Appendix A Language Summary

A.1 Introduction

In this appendix we review the basic features of the language. The standard libraries are covered in Appendix B.

A.2 Comments

Comments are introduced with /* and terminated with */. Comments cannot be nested and they cannot occur within string literals.

A.3 Identifiers

Identifiers are names for objects such as keywords, functions, variables and constants. An identifier must

- begin with a letter
- contain only letters and digits and underscores

An upper case letter is not the same as its lower case version. Usually, identifiers may be any length but the first thirty one characters are significant. However the operating system may impose further restrictions on external identifiers such as those used for disk file names.

A.4 Keywords

The following identifiers are reserved for keywords.

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typdef
char        exterm      return      union

const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
defined     if          static      while
do
```

These keywords must be used according to their definition; they cannot be used in any other way.

A.5 Constants

An integer constant is a sequence of digits. An integer constant may be suffixed with *U* (for *unsigned int*), *L* (for *long int*) or UL (for *unsigned long int*). The letters *u* and *l* serve the

same purpose. An integer constant without a suffix is an ordinary *int* value. Examples of integer constants are

```
76, 0U, -47309L, 234567UL
```

A character constant is one (or two) characters enclosed within single quotes. For example, `'a'`, `'Z'`. A single character is translated into its numeric equivalent at run-time. Character constants containing two characters include

`\n`	newline	`\\`	the backslash itself
`\b`	backspace	`\?`	the question mark
`\r`	return	`\'`	the single quote
`\a`	alert or bell	`\"`	the double quote

So, even though *'\n'* looks like two characters, it represents just one - the newline character.

A float constant is a number with a decimal point; it has the suffix *F* (or *f*) for *float* or *L* (or *l*) for *long double*. A float value without a suffix is a *double*.

```
3.1416F,   193457.320234L, 23.007
```

The identifiers in an enumeration represent constant values of type *int*.

```
enum { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
```

Here, *Sun* has the value *0*, *Mon* the value *1*, *Tue 2*, and so on.

A.6 String Literals

A string literal (also known as a string constant) is a sequence of characters enclosed by double quotes. The last character, automatically placed there by the compiler, is the end-of-string character, *\0*. A string literal has type *array of char* and storage class *static*.

```
char message[] = "Program halted - unexpected error.";
```

Adjacent string literals are concatenated into a single string literal. For example the six string literals listed here

```
char menu[] = "Main menu.\n";
              "1  New\n";
              "2  Open\n";
              "3  Close\n";
              "9  Quit\n";
              "Choice? ";
```

are automatically joined into one long one, complete with the terminating NULL character, and stored in menu.

A.7 Storage Class

Storage classes include *auto* and *static*. A storage class describes how an object is to be

stored in memory.

Variables of storage class *auto* are created every time the block in which they are declared is entered, and are destroyed every time the block is exited. This is the default storage class.

Variables of storage class *static* are created the first time the block in which they are defined is entered and remain in existence until program execution terminates.

```
void function(void)
{
  auto int number;
  /* auto is not explicitly required.
     Creates a new variable on entry to the function.
     Destroyed on exit from the function. */
  static int state
  /* Persists between function calls. */
  ...
}
```

A.8 Basic Types

The fundamental types include

`char`	a character
`int`	an integer
`float`	a single precision floating point number
`double`	a double precision floating point number
`short`	`short int`
`long`	`long int`
`unsigned`	`unsigned int`
`unsigned long`	`unsigned long int`
`void`	the empty type has no values

A.9 The Usual Arithmetic Conversions

The usual arithmetic conversions ensure that both operands are of the same type before an arithmetic process takes place. In general, if two operands have different types then the operand with the 'lower' type is promoted to the same type as the other operand. The following table lists the types in order.

`long double`	highest type
`double`	
`float`	
`unsigned long`	
`long`	
`unsigned`	
`int`	
`short`	
`char`	lowest type

Conversions between signed and unsigned values depend on the implementation. For

example, suppose one operand is *long* and the other is *unsigned*: if the *long* type can represent all the *unsigned int* values, then the *unsigned* operand is converted to *long*, otherwise, both are converted to *unsigned long*. The result of a calculation which involves both *signed* and *unsigned* values should be carefully checked.

A.10 Promotion

On assignment, a value is automatically promoted to a higher type if necessary. The hierarchy of types is shown is section A.9 above. It is perhaps best to explicitly promote values with a cast operator. For example

```
(double)intValue;
```

converts the contents of *intValue* into one of type *double*. Demoting a value to one of a lower type via casts should be done with care.

A pointer may be converted (via a cast) to *void* * and back again without loss of information.

A.11 The Operators

An operator specifies a process to be applied to one or two operands. The C operators include

`!`	not
`!=`	not equal
`#`	substitute pre-processor token
`%`	modulus ie remainder after dividing one number by another
`&&`	logical and
`()`	cast operator
`*`	multiply two numbers
`+`	add two numbers
`++`	increment, add 1 to a number
`-`	subtract one number from another
`--`	decrement, subtract one from a number
`->`	select a member of (a pointer to) a structure
`.`	select a member of a structure
`/`	divide one number by another
`<`	less than
`<=`	less than or equal to
`=`	assignment
`==`	equality
`>`	greater than
`>=`	greater than or equal to
`[]`	array subscript
`defined`	check if a macro has been defined
`sizof`	size of operand in bytes
`||`	logical or

Precedence is the default order in which the operators in an expression are evaluated.

Precedence is always overridden by (). The precedence of the operators is shown in the following table; operators higher up in the table have a higher precedence than those lower down.

Operator							Associativity
()	[]	->	.				left to right
!	++	--	(type)	*	&	sizeof	right to left
*	/	%					left to right
+	-						left to right
<	<=	>	>=				left to right
==	!=						left to right
&&							left to right
||							left to right
=							right to left
,							left to right

Within a given level of precedence, associativity describes the direction of evaluation.

In the expression

```
while (c = 'a' != EOF)
```

!= has higher precedence than = and so the test *!=* is done before the assignment and *c* is assigned either true (1) or false (0) depending on whether *'a'* is not equal to *EOF*.

In

```
while ((c = 'a') != EOF)
```

the *()* has the highest precedence of all and so *c* is assigned the character *'a'* before it is tested for inequality with *EOF*.

A.12 Function Calls

A function call is an expression which involves a function name together with a pair of braces. The braces may either contain nothing or a list of expressions known as arguments. A copy of each argument value is passed, by the function call, to a corresponding function parameter.

returnedValue = function(argument-1, argument-2, ... argument-n);

A function may change its parameter values without any affect whatsoever on any argument variables. However, a function can indirectly change the contents of a variable if it is handed a pointer to the variable.

Both the arguments in a function call and the parameters in a parameter list must match (usually) one for one the parameters declared in the function prototype.

The order in which a functions parameter expressions are evaluated is not defined. So, for

example, if two functions are used as arguments, then there is no knowing which function will be executed first.

returnedValue = functionName(function-1(argList-1), function-2(argList-2))

The value returned by a function may either be ignored or used.

A.13 Increment and Decrement Operators

The increment-by-one operator is ++ and the decrement-by-one operator is --. They can either precede (prefix) or follow (postfix) an integer variable.
++i and *i++* both have the same affect - to increase the value stored in *i* by one. But, in the prefix version, *i* is incremented before it is used in an expression and, in the postfix version, *i* is incremented after it is used. For example, if numbers is an array of *int* and

```
int i = 2;

printf("%d", numbers[++i]);   /* increment i then print. */
printf("%d", numbers[i++];    /* print then increment i. */
```

A.14 The Address Operator

The address operator, &, returns the address of its operand. The operand must be a variable (or a function). The result is a pointer to the variable (or function).

```
int number = 2;
int *pointerToNumber = &number;
/* address of number is assigned to pointerToNumber. */
```

A.15 The Indirection Operator

The indirection operator, *, returns the value stored in the variable pointed to by its operand.

```
int number = 2;
int *pointerToNumber = &number;
/* pointerToNumber contains the address of number. */
*pointerToNumber == 2;
```

The indirection operator may also return the function to which its operand points.

A.16 The Array Subscript Operator

An array type is a sequence of adjacent indexed storage locations all of which hold values of the same type. For example

```
char letters[12] = "Catastrophe";
```

defines an array named *letters* of size 12 elements, indexed from zero up to 11, each element of which contains a single value of type *char*.

C	a	t	a	s	t	r	o	p	h	e	\0
0	1	2	3	4	5	6	7	8	9	10	11

Arrays are always indexed from zero upwards. Each element is referenced by its index. For example,

```
letters[4] == 's';
```

Each element of an array can itself be an array. For example

```
char words[5][12];
```

defines a two-dimensional array named words which can up to store five lines each up to twelve characters long.

words

0	C	a	t	a	s	t	r	o	p	h	e	\0
1	t	h	e		c	a	t	\0				
2	i	s		a	s	l	e	e	p	\0		
3	o	n		t	h	e						
4	m	a	t	.	\0							
	0	1	2	3	4	5	6	7	8	9	10	11

To reference a particular element, the row index is given first. For example

```
words[2][5] == 'l'
```

A.17 The Logical Not Operator

The logical not operator is denoted by *!*. Its operand must either be an arithmetic type or a pointer. It returns one (true) if if its operand is equal to zero (false), otherwise it return zero.

```
int n = 5;
!(n == 2) == 0;  /* false */
```

A.18 The sizeof Operator

The *sizeof* operator returns the number of bytes required to store a value of its operand type.

```
sizeof(char) == 1
```

sizeof(arrayType) == number of bytes in the array

sizeof(structureType) == number of bytes in the structure

A.18 Casts

A cast causes a temporary change in the type of an expression to the type enclosed within brackets. For example

```
int intValue;
...
(double)intValue;
```

converts *intValue* to a value of type *double*.

A.19 The Arithmetic Operators

The arithmetic operators are

`*`	multiplication	both operands must be arithmetic types eg *int*, *double*
`/`	quotient	both operands must be arithmetic types. The second operand must not be zero
`%`	remainder	both operands must be integer types only. The second operand must not be zero. The result is not defined if either operand is negative.
`+`	addition	
`-`	subtraction	

A.20 The Relational Operators

The relational operators are

`<`	less than
`<=`	less than or equal to
`==`	equal to
`!=`	not equal to
`>=`	greater than or equal to
`>`	greater than

The expression operand-1 relational-operator operand-2 returns either one (true) or zero (false).

```
(5 > 3) == 1;  /* true */
(5 < 3) == 0;  /* false */
```

A.21 The Logical Operators

The logical operators are && (logical and) and || (logical or).

`(A && B)`	returns one (true) if both A and B are one (true) otherwise returns zero (false). Evaluation is always from left to right and halts as soon as the result is known.
`(A || B)`	returns one (true) if either one of A or B is zero (false), otherwise returns zero (false). Evaluation is always from left to right.

For example,

```
(5 > 3) && (5 < 3) == 0;  /* false */
(5 > 3) && (3 > 1) == 1;  /* true  */
(5 > 3) || (5 < 3) == 1;  /* true. */
```

```
(5 < 3) || (3 < 1) == 0;  /* false */
```

A.22 The Assignment Operators

The assignment operators include

```
=    *=   /=   %=   +=   -=
```

The first (left) operand must be a modifiable item such as a variable or a structure. It must not be a const or an array or a function.

An expression such as

```
m *= n;
```

is equivalent to

```
m = m * n;
```

A.23 The Comma Operator

A pair of expressions separated by a comma are evaluated from left to right. For example,

```
int n = 2, n++;
```

leaves *n* containing *3*.

A.24 Structure Type Declarations

A structure consists of a collection of named objects called members. Each member has an identifier and a type. A structure declaration begins with the keyword *struct*. The declaration

```
typedef struct {
  char name[30];
  char payrollNumber[10];
  double annualSalary;
} Employee;
```

defines the *struct* type *Employee*. Variables of this type, as well as pointers to it, can be defined.

```
Employee employee;
Employee *pointerToEmployee;
```

The . member selector operator is used to refer to the contents of a particular member within a structure.

```
employee.salary.
```

If a pointer to a structure is involved, the offset operator -> is used to access a particular member.

```
pointerToEmployee->salary;
```

A.25 Enumerations

An enumeration is a list of identifiers which represent constants of type *int*. The values 0, 1, 2, ... are automatically assigned to each identifier as written from left to right. However, if an identifier is assigned an explicit value, then subsequent identifiers are automatically assigned values in sequence from that value onwards.

```
enum { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
```

assigns *0* to *Sun*, *1* to *Mon*, *2* to *Tue*, ...

```
enum { lastIndex = 5, arraySize }
```

assigns *5* to *lastIndex* and *6* to *arraySize*.

A.26 Declarations

Declarations have the format type identifier. Examples of declarations include

```
int number;      /* integer number */
int numbers[];   /* array of integers */
int *number;     /* pointer to integer */
int *numbers[];  /* array of pointers to integers */
const int n = 5; /* integer constant - value cannot be changed
                    during program execution. */
```

A.27 Function Declarations

A function declaration announces a function's signature, that is, its return type, name and the number and type of its parameters. For example

```
int strcpy(char *destination, const char *source);
```

declares a function named *strcpy*. *strcpy* returns an *int* and has two parameters. The first parameter, named *destination*, is a pointer to *char*. The second parameter is a pointer to constant *char*; this means that the sequence of characters pointed to by *source* cannot be changed by the function.

A.28 Initialisation

Variables may be assigned their initial values at their point of declaration. Examples are

```
int number = 0;
unsigned long number = 0UL;
int *pointerToNumber = &number;
```

```
typedef struct {
  char name[30];
  char payrollNumber[10];
  double annualSalary;
} Employee;

Employee nullEmployee = { " ", " ", 0.00 };  /* each member is
filled */

int days[] = { 0,31,28,31,30,31,30,31,31,30,31,30,31 };
/* size of array automatically computed from the number of
   initialisers. */

int numbers[5] = { 0, 0, 0, 0, 0 };
/* 0 is stored in each element */

char fatalErrorMessage[] = "Program halted - unexpected
problem\n";
/* one character is stored in each successive location. */
```

A two-dimensional array might be initialised like this.

```
int studentsInYear[2][5] = { { 25, 17, 30, 25, 25 },
                             { 90, 91, 92, 93, 94 } }
```

To initialise a pointer to a structure, storage must first be allocated with *malloc*.

```
typedef  struct {
  char name[30];
  char payrollNumber[10];
  double annualSalary;
} Employee;

Employee *pointerToEmployee = (Employee
*)malloc(sizeof(Employee));
```

A pointer MUST be initialised with the address of an appropriate variable before it is dereferenced, that is, before an attempt is made to retrieve from, or write to, the location whose address is stored in the pointer, otherwise garbage may be retrieved or the contents of an arbitrary location in memory may be overwritten (disastrous if that location contains a part of the operating system).

A.29 Typedef

typedef provides a new name for an existing type. For example

```
typedef enum { false, true } Boolean;
```

Here, the *enum { false, true }* is named *Boolean*. Hence we can declare Boolean variables thus

```
Boolean valueIsValid;
```

A.30 Statements

Statements include assignments and function calls.

A compound statement comprises a sequence of statements enclosed within braces *{* and *}*. A compound statement may be used wherever a statement would be used.

A.31 Selections

A selection statement transfers control. There are several forms.

if (expression)
statement

if (expression)
statement-1
else
statement-2

switch(expression)
statement

For example

```
if (cannotOpenFile) {
  printf("Program halted - cannot open the file.\n");
  exit(EXIT_FAILURE);
}
```

The *printf* and *exit* statements are executed only if *cannotOpenFile* is true, that is, only if *cannotOpenFile == 1*.

```
if (isFinished)
  fclose(file);
else
  fread(&record, sizeof(RecordStructure), 1 file);
```

The *fclose* statement is executed only if *isFinished* is true. However, if *isFinished* is not true, then *fread* is executed. Either *fclose* or *fread* is selected for execution.

The *switch* statement causes transfer of control to a case-labelled statement depending on value of expression. Only one default label is allowed per *switch* statement. *switch* statements may be nested.

```
switch (day) {
case 1:
case 31: printf("st");
  break;
case 2: printf("nd");
  break;
case 3: printf("rd");
  break;
default: printf("th");
  break;
}
```

When the *switc*h statement is executed, *day* is evaluated. If its value is *1* or *31*, *printf("st")* is selected for execution; if its value is *2*, *printf("nd")* is selected; if its value is *3*, *printf("rd")* is selected. But if the value of day is neither *1*, *31*, *2* nor *3*, the *default* case is executed and *printf("th")* is selected.

Once a case has been selected for execution, the following cases are also executed - unless control is transferred by a statement such as *break*, *return* or *exit*.

A.32 Iteration

An iteration statement controls the number of times a statement is executed. One form is

while (expression)
  statement

For example

```
while (!feof(file)) {
  printRecord(record);
  fread(&record, sizeof(RecordStructure), 1, file);
}
```

For as long as *feof(file)* is not true, execute the *printRecord* and *fread* statements. It is possible that no iterations are made at all: if *feof(file)* is initially true, then neither the *printRecord* nor the *fread* statement is executed.

Another form is

for (expression-1; expression-2; expression-3)
  statement

expression-1, 2 and 3 are optional.

```
for (i = 0; i <= lastIndex; i++)
  printf("%d\n", intArray[i]);
```

Initially, *i* is zero. Then for as long as *i* remains less than or equal to *lastIndex*, *printf* is executed and *i* is incremented.

A.33 Unconditional Transfer of control

The unconditional transfer of control statements include *break* and *return*.

break may appear only in an iteration or a switch - it terminates execution of the smallest enclosing statement.

return causes an immediate exit from a function, returning control to its caller. When *return* is followed by expression, the value is returned to the caller. Falling off the end of a function is equivalent to returning without a value.

A.34 Function Definitions

A function definition is made up of a function declarator (ie function heading) together with a function body. A function declarator must match its prototype declaration. An array passed as a value to a function parameter is automatically converted to a pointer.

A function body usually comprises variable definitions together with a sequence of statements all enclosed within braces *{* and *}*

```
int strcpy(char *destination, const char *source)
{
  int n = 0;

  while ((*destination = *source) != '\0') {
    destination++;
    source++;
    n++;
  }
  return n;
}
```

A.35 Scope

The scope of an identifer is that part of a program in which the named object may be used.

```
/* program 11.1 - demonstrates scope. */
#include <stdio.h>

void function1(int prototypeParameter);

int globalVariable = 10;

int main()
{
  int localVariable = 2;
  function1(localVariable);
}

void function1(int parameterVariable)
{
  int localVariable;  /* no conflict with localVariable in
                         main. */
```

```
  for (localVariable = 0; localVariable < 5; localVariable++)
{
    int blockVariable = localVariable * parameterVariable;
    printf("%d ", blockVariable);
  }
  printf("%d\n", globalVariable);
}
```

When run, the program prints

```
0 2 4 6 8 10
```

on the screen.

The scope of the prototype parameter is just the prototype itself; its purpose is for documentation only.

A global variable may be by any function used from where it is first declared onwards. If a local variable has the same name as a global variable, then it *masks* the global variable.

A local variable (including parameter variables) may be used only by the function in which it is declared.

A block variable may be use only within the block (delimited by { and }) in which it is declared. Again, a block variable "hides" a function variable with the same name.

Identical identifiers may be used without conflict providing their scopes are disjoint (that is, separate).

A.36 The Pre-Processor

The pre-processor performs tasks such as text substitution and file inclusion before translation takes place from C text (that is, C source code) into C object code. Instructions to the pre-processor, known as pre-processor directives, begin with #. Each directive must be written on a line of its own.

```
#define maxSize 100
```

replaces all occurrences of *maxSize* in a program file with *100* - except in comments and in quoted string literals.

```
#include <stdio.h>
#include "header.h"
```

includes the entire contents of stdio.h and header.h in the program file before compilation takes place. The angle brackets usually indicates that the file to be included is in the C system directory. The quotation marks usually indicates that the file to be included is in the same directory as the program text file.

It would be an error to include the same text more than one in the same program file. So, header files should contain something like

```
#if !defined headerFileIncluded
  #define headerFileIncluded
     /* contents of header file go here. */
#endif
```

On the first inclusion of the header file, *headerFileIncluded* is not defined. So, it is defined and the text up to the next *#endif* is included. On any subsequent inclusion, *headerFileIncluded* has already been defined and so the text up to the next *#endif* is not included.