanned for useless or redundant machine instructions. See §4.3.2 on page 137 for more information.

4.3 Code optimization

Code optimization is a technique that permits speeding up the execution of the generated code and to reduce the size of the code itself. There are many types of optimizations, so the most common ones are summarized in this section. We can distinguish the optimizations in two categories, the optimizations done before the code generation, and the optimization done after the code generation. An excellent reference book on advanced optimizations is [Muc97].

4.3.1 Pre code-generation optimizations

This subsection includes the optimizations performed by the compiler before the machine code generation.

Constant folding

This optimization tries to evaluate at compile time the expressions whose values are known to be constant. For example, in the following statement:

```
a = 12 * 2 + 3;
```

the optimizer can safely evaluate the expression to:

```
a = 27;
```

Algebraic simplifications and reassociation

This optimization uses algebraic properties of operators to simplify expressions. For example, the following expression:

```
a = -(0 - (b >> 0)) * -1;
```

can be safely evaluated by the optimizer to:

```
a = -b;
```

Copy propagation

This optimization involves that, after an assignment `a = b`, later uses of `a` are replaced with `b`, as long as the value of `a` has not been changed by other instructions. For example, the following expressions:

```
a = b;
c = a;
```

can be optimized to:

```
a = b;
c = b;
```

But the following expression cannot be optimized with copy propagation:

```
a = b;
if (x > 10)
    a = c;
d = a;
```

Common-subexpression elimination

This optimization involves that an expression that occur two or more times in a block[3], and whose evaluation is constant in that block, can be evaluated only one time. For example, the following code:

[3] If the scope of the expression is the function, the optimization is called global common-subexpression elimination, otherwise if the scope is a single basic block, it is called local common-subexpression elimination

```
a = b * 2;
if (x > 3)
    a = 1;
c = b * 2;
```

can be optimized to:

```
t1 = b * 2;
a = t1;
if (x > 3)
    a = 1;
c = t1;
```

where `t1` is a temporary compiler generated variable. But the following code cannot be optimized:

```
a = b * 2;
if (x > 3)
    b = 1;
c = b * 2;
```

Loop-invariant code motion

This optimization tries to move out of a loop the expressions that evaluates to the same value on every iteration of the loop. For example, the following code:

```
for (a = 1; a < 5; ++a)
    b += c * 20 + a;
```

can be optimized to:

```
t1 = c * 20;
for (a = 1; a < 5; ++a)
    b += t1 + a;
```

The global common-subexpression elimination and loop-invariant code motion optimizations can be combined in another technique, the partial-redundancy optimization, that permits some additional code improvements as well.

Code hoisting

This optimization permits moving duplicated expressions that evaluate to the same value out of different execution paths. The resulting code may be as fast as or slower than the original code but this depends on several factors.

For example, the following code:

```
if (x > 10) {
    e = a + b;
    f = c + d;
} else {
    if (y < 5)
        g = a + b;
    else
        h = c + d;
}
```

can be optimized to:

```
t1 = a + b;
t2 = c + d;
if (x > 10) {
    e = t1;
    f = t2;
} else {
    if (y < 5)
        g = t1;
    else
        h = t2;
}
```

Removal of induction-variables

This optimization locates the variables whose successive values form an arithmetic progression in a part of a program and tries to remove the unnecessary progressions. For example, the following code:

```
for (i = 0, j = 1; i < 10; ++i, ++j)
    v[i] = j;
```

can be optimized to:

```
for (i = 0; i < 10; ++i)
    v[i] = i + 1;
```

Strength reduction

This optimization tries to replace the expensive operations, such as multiplications and divisions, by less expensive ones, such as additions and subtractions. For example, the following code:

```
for (i = 0; i < 10; ++i)
    v[i] = 4 * i;
```

can be optimized to:

```
t1 = 0;
for (i = 0; i < 10; ++i) {
    v[i] = t1;
    t1 += 4;
}
```

Tail-recursion elimination

This optimizations tries to transform a self-calling recursive function into non-recursive one. This can be done easily if the recursive call is at the tail of the function. For example, the following code:

```
void append(struct symbol *p1, struct symbol *p2)
{
    if (p1->next == NULL)
        p1->next = p2;
    else
        append(p1->next, p2);
}
```

can be easily optimized to:

```
void append(struct symbol *p1, struct symbol *p2)
{
l1: if (p1->next == NULL)
        p1->next = p2;
    else {
        p1 = p1->next;
        goto l1;
    }
}
```

Tail-call optimization

This optimization tries to convert a tail call to a function to a simple jump into that function. For example, suppose that the following code:

```
void f1(int a, int b)
{
```

```
    ...
}

void f2(int a, int b)
{
    ...
    f1(a, b);
}
```

generates the following hypothetical code:

```
_f1:
    ...
    return        return to the caller

_f2:
    ...
    push r0       push the parameters on the stack
    push r1
    call _f1      call the function
    ...
    return        return to the caller
```

This machine code can be optimized to:

```
_f1:
    ...
    return        return to the caller

_f2:
    ...
    goto _f1      jump into the called function
```

In-line expansion

This optimization permits the integration of short functions into other functions avoiding the overhead of a function call. For example, the following code:

```
int abs(int x)
{
    return (x >= 0) ? x : -x;
}
```

```
int f(int x)
{
    if (abs(x) > 10)
        return 1;
    return 0;
}
```

can be optimized to:

```
int f(int x)
{
    if (((x >= 0) ? x : -x) > 10)
        return 1;
    return 0;
}
```

Register allocation

This optimization determines which of the program values like variables and constants should be stored in the machine registers at each point in the execution of a program. In the C language the programmer can give the compiler a hint using the `register` keyword. In old C code is common to find the `register` keyword, like in the following function:

```
char *strcat(char *dest, const char *src)
{
    register char *dp = dest;
    register const char *sp = src;

    while (*dp != '\0')
        ++dp;
    while ((*dp++ = *sp++) != '\0')
        ;

    return dest;
}
```

In the present days the `register` usage is discouraged, because the most modern C compilers are now able to recognize which variables should be stored in registers and which not; some compilers may also ignore the programmer hints and decide its own assignments to registers.

Unreachable-code elimination

This optimization locates code that cannot possibly be executed, regardless of the input data. For example, in the following code:

```
if (x > 0)
    a = 1, b = 0;
else {
    a = 0;
    b = a + 1;
}
if (a == b) {
    ...
}
```

the last block of code can be eliminated, because a and b will never be equal.

Dead-code elimination

This optimization removes all the instructions that computes only values that are not used on any executable path leading from the instruction. For example, the following function:

```
void f(void)
{
    int i = 1, j, k, w = 5;
    for (j = 0, k = 1; j < i; ++j)
        k += j;
    other_func(w);
}
```

can be simplified to:

```
void f(void)
{
    int w = 5;
    other_func(w);
}
```

and than optimized again removing the w variable definition and replacing the function argument with the value 5.

Loop unrolling

This optimization replaces the simple loops by nonlooping code determining how many times the loop is executed. For example, the following code:

```
for (i = 0, j = 2; i < 3; ++i, j += 2)
        other_func(i, j);
```

can be unrolled to:

```
    i = 0;
    j = 2;
    other_func(i, j);
    ++i;
    j += 2;
    other_func(i, j);
    ++i;
    j += 2;
    other_func(i, j);
    ++i;
    j += 2;
```

or evaluated at compile time:

```
    other_func(0, 2);
    other_func(1, 4);
    other_func(2, 6);
    i = 3;
    j = 8;
```

The last two statements of this two examples can be eliminated by the dead-code elimination optimization if the two variables `i` and `j` are never referenced later.

A clever compiler may evaluate a complex statement like this:

```
for (a = 1; a < 5; ++a)
    b += c * 20 + a;
```

to the simpler expression

```
b += c * 80 + 10;
```

combining various optimizing techniques (assuming that the value of `a` is not used after the loop).

Code scheduling

The most modern machines (RISC and later CISC machines, including Intel 80486, Pentium and Pentium Pro) are able to execute more than one instruction at once. The order of the generated instruction is then more important than in the past and methods for scheduling or reordering instructions are required to improve performance taking advantage of the native scheduling and pipelining features.

4.3.2 Post code-generation optimizations

The most common optimization done after code generation is the **peephole optimization**. In this phase the peephole optimizer scans the compiler generated code for useless, redundant or inappropriate machine instructions. When the peephole optimizer is a standalone program separated from the compiler, the compiler output code is usually generated in the assembler source format, so the peephole optimizer can be written with any text processing language, like AWK or Perl.

Usually a pattern-matching approach is used; for example, from the following C code:

```
int a, b;
void f(void)
{
    a = 1;
    b = a;
    return;
}
```

a simple C compiler may generate the following hypothetical machine code:

```
.global _a(4)
.global _b(4)
_f:
...
    mov 1, r0           store the value 1 into the r0 register
    mov r0, *_a         store the r0 register into _a
    mov *_a, r0         store the value from _a into r0
    mov r0, *_b         store the r0 register into _b
    goto _f_end
_f_end:
...
    return
```

The third machine instruction "`mov *_a, r0`" of the machine code can be easily thrown away, because the `_a` value is already stored in the `r0` register. The useless `goto` generated by the `return` statement can also be removed. An hypothetical pattern that optimizes this code can be the following:

```
match [
    mov r@1, *@2
    mov *@2, r@1
] replace [
    mov r@1, *@2
]
match [
    goto @1:
@1:
] replace [
@1:
]
```

where the `@` characters, followed by a digit, means "match everything". Applied, the peephole pattern would filter the generated code to:

```
.global _a(4)
.global _b(4)
_f:
...
    mov 1, r0
    mov r0, *_a
    mov r0, *_b
_f_end:
...
    return
```

This type of optimization is usually not done by simple compilers at the fourth optimization phase, because redundant instructions may be generated by different statements.

4.4 Compiler generators

The lexical analyzer and the syntactic analyzer are usually written with the help of generators. The most common lexical and parser generators are `lex` and `yacc`. The first tool, `lex`, generates a complete scanner from a written description file of the tokens needed to be recognized. The generated scanner provides a `yylex` interfacing function. This function can be called sequentially and returns, one per time, the tokens recognized in the input stream of characters.

Appendix A

Summary of the ISO C Headers

A.1 Introduction

This chapter provides a summary of all the eighteen standard headers, useful for reference when writing a C library implementation.

The symbols with a *implementation defined* comment like this

```
#define offsetof(t, m)   implementation defined
```

are to be defined by the implementator since a standard definition does not exist because is not portable between different architectures or because different implementation techniques exist.

A.2 The <assert.h> header

```
#undef assert

#ifdef NDEBUG
#define assert(ignore)   implementation defined null expression
#else
#define assert(e)        implementation defined assertion
#endif
```

A.3 The <ctype.h> header

```
#ifndef __CTYPE_H
#define __CTYPE_H

int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

```
int tolower(int c);
int toupper(int c);

#endif /* !__CTYPE_H */
```

A.4 The <errno.h> header

```
#ifndef __ERRNO_H
#define __ERRNO_H

#define EDOM            implementation defined
#define EILSEQ          implementation defined
#define ERANGE          implementation defined

#define errno           implementation defined

#endif /* !__ERRNO_H */
```

A.5 The <float.h> header

```
#ifndef __FLOAT_H
#define __FLOAT_H

#define FLT_ROUNDS      implementation defined
```

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

```
#define FLT_DIG         6
#define FLT_MANT_DIG    implementation defined
#define FLT_MAX_10_EXP  +37
#define FLT_MAX_EXP     implementation defined
#define FLT_MIN_10_EXP  -37
#define FLT_MIN_EXP     implementation defined
#define FLT_RADIX       2
#define DBL_DIG         10
```

```
#define DBL_MANT_DIG     implementation defined
#define DBL_MAX_10_EXP  +37
#define DBL_MAX_EXP      implementation defined
#define DBL_MIN_10_EXP  -37
#define DBL_MIN_EXP      implementation defined
#define LDBL_DIG         10
#define LDBL_MANT_DIG    implementation defined
#define LDBL_MAX_10_EXP +37
#define LDBL_MAX_EXP     implementation defined
#define LDBL_MIN_10_EXP -37
#define LDBL_MIN_EXP     implementation defined
```

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal or greater than those shown.

```
#define FLT_MAX          1E+37
#define DBL_MAX          1E+37
#define LDBL_MAX         1E+37
```

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal or less than those shown.

```
#define FLT_EPSILON      1E-5
#define FLT_MIN          1E-37
#define DBL_EPSILON      1E-9
#define DBL_MIN          1E-37
#define LDBL_EPSILON     1E-9
#define LDBL_MIN         1E-37

#endif /* !__FLOAT_H */
```

A.6 The `<iso646.h>` header

```
#ifndef __ISO646_H
#define __ISO646_H

#define and            &&
#define and_eq         &=
#define bitand         &
#define bitor          |
#define compl          ~
#define not            !
#define not_eq         !=
#define or             ||
#define or_eq          |=
#define xor            ^
#define xor_eq         ^=

#endif /* !__ISO646_H */
```

A.7 The `<limits.h>` header

```
#ifndef __LIMITS_H
#define __LIMITS_H
```

The minimum magnitudes shown shall be replaced by implementation-defined magnitudes with the same sign.

```
#define CHAR_BIT       8
#define CHAR_MAX       UCHAR_MAX or SCHAR_MAX
#define CHAR_MIN       0 or SCHAR_MIN
#define INT_MAX        +32767
#define INT_MIN        -32767
#define LONG_MAX       +2147483647
#define LONG_MIN       -2147483647
#define MB_LEN_MAX     1
#define SCHAR_MAX      +127
#define SCHAR_MIN      -127
#define SHRT_MAX       +32767
#define SHRT_MIN       -32767
#define UCHAR_MAX      255
```

```
#define UINT_MAX        65535
#define ULONG_MAX       4294967295
#define USHRT_MAX       65535

#endif /* !__LIMITS_H */
```

A.8 The <locale.h> header

```
#ifndef __LOCALE_H
#define __LOCALE_H

#define LC_ALL          implementation defined
#define LC_COLLATE      implementation defined
#define LC_CTYPE        implementation defined
#define LC_MONETARY     implementation defined
#define LC_NUMERIC      implementation defined
#define LC_TIME         implementation defined
#define NULL            implementation defined

struct lconv {
    char *decimal_point;
    char *thousands_sep;
    char *grouping;
    char *int_curr_symbol;
    char *currency_symbol;
    char *mon_decimal_point;
    char *mon_thousands_sep;
    char *mon_grouping;
    char *positive_sign;
    char *negative_sign;
    char int_frac_digits;
    char frac_digits;
    char p_cs_precedes;
    char p_sep_by_space;
    char n_cs_precedes;
    char n_sep_by_space;
    char p_sign_posn;
    char n_sign_posn;
};
```

```
char *setlocale(int category, const char *locale);
struct lconv *localeconv(void);

#endif /* !__LOCALE_H */
```

A.9 The <math.h> header

```
#ifndef __MATH_H
#define __MATH_H

#define HUGE_VAL        implementation defined

double acos(double x);
double asin(double x);
double atan(double x);
double atan2(double y, double x);
double cos(double x);
double sin(double x);
double tan(double x);
double cosh(double x);
double sinh(double x);
double tanh(double x);
double exp(double x);
double frexp(double x, int *iptr);
double ldexp(double x, int exp);
double log(double x);
double log10(double x);
double modf(double x, int *iptr);
double pow(double x, double y);
double sqrt(double x);
double fabs(double x);
double floor(double x);
double ceil(double x);
double fmod(double x, double y);

#endif /* !__MATH_H */
```

A.10 The <setjmp.h> header

```
#ifndef __SETJMP_H
#define __SETJMP_H

typedef implementation defined jmp_buf;

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);

#endif /* !__SETJMP_H */
```

A.11 The <signal.h> header

```
#ifndef __SIGNAL_H
#define __SIGNAL_H

typedef implementation defined sig_atomic_t;

#define SIG_DFL        implementation defined
#define SIG_ERR        implementation defined
#define SIG_IGN        implementation defined
#define SIGABRT        implementation defined
#define SIGFPE         implementation defined
#define SIGILL         implementation defined
#define SIGINT         implementation defined
#define SIGSEGV        implementation defined
#define SIGTERM        implementation defined

void (*signal(int sig, void (*func)(int)))(int);
int raise(int sig);

#endif /* !__SIGNAL_H */
```

A.12 The <stdarg.h> header

```
#ifndef __STDARG_H
#define __STDARG_H

typedef implementation defined va_list;

#define va_start(ap, paramN)    implementation defined
#define va_arg(ap, type)        implementation defined
#define va_end(ap)              implementation defined

#endif /* !__STDARG_H */
```

A.13 The <stddef.h> header

```
#ifndef __STDDEF_H
#define __STDDEF_H

#define NULL            implementation defined

typedef implementation defined ptrdiff_t;
typedef implementation defined size_t;
typedef implementation defined wchar_t;

#define offsetof(t, m)  implementation defined

#endif /* !__STDDEF_H */
```

A.14 The <stdio.h> header

```
#ifndef __STDIO_H
#define __STDIO_H

#define _IOFBF          implementation defined
#define _IOLBF          implementation defined
#define _IONBF          implementation defined
#define BUFSIZ          implementation defined
#define EOF             implementation defined
#define FILENAME_MAX    implementation defined
#define FOPEN_MAN       implementation defined
#define L_tmpnam        implementation defined
#define NULL            implementation defined
#define SEEK_CUR        implementation defined
#define SEEK_END        implementation defined
#define SEEK_SET        implementation defined
#define TMP_MAX         implementation defined

typedef implementation defined FILE;
typedef implementation defined fpos_t;
typedef implementation defined size_t;

FILE *stderr;
FILE *stdin;
FILE *stdout;

int remove(const char *filename);
int rename(const char *old, const char *new);
FILE *tmpfile(void);
char *tmpnam(char *s);
int fclose(FILE *stream);
int fflush(FILE *stream);
FILE *fopen(const char *filename, const char *mode);
FILE *freopen(const char *filename, const char *mode,
              FILE *stream);
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

```
int printf(const char *format, ...);
int scanf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
int vfprintf(FILE *stream, const char *format, va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format, va_list arg);
int fgetc(FILE *stream);
char *fgets(char *s, int n, FILE *stream);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *s);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
int ungetc(int c, FILE *stream);
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t n,
              FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
long int ftell(FILE *stream);
void rewind(FILE *stream);
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);

#endif /* !__STDIO_H */
```

A.15 The <stdlib.h> header

```
#ifndef __STDLIB_H
#define __STDLIB_H

#define EXIT_FAILURE    implementation defined
#define EXIT_SUCCESS    implementation defined
#define MB_CUR_MAX      implementation defined
#define NULL            implementation defined
#define RAND_MAX        implementation defined

typedef struct {
    int quot, rem;
} div_t;
typedef struct {
    long quot, rem;
} ldiv_t;
typedef implementation defined size_t;
typedef implementation defined wchar_t;

double atof(const char *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
double strtod(const char *nptr, char **endptr);
long int strtol(const char *nptr, char **endptr, int base);
unsigned long int strtoul(const char *nptr, char **endptr, int base);
int rand(void);
void srand(unsigned int seed);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void abort(void);
int atexit(void (*func)(void));
void exit(int status);
char *getenv(const char *name);
int system(const char *string);
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
int abs(int j);
div_t div(int number, int denom);
long int labs(long int j);
ldiv_t ldiv(long int number, long int denom);
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);

#endif /* !__STDLIB_H */
```

A.16 The <string.h> header

```
#ifndef __STRING_H
#define __STRING_H

#define NULL              implementation defined

typedef implementation defined size_t;

void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char *s1, const char *s2, size_t n);
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
void *memset(void *s, int c, size_t n);
char *strerror(int errnum);
size_t strlen(const char *s);

#endif /* !__STRING_H */
```

A.17 The <time.h> header

```
#ifndef __TIME_H
#define __TIME_H

#define CLOCKS_PER_SEC  implementation defined
#define NULL            implementation defined

typedef implementation defined clock_t;
typedef implementation defined time_t;
typedef implementation defined size_t;

struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};

clock_t clock(void);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm *timeptr);
time_t time(time_t *time);
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timeptr);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
size_t strftime(char *s, size_t maxsize, const char *format,
                const struct tm *timeptr);

#endif /* !__TIME_H */
```

A.18 The <wchar.h> header

```
#ifndef __WCHAR_H
#define __WCHAR_H

#define NULL            implementation defined
#define WEOF            implementation defined
#define WCHAR_MAX       implementation defined
#define WCHAR_MIN       implementation defined

typedef implementation defined size_t;
typedef implementation defined wchar_t;
typedef implementation defined wint_t;
typedef implementation defined mbstate_t;

wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
int fputws(const wchar_t *s, FILE *stream);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t ungetwc(wint_t c, FILE *stream);
double wcstod(const wchar_t *nptr, wchar_t **endptr);
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr,
                          int base);
wchar_t *wcscpy(wchar_t *s1, const wchar_t *s2);
wchar_t *wcsncpy(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);
wchar_t *wcsncat(wchar_t *s1, const wchar_t *s2, size_t n);
int wcscmp(const wchar_t *s1, const wchar_t *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
size_t wcsxfrm(wchar_t *s1, const wchar_t *s2, size_t n);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);e
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
size_t wcslen(const wchar_t *s);
wchar_t *wcschr(const wchar_t *s, wchar_t c);
wchar_t *wcsrchr(const wchar_t *s1, wchar_t c);
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
int fwprintf(FILE *stream, const wchar_t *format, ...);
int fwscanf(FILE *stream, const wchar_t *format, ...);
```

```
int wprintf(const wchar_t *format, ...);
int wscanf(const wchar_t *format, ...);
int swprintf(wchar_t *s, size_t n, const wchar_t *format, ...);
int swscanf(const wchar_t *s, const wchar_t *format, ...);
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);
int vwprintf(const wchar_t *format, va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format,
              va_list arg);
wint_t fputwc(wchar_t c, FILE *stream);
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
int fwide(FILE *stream, int mode);
extern wchar_t *wcstok(wchar_t *s1, const wchar_t *s2, wchar_t **ptr);
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
size_t wcsftime(wchar_t *s, size_t n, const wchar_t *format,
                const struct tm *timeptr);
wint_t btowc(int c);
int wctob(wint_t c);
int mbsinit(const mbstate_t *ps);
size_t mbrlen(const char *s, size_t n, mbstate_t *pcs);
size_t mbrtowc(wchar_t *ws, const char *s, size_t n, mbstate_t *ps);
size_t wcrtomb(char *s, wchar_t c, mbstate_t *s);
size_t mbsrtowcs(wchar_t *ws, const char **src, size_t n,
                 mbstate_t *ps);
size_t wcsrtombs(char *s, const wchar_t **src, size_t n,
                 mbstate_t *ps);

#endif /* !__WCHAR_H */
```

A.19 The <wctype.h> header

```
#ifndef __WCTYPE_H
#define __WCTYPE_H

#define WEOF            implementation defined

typedef implementation defined wint_t;
typedef implementation defined wctype_t;
typedef implementation defined wctrans_t;

int iswalnum(wint_t c);
int iswalpha(wint_t c);
int iswcntrl(wint_t c);
int iswdigit(wint_t c);
int iswgraph(wint_t c);
int iswlower(wint_t c);
int iswprint(wint_t c);
int iswpunct(wint_t c);
int iswspace(wint_t c);
int iswupper(wint_t c);
int iswxdigit(wint_t c);
wint_t towlower(wint_t c);
wint_t towupper(wint_t c);
wctype_t wctype(const char *property);
int iswctype(wint_t c, wctype_t desc);
wctrans_t wctrans(const char *property);
wint_t towctrans(wint_t c, wctrans_t desc);

#endif /* !__WCTYPE_H */
```

Appendix B

The ISO C Grammar

B.1 Introduction

This appendix describes a full ISO C grammar. The symbols typeset in the `typewriter` font are literal terminal symbols; the symbols typeset in the *italic* font are undefined terminal symbols. The *identifier* and *typedef-name* terminals can be distinguished only during the translation phase, using a symbol table. This grammar, processed by a LALR(1) compiler-generator (like `yacc`), contains one conflict in the *selection-statement* (§B.2.3) production, generated by the `if-else` construct). The expressions that contain the "$_{opt}$" suffix need to be duplicated, one time with the optional symbol, one time without. The topmost production of the grammar is *translation-unit* (§B.2.4).

B.2 The grammar

B.2.1 Expressions

primary-expression:
    identifier
    constant
    string-literal
    `(` expression `)`

postfix-expression:
    primary-expression
    postfix-expression `[` expression `]`
    postfix-expression `(` argument-expression-list$_{opt}$ `)`
    postfix-expression `.` *identifier*
    postfix-expression `->` *identifier*
    postfix-expression `++`
    postfix-expression `--`

argument-expression-list:
    assignment-expression
    argument-expression-list `,` assignment-expression

unary-expression:
    postfix-expression
    `++` unary-expression
    `--` unary-expression
    unary-operator cast-expression
    `sizeof` unary-expression
    `sizeof` `(` type-name `)`

unary-operator: one of
    & * + - ~ !

cast-expression:
    unary-expression
    (type-name) cast-expression

multiplicative-expression:
    cast-expression
    multiplicative-expression * cast-expression
    multiplicative-expression / cast-expression
    multiplicative-expression % cast-expression

additive-expression:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression

shift-expression:
    additive-expression
    shift-expression << additive-expression
    shift-expression >> additive-expression

relational-expression:
    shift-expression
    relational-expression < shift-expression
    relational-expression > shift-expression
    relational-expression <= shift-expression
    relational-expression >= shift-expression

equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression

and-expression:
    equality-expression
    and-expression & equality-expression

exclusive-or-expression:
  and-expression
  exclusive-or-expression `^` and-expression

inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression `|` exclusive-or-expression

logical-and-expression:
  inclusive-or-expression
  logical-and-expression `&&` inclusive-or-expression

logical-or-expression:
  logical-and-expression
  logical-or-expression `||` logical-and-expression

conditional-expression:
  logical-or-expression
  logical-or-expression `?` expression `:` conditional-expression

assignment-expression:
  conditional-expression
  unary-expression assignment-operator assignment-expression

assignment-operator: one of
  `= *= /= %= += -= <<= >>= &= ^= |=`

expression:
  assignment-expression
  expression `,` assignment-expression

constant-expression:
  conditional-expression

B.2.2 Declarations

declaration:
  declaration-specifiers init-declarator-list$_{opt}$ `;`

declaration-specifiers:
    storage-class-specifier declaration-specifiers$_{opt}$
    type-specifier declaration-specifiers$_{opt}$
    type-qualifier declaration-specifiers$_{opt}$

init-declarator-list:
    init-declarator
    init-declarator-list `,` init-declarator

init-declarator:
    declarator
    declarator `=` initializer

storage-class-specifier:
    `typedef`
    `extern`
    `static`
    `auto`
    `register`

type-specifier:
    `void`
    `char`
    `short`
    `int`
    `long`
    `float`
    `double`
    `signed`
    `unsigned`
    struct-or-union-specifier
    enum-specifier
    typedef-name

struct-or-union-specifier:
    struct-or-union *identifier*$_{opt}$ `{` struct-declaration-list `}`
    struct-or-union *identifier*

struct-or-union:
    `struct`
    `union`

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration

struct-declaration:
    specifier-qualifier-list struct-declarator-list `;`

specifier-qualifier-list:
    type-specifier specifier-qualifier-list$_{opt}$
    type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
    struct-declarator
    struct-declarator-list `,` struct-declarator

struct-declarator:
    declarator
    declarator$_{opt}$ `:` constant-expression

enum-specifier:
    `enum` *identifier*$_{opt}$ `{` enumerator-list `}`
    `enum` *identifier*

enumerator-list:
    enumerator
    enumerator-list `,` enumerator

enumerator:
    identifier
    identifier `=` constant-expression

type-qualifier:
    `const`
    `volatile`

declarator:
    pointer$_{opt}$ direct-declarator

direct-declarator:
    identifier
    (declarator)
    direct-declarator [constant-expression$_{opt}$]
    direct-declarator (parameter-type-list)
    direct-declarator (identifier-list$_{opt}$)

pointer:
    * type-qualifier-list$_{opt}$
    * type-qualifier-list$_{opt}$ pointer

type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier

parameter-type-list:
    parameter-list
    parameter-list , ...

parameter-list:
    parameter-declaration
    parameter-list , parameter-declaration

parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
    identifier
    identifier-list , *identifier*

type-name:
    specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
    pointer
    pointer$_{opt}$ direct-abstract-declarator

direct-abstract-declarator:
    (abstract-declarator)
    direct-abstract-declarator$_{opt}$ [constant-expression$_{opt}$]
    direct-abstract-declarator$_{opt}$ (parameter-type-list$_{opt}$)

initializer:
    assignment-expression
    { initializer-list }
    { initializer-list , }

initializer-list:
    initializer
    initializer-list , initializer

B.2.3 Statements

statement:
    labeled-statement
    compound-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement

labeled-statement:
    identifier : statement
    `case` constant-expression : statement
    `default` : statement

compound-statement:
    { declaration-list$_{opt}$ statement-list$_{opt}$ }

declaration-list:
    declaration
    declaration-list declaration

statement-list:
    statement
    statement-list statement

expression-statement:
    expression$_{opt}$;

selection-statement:
    `if` (expression) statement
    `if` (expression) statement `else` statement
    `switch` (expression) statement

iteration-statement:
    `while` (expression) statement
    `do` statement `while` (expression) ;
    `for` (expression$_{opt}$; expression$_{opt}$; expression$_{opt}$) statement

jump-statement:
    `goto` *identifier* ;
    `continue` ;
    `break` ;
    `return` expression$_{opt}$;

B.2.4 External definitions

translation-unit:
    external-declaration
    translation-unit external-declaration

external-declaration:
    function-definition
    declaration

function-definition:
    declaration-specifiers$_{opt}$ declarator declaration-list$_{opt}$ compound-statement

Bibliography

[ANS89] ANSI. *ANSI C 3.159-1989 Standard - The C Programming Language.* American National Standards Institute, 1989.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[CN91] Brad J. Cox and Andrew J. Novobilski. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, MA, 1991.

[DH72] O-J. Dahl and C. A. R. Hoare. *Hierarchical Program Construction in Structured Programming.* Academic Press, New York, 1972.

[DMN70] O-J. Dahl, B. Myrhaug, and K. Nygaard. *SIMULA Common Base Language.* Norwegian Computing Center S-22, Oslo, Norway, 1970.

[EA98] UK Defence Evaluation and Research Agency. *The TenDRA Compiler.* URL: `http://alph.dra.hmg.gb/TenDRA`, 1998.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley, Reading, MA, 1995.

[FH98] Christopher W. Fraser and David R. Hanson. *The* `lcc` *Retargetable C Compiler.* URL: `http://www.cs.princeton.edu/software/lcc`, 1998.

[HJ95] S. P. Harbison and G. L. Steele Jr. *C: A Reference Manual.* Prentice Hall, Englewood Cliffs, NJ, fourth edition, 1995.

[I+79] Jean D. Ichbiah et al. *Rationale for the Design of the ADA Programming Language*, volume 14 No. 6. SIGPLAN Notices, 1979.

[ISO90] ISO. *ISO C 9899:1990 Standard - The C Programming Language.* International Standards Organization, Geneva, Switzerland, 1990.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, 1978.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[L$^+$79] Barbara Liskov et al. *Clu Reference Manual.* MIT/LCS/TR-225, MIT Cambridge, MA, 1979.

[LMB95] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly & Associates, Sebastopol, CA, second edition, 1995.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[Pla92] P. J. Plauger. *The Standard C Library.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[Pla96a] P. J. Plauger. *The Standard C++ Language.* Prentice Hall, Englewood Cliffs, NJ, 1996.

[Pla96b] P. J. Plauger. *The Standard C++ Library.* Prentice Hall, Englewood Cliffs, NJ, 1996.

[Sal94] Peter H. Salus. *A Quarter Century of UNIX.* Addison-Wesley, Reading, MA, 1994.

[SF$^+$98] Richard M. Stallman, Free Software Foundation, et al. *The GNU CC Compiler.* URL: `http://www.gnu.org/software/gcc`, 1998.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, Reading, MA, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, third edition, 1997.

[WB74] P. M. Woodward and S. G. Bond. *Algol 68-R Users Guide.* Her Majesty's Stationery Office, London, England, 1974.

[Wik87] Åke Wikström. *Functional Programming Using ML.* Prentice Hall, Englewood Cliffs, NJ, 1987.

Index