Knowledge Software Ltd
Farnborough, Hants, England

Support

Knowledge Software Ltd provides telephone and mail support for those users who have purchased their systems from Knowledge Software Ltd. All other users of this system must contact their supplier for support. Knowledge Software Ltd does not have the resources to support users who have purchased their software from other vendors.

Disclaimer

This document and the software it describes are subject to change without notice. No warranty, express or implied, covers their use. Neither the manufacturer nor the seller is responsible or liable for any consequences of their use.

TradeMarks

CtoP and POP-NCG are trademarks of Knowledge Software Ltd.

POPTYSER and EDIP are registered trademarks of Knowledge Software Ltd.

UCSD, UCSD Pascal and UCSD p-System are trademarks of the Regents of the University of California.

Knowledge Software Ltd, 62 Fernhill Road, Farnborough, Hants GU14 9RZ.

Tel: +(44) 01252-520667

e-mail: ctop@knosof.co.uk

Chapter 1

Introduction to CtoP

It is theoretically possible to make a 100% conversion from ‘C’ to Pascal. However, after such a conversion the Pascal would be as unreadable as the original ‘C’.

Our aim with CtoP has been to produce output which is understandable. This has meant that some source conversion is not performed. Instead the offending source is flagged and the programmer given suggestions about possible conversion strategies.

CtoP makes three passes over the source code. The first pass gathers information necessary for generating the **INTERFACE** sections in the converted program. The second pass analyses the declarations and uses of variables within each source file and the converted program and cross reference information are output on the third pass.

1.1 Using CtoP

The best results are obtained by giving CtoP all source code to a program. It is possible to give complete source source code as input and obtain selected source code as output.

To be expanded…

1.2 Reading this manual

The reader is assumed to know Pascal. Sufficient explanations of ‘C’ are given in order for the programmer to gain some understanding of the problems presented in performing a conversion.

Most sections have the format:

What ‘C’ does.	What Pascal does.

What CtoP does.

What a programmer may have to do.

Information provided by CtoP.

1.3 Which C?

CtoP uses Unix System V ‘C’ on the 68000 as its reference point. The Kernighan & Ritchie ‘C’ is a small language in comparison with the above. The draft ANSI standard is currently updated every 3 months.

1.4 An overview of C programming

‘C’ was designed for people who get soup in their beards.

‘C’ encourages trickery. A lot of experience is required to become familar with these tricks.

Chapter 2

Lexical Elements

There are differences between lexical elements in the two languages. For the most part these differences are handled as part of the construct in which they occur. The one exception is constants. Here ‘C’ has characters that have ’magical’ properties; these are dealt with on page 7.

We will go through the differences here in order to provide background information.

2.1 Layout of the source code

Both languages are relatively free of restrictions regarding the layout of source code. CtoP attempts to provide lines with an indentation scheme applicable to Pascal. Except where multiple ‘C’ statements appear on one line, CtoP will map input lines to output lines.

In ‘C’ a ’\’ as the last character on a line signifies that the following line is to be appended. This feature is used in defining macro bodies and strings. CtoP correctly joins the lines and removes the ’\’.

The preprocessor has its own layout conventions. No attempt will be made to follow unpreprocessed layout.

2.2 The source character set

The ASCII character set is assumed. See appendix 1.

2.3 Comments

/* */	(* *)

Comments have the status of the space character in both languages.

Comments not enclosed within function bodies will be output as they are encountered.

Comments within declarations will occur at the head of the surrounding scope in which they occur.

Comments within statements will occur as close as possible in the output to their original position in the input.

2.4 Tokens

'C' attempts to form the longest possible token. The implications of this rule only involve tokens that are not in Pascal. CtoP handles these automatically.

2.5 Operators and separators

%	**MOD**
& &&	**AND**
| ||	**OR**
^ ~	**NOT**
/	**DIV**
!=	**<>**
==	**=**
->	**.** or **^.**
=	**:=**

'*;*' is treated as part of the syntax of statements.

'**;**' is treated as a separator.

2.6 Identifiers

Case is significant	Case is not significant
'_' is significant	'_' is not significant
'_' may occur as the first character	'_' may not occur as the first character
'*$*' is usually a legal character and is significant	'**$**' is not a legal character in a name
More than 8 characters are usually considered significant. The proposed Ansi standard specifies a limit of 31 significant characters	Only the first 8 characters are considered significant.
Some linkers impose a limit on the number of significant characters in *extern* names. The porposed Ansi standard specifies 7.	Some linkers impose a limit on the number of significant characters in **EXTERN** names. A limit of 6 or 8 is common.

Letters within names are output as they appear in the input.

The '*$*' character is output as a '**D**'.

Names having a '_' as the first character have a '**U**' added to the front of the name.

If names in the same scope clash the second name is made unique using the following rules:

Digits are inserted into the name in the least significant positions to make the Pascal identifiers unique. e.g.,
if 'C' defines the names *ALongIdentifier a_long_identifier A_LONG_IDENTIFIER*, they are distinct names (in 'C'). CtoP will produce the following (Pascal) identifiers:
ALongIdentifier a_long_id1entifier A_LONG_ID2ENTIFER.

2.6.1 #define names

An attempt will be made to carry constants defined as macro names over to Pascal as constant identifiers. See **§3** for further details.

2.6.2 Labels

Labels are represented by names following the same rules as those for identifiers.	Labels are represented by positive constant integers.

Unique numbers are allocated for every label identifier in a function body. The Pascal label **999** is reserved to follow the last executable statement in the function body.

2.7 Reserved words

'C' contains reserved words that are not in Pascal. This simply prevents programmers using these names as identifiers in 'C'.

Pascal has its own set of reserved words and also has the concept of predefined words. 'C' identifiers that clash with Pascal reserved words will be made unique. 'C' identifiers that clash with predefined words will be made unique if the 'C' names do not perform the same function.

‘C’ reserved words:

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

(UCSD) Pascal reserved words:

AND ARRAY BEGIN CASE CONST DIV DO DOWNTO ELSE END EXTERNAL FILE FOR FORWARD FUNCTION GOTO IF IMPLEMENTATION IN INTERFACE LABEL MOD NOT OF OR PACKED PROCEDURE PROCESS PROGRAM RECORD SEGMENT SEPARATE SET THEN TO TYPE UNIT UNTIL USES VAR WHILE WITH

(UCSD) Pascal predefined names:

abs arctan atan attach blockread blockwrite boolean char chr close concat copy cos crunch delete dispose eof eoln exit exp false fillchar get gotoxy halt idsearch input insert integer interactive ioresult keyboard length ln log lock mark maxint memavail memlock memswap moveleft moveright new nil normal odd ord output pack page pmachine pos pred processid purge put pwroften read readln real release reset rewrite round scan seek semaphore seminit signal sin sizeof sqr sqrt start str string succ text time treesearch true trunc unitbusy unitclear unitread unitstatus unitwait unitwrite unpack varavail varnew wait write writeln

2.8 Constants

2.8.1 Integer constants

Octal and hexadecimal digits may be represented.

There is no representation for octal and hexadecimal digits.

Long integer constants may be represented.

Long integer constants may be represented (UCSD Pascal extension).

Octal and hexadecimal constants will be converted to their decimal equivalent.

See page 25 for a discussion of long integers.

2.8.2 Floating-point constants

The type of a floating-point constant is always *double* . The size of real constants is the same as the size of real variables. For a discussion of real size see page 25

Fixed point and exponential notation may be used to represent floating-point constants in both languages.

Examples

4.3 7E-4 -23.7504

2.8.3 Character constants

Non-printing characters may be represented.	Non-printing characters may not be represented.

The ‘C’ convention of representing non-printing characters will be carried over into the output. See below for a discussion of escape characters.

2.8.4 String constants

Strings are delimited by the ’"’ character.	Strings are delimited by the ’’’ character.
The string delimiter is represented in a string by \"	The string delimiter is represented in a string by ’’
Strings may contain characters that are interpreted to have special meaning by the runtime support routines.	There are no conventions specified for interpreting any characters as having special meanings.
The type of a string constant is "array of *char*".	The type of a string constant is **PACKED ARRAY OF CHAR**.
The value of *sizeof("abc")* is 4.	**SIZEOF(’abc’)** is illegal.

CtoP will change the delimiting character, convert \" to ", \’ to ’’ and copy the string unchanged to the output file.

2.8.5 Escape characters

These are sequences of characters used to denote other characters that cannot be easily represented in a source program.

```
escape-character ::= ’\’ escape-code

escape-code      ::= character-escape-code | numeric-escape-code

character-escape-code ::= ’b’ | ’f’ | ’n’ | ’r’ | ’t’ | ’v’ | ’’’ | ’"’
```

```
numeric-escape-code   ::= octal-digit { octal-digit { octal-digit }?}?
```

2.8.6 Character escape codes

The meaning of the character escape codes are as follows:

b	backspace
f	form feed
n	newline
r	carriage return
t	horizontal tabulate
v	vertical tabulate
’	single quote
"	double quote

2.8.7 Numeric escape codes

Any character may represented by writing that character as its octal encoding.	**CHR** converts decimal integers into char type.

In ‘C’ these escape codes are usually assigned to *#define* names (**CONST** identifiers in Pascal). Pascal does not allow the use of **CHR** in assignments to **CONST** identifiers. CtoP outputs the **CHR** form and leaves the programmer to make any changes in **CONST** identifiers.

Escape codes may also appear in ‘C’ *case* labels. If the selector value has type *char* CtoP outputs the **CHR** form.

Note that the handling of escape codes differs in strings and characters.

Chapter 3

The C preprocessor

'C' compilers come with a preprocessor. Pascal does not have any macro facilities. The input, possible containing macros, is expanded into 'C' and this 'C' is then converted.

The two most common uses of macros in 'C' are:

1. To give names to constants. CtoP attempts to carry these over as constant identifiers.
2. To give a name to a commonly used piece of code. There are two reasons for wanting to do this:

    a) The overhead of placing the code in a function and calling it is considered too great.

    b) The 'C' syntax is extended in some way.

    CtoP expands out the macro body and converts the 'C' to Pascal.

For those users wanting a greater understanding of how macros are expanded CtoP has a debug mode which prints out information on macro expansion as it happens.

Those books purporting to describe 'standard' 'C' all point out that the preprocessor works at the token, not the character level. Some preprocessors work at the character level. CtoP works at the token level. The impact of this deviation by some compilers will only become apparent if the original author had used this feature in devious ways.

Chapter 4

Declarations

4.1 Organization of Declarations

The declaration of variables, functions and types may be mixed in any order.	Declarations must follow a strict order: labels, constants, types, variables, functions (and procedures) and main body.
Declarations may occur at the top level or at the head of any block (compound statement).	Declarations are global, or local to a function (or procedure).
Variables and functions may be refered to before their defining point.	Variables and functions must be defined before they can be referenced.

CtoP lifts head-of-block declarations out to the local level and if necessary, modifies the identifiers to avoid any resulting name clashes.

CtoP does not reorder function declarations to place defining points before uses.

4.2 Terminology

4.2.1 Scope

A top-level declaration is in scope from its declaration to the end of the source program file.	Equivalent to a global declaration in Pascal.
A formal parameter is in scope from its declaration to the end of the function body.	Similarly for formal parameters and locals.

A head-of-block declaration is in scope from its declaration to the end of the block.	Declarations inside blocks are not allowed.
… *{* … *for (…)* *{* *int i;* *…i…* *};* … *}*	**VAR** **i : INTEGER;** … **BEGIN** … **FOR** … **DO** **BEGIN** …**i**… **END;** … **END**
A statement label is in scope throughout the function body in which it appears.	Same as ‘C’.

4.2.2 Visibility

Identifiers are introduced into the name space of the scope in which they occur in the textual order they are encountered.	Identifiers are introduced simultaneously into the name space of the scope in which they occur.
int j; *{* *int i=j;* *int j=0;* … *}*	**VAR** **j : INTEGER;** **i : INTEGER;** **j1: INTEGER;** **BEGIN** **i := j;** **j1 := 0;** … **END**

4.2.3 Forward References

Use of a label is permitted before its declaration point (i.e., the statement which it labels).	All labels must be declared before use.
A structure, union or enumeration tag may be used before it is declared when the size of the structure etc is not needed.	It is possible to declare a pointer to a type before that type is declared, but otherwise declarations must precede use.
struct list { *struct list *next;* … *};*	**TYPE** **P_list = ^ list;** **list = RECORD** **next : P_list;** … **END;**

4.2.4 Overloading Of Names

'C' allows items to have the same name, if they refer to objects in different name spaces:

Statement labels.	Labels in Pascal are positive integers.
Structure, union and enumeration tags.	These correspond to type names in Pascal, but must be distinct from other identifiers.
Variables, functions, *typedef*-names, enumeration constants etc.	In Pascal, this name space includes type names too (structure etc. tags).

4.2.5 Duplicate Declarations

Any number of external declarations for the same name may exist (as long as their types agree).	Multiple declarations of the same name in the same scope are illegal (with the exception of **FORWARD**).
Duplicate declarations are a method of introducing variables and functions before their defining point.	Functions and procedures may be declared forward.

CtoP removes any duplicate declarations.

4.2.6 Duplicate Visibility

{	**VAR**
int i;	**i : INTEGER;**
…i…	**i1 : INTEGER;**
{	**BEGIN**
int i;	…**i**…
…i…	
};	…**i1**…
…i…	
}	…**i**…
	END

CtoP lifts the head-of-block declarations out to the level local to the enclosing function/procedure body. Name clashes are resolved by renaming the nested identifiers (see page 4).

4.2.7 Extent

The differences between 'C' and Pascal, if any, do not affect CtoP. See also pages 16 and 23.

4.2.8 Initial Values

Initialization may be specified for most variable declarations.	Initialisation on declarations is not allowed.

Initialization occurs when storage is allocated for that variable. CtoP arranges for local initializations to occur at the beginning of the function body.

Static variables keep their value, unchanged, during execution outside their scope. CtoP exports such variables to the global level, and their initializations occur at the beginning of the main program.

The mechanism used to convert 'C' initializers into Pascal keeps things legal.

CtoP initialises *static* variables to zero.

4.2.9 External Names

Declaring a name *external* is a method of providing information about variables and functions that is not available by any other means.	UCSD Pascal provides an explicit mechanism for importing variables and functions. A function declared as **EXTERNAL** is assumed to be in assembly language. An explicit link stage is required to incorporate these external assembly language routines.

4.3 Storage Class Specifiers

auto	Local variables in procedures.
extern	Global variables imported from other compilation units.
register	The first 16 locations in a procedure can be handled more efficiently than subsequent locations.
static	Global variables.
typedef	**TYPE** declarations.

Local variables declared as *static* will be made global by CtoP.

Local variables declared *register* will be reordered to the beginning of the local declarations to take advantage of the extra efficiency (most architectures have short form addressing that allow variables with small offsets to be access more efficiently than other variables). This reordering is safe because it is illegal to "point to" *register* variables in ‘C’.

4.3.1 Default Storage Class Specifiers

Global variables and functions are assumed to have storage class *extern*.	No variable, function or procedure may be exported unless explicitly stated, i.e., by placing it in an **INTERFACE** part.
For local variables *auto* is assumed.	Local variables are equivalent to *auto*.

4.4 Type Specifiers

enum { one, two,…}	**(one, two,**…**)**
float	**REAL**
int *char*	**INTEGER** **CHAR**
struct {…}	**RECORD** … **END**
Typedef-name.	User-defined type name.
union {…}	Variant record.
void as in "function returning *void*"	**PROCEDURE** … Procedures/functions are not available as variable types.

4.4.1 Default Type Specifiers

The type specifier may be omitted from a declaration, in which case it defaults to *int*.	A type specifier is required for all declarations.

CtoP generates **INTEGER** if no type specifier is given in the original 'C' source.

4.4.2 Missing Declarators

The declarator may be omitted.	The declarator may not be omitted.
A tagged type specifier (*struct*, *union* or *enum*), declares a new type.	This corresponds to a **TYPE** declaration.

4.5 Declarators

Declarations introduce the name being declared.

4.5.1 Simple Declarators

struct date { *int day, month, year;* *} today;*	**TYPE date = RECORD** **day : INTEGER;** **month : INTEGER;** **year : INTEGER** **END;**
int i; *float time;*	**VAR** **i : INTEGER;** **time : REAL;** **today : DATE;**

4.5.2 Pointer Declarators

*int *p;*	**p : ^INTEGER;**

Although this declaration declares *p* to be a pointer to an integer in 'C' it will frequently be assigned an *array* of *int*. For these assignments the address of the first element of the *array* is assigned to *p*. Arithmetic may be performed on *p* to move it along the *array*. See **§7.14** for further discussion.

4.5.3 Array Declarators

The lower bound is always zero.	The lower bound is user selectable.
The number of elements in the array need not be explicitly given in formal parameters.	Conformant array parameters need not have their explicit bounds provided.
The number of elements in the array need not be explicitly given if an initialiser is specified.	Initialisers are not available.
int a[10]; *int aa[10][20];* *char hello[]="Hello World";*	**VAR** **a : ARRAY [0..9] OF INTEGER;** **aa : ARRAY [0..9] OF** **ARRAY [0..19] OF INTEGER;** **hello : STRING[11];** … **BEGIN** **hello := CONCAT('Hello World', C_Null_String);** …

CtoP will work out the number of elements in an array if an initialiser is provided.

When the initializer is a string, CtoP generates **STRING** instead of **ARRAY**…**OF CHAR**, and outputs initialization code for a null-terminated string. **C_Null_String** is CtoP-generated variable in the interface of TypeUnit, initialized to the equivalent of 'C's null-string ″″. See page 66 for a discussion of the CtoP-generated **UNIT** containing type declarations.

4.5.4 Function Declarators

Variables of type "function returning …" may be declared.	Variables of type "function returning …" are not available.

4.5.5 Composition of Declarators

4.6 Initializers

When a variable is declared an initial value may be specified. This value is assigned to that variable at the start of that variables lifetime.	Variables may not be given an initial value at their defining point.
Static variables are defined to have an initial value of *0*, even if no initialiser is specified.	All variables are undefined at the start of program execution.

CtoP moves initialisation code to the start of the procedure or function. They are moved to the main program in the case of *static* variables.

'C' compilers usually impose restrictions on the forms of expressions that may occur as initialisers to different storage classes. Since CtoP converts initialisation code into assignments these restrictions are no longer applicable.

4.6.1 Integers

Such initializations are correctly translated by CtoP.

4.6.2 Floating-point

Such initializations are correctly translated by CtoP.

4.6.3 Pointers

Such initializations are correctly translated by CtoP.

4.6.4 Arrays

The 'C' text of the initializer is output as-is.

4.6.5 Enumerations

Such initializations are correctly translated by CtoP.

4.6.6 Structures

The 'C' text of the initializer is output as-is.

4.6.7 Unions

The 'C' text of the initializer is output as-is.

4.6.8 Other Types

Function types and *void* may not occur in initialisers

4.6.9 Eliding Braces

This is a ‘C’ specific feature handled by CtoP as part of the conversion process.

4.7 Implicit Declarations

It is permitted to call an external function that has not been declared. The declaration *extern int …();* is assumed.	All names must be declared before use.

CtoP generates the appropriate function declaration. See page 66 on **UNIT**s generated by CtoP.

4.8 External Names

‘C’ compilers adopt various strategies to determine which declaration, of a multiply declared identifier represents its definition.

The definition of a function is self-evident since it has a body. All multiple declarations of a function resolve onto this definition.

Variables are more of a problem. Some compilers assume that all *extern* declarations are simply references and a non-*extern* declaration of the identifier is its definition. Other compilers insist on the defining declaration being indicated by use of an initializer.

CtoP assumes that the non-*extern* declaration is the defining instance. This is reasonable because implicit declarations are always assumed to be *extern* in ‘C’.

The implications of this are only important if the ‘C’ source assumes that defining instances may be *extern*. In this case CtoP will flag the identifier as "not declared in the suite of ‘C’ source programs".

Chapter 5

Types

The type checking is only strong for aggregates.	The type checking is always strong.
Implicit type conversions may occur.	Implicit type conversions never occur.

Categories of types:

void	No equivalent
Function	Function or procedure variables may only occur as formal parameters
Scalar types	Scalar types
int, *char*, *	**INTEGER**, **CHAR**, **^**, **BOOLEAN**
Enumerated types	Enumerated types
Aggregate types	Aggregate types
array, *struct*, *union*	**ARRAY**, **RECORD**

5.1 Storage Units

The size of an object of type *char* is taken as 1 unit of storage.	UCSD Pascal takes 1 word of 16 bits to be its unit of storage.

To be expanded…

5.2 Integer Types

Integer types are used to represent booleans as well as the more usual integral numbers. Booleans are represented by the integers *0* for **FALSE** and *1* for **TRUE**.

Integers are 16 bit signed quantities. Unsigned integer fields of packed records may contain less than 16 bits.

5.2.1 Signed Integer Types

There are three sizes of signed integer, denoted by *short*, *int* and *long*. The only assumption made is that *int* is at least as large as *short*, and *long* is at least as large as *int*.

Pascal provides one size, **INTEGER**. UCSD has **INTEGER[n]** as an extension, where **n** is the number of digits required.

5.2.2 Unsigned Integer Types

An unsigned integer type represented by *n* bits will hold values in the range 0 to 2^n-1. Unsigned integers are available in the same three sizes as signed integers (see above), and the same assumption holds.

Unsigned integers in the range **0..MaxInt** may be declared using subranges.

All arithmetic involving unsigned integers delivers unsigned results. Comparisons with negative values will give misleading results, e.g., the comparison *(u > -1)*, where *u* is unsigned, will yield *0* (**FALSE**) because *-1* is converted to an unsigned value (i.e., 2^n-1) before the comparison.

All arithmetic on positive subranges is done using signed arithmetic.

5.2.3 Character Type

Characters are represented by their ordinal values. It is unspecified whether or not characters behave as signed or unsigned integers.

Characters are represented by their ordinal types.

Expressions of type char may be converted implicitly to *int*.

Expressions of type char may explicitly converted to **INTEGER** using the built in function **ORD**.

5.2.4 Long Integers

These are integers that can take on values in the range -2147483647…2147483647.	**INTEGER[32]** may contain upto 32 digits.
Usually held in twos complement form.	Not usually held in twos complement form (BCD is common).

Programs using *long int* to represent bit vectors longer than 16 bits are not going to find the UCSD **INTEGER[]** of much use.

5.3 Floating-Point Types

Single and double precision floating point numbers are provided by *float* and *double* (or *long float*) respectively. All values of type *float* are converted to *double* in expressions. As with integer types, the assumption is that the size and precision of *double* is at least as great as *float* although they may be implemented as the same size.	Pascal provides one floating-point size. A variable of type **REAL** has an implementation defined size that may correspond to the 'C' *float* or *double*.

5.4 Pointer Types

Pointers may point at the stack or heap.	Pointers may only address objects on the heap, although this can be circumscribed.
A special pointer value, *0*, is defined as the null pointer.	The null pointer is represented by the predefined word **NIL** (this has an implementation defined value).

CtoP will convert *0* to **NIL** where the context requires a pointer type.

5.4.1 Pointer Arithmetic

Pointers are very similar to arrays, and may be indexed as if they were arrays.	Pointers and arrays are completely different types.

5.4.2 Some Problems With Pointers

Because of the multifarious use of pointers there are sometimes problems with addressing on some machine architectures.	The operations on pointers are sufficiently restrictive to allow reasonable handling on most machines.

CtoP assumes that a programmer has not made use of a particular, machine specific, representation form for pointers.

5.5 Array Types

An array may be declared to have elements of any type except *void* and "function returning…". All arrays are restricted to one dimension and are zero-origin (i.e., their lower bound is always 0).	An array may be declared **VAR a : ARRAY [n..m] OF** …**;** where the lower bound, **n**, may be non-zero. (**n** and **m** must be constants)
int a[10];	**a : ARRAY[0..9] OF INTEGER;**
char c[1];	**c : ARRAY[0..0] OF CHAR;**

5.5.1 Arrays and Pointers

When an array name appears in a 'C' expression, it is converted to a pointer type that refers to the first element of the array. Indexing is thus defined in terms of pointers and indirection as follows: *a[i]* is defined as **(a + i)*.	Array names may not appear in an expression unless they are subscripted, except as an argument to a procedure, function or whole array assignment.

See also page 62 and **§7.14**.

5.5.2 Multidimensional Arrays

A multidimensional array is declared as "array of array of …", i.e., *…a[n][m]…;*	Multidimensional arrays are declared **VAR a : ARRAY [i..n, j..m] OF** …**;** **aa : ARRAY [i..n] OF** **ARRAY [j..m] OF** …**;**

5.5.3 Array Bounds

Subscripts are not normally checked to be within range.	Subscripts are checked, but this range checking may be switched off.
Array bounds may be omitted from a declaration when that array is single dimensioned and either a formal parameter to a function or an externally defined array.	The bounds may never be omitted from a declaration.

CtoP generates a conformant array declaration for array parameters whose bounds have been omitted. See page 63 for a discussion of how formal parameter declarations are handled.

5.5.4 Operations

'C' has a restricted set of operations that may be performed on pointers. CtoP handles the conversion.

5.6 Enumeration Types

Enumeration types associate an integer value with each enumeration constant.	Enumeration types are distinct types.
The integer values assigned to the enumeration identifiers implicitly start at zero and increase in increments of 1.	The integer values assigned to the enumeration identifiers implicitly start at zero and increase in increments of 1.
The integer value of enumeration identifiers may be overriden by an explicit assignment.	The internal values of enumeration identifiers may not be overriden.
The integer values assigned to each enumeration identifier need not be unique.	The internal values assigned to each enumeration identifier are unique.

5.6.1 Detailed Semantics

To be expanded…

5.7 Structure Types

struct name {…};	**TYPE name = RECORD** … **END;**
struct {…} name;	**VAR name : RECORD** … **END;**
struct sname {…} vname;	**TYPE sname = RECORD** … **END;** **VAR vname : sname;**

5.7.1 Operations on Structures

sname.cname	**sname.cname**
spname->cname	**spname^.cname** or **spname.cname** if **spname** has been identified as a **VAR** parameter, rather than a true pointer.
Structures may be returned by functions.	Structures may not be returned by functions.

See page 62 for more information on **VAR** parameters.

5.7.2 Components

The differences between ‘C’ and Pascal, if any, do not affect CtoP.

5.7.3 Structure Component Layout

There is an implied order of assigning variables to memory in ‘C’.

```
struct {
  int a, b, c;
  int d;
  int e;
}
```

It is assumed that *a* is followed by *b*, then *c*, *d* and finally *e*.

In Pascal, no assumptions are made. In practice, the mechanics of most implementations ensure that:

a) **RECORD a, b, c : INTEGER; END**

b) **RECORD a : INTEGER; b : INTEGER; c : INTEGER; END**

a) and b) have a different layout. b) is consistent with ‘C’, so fields should always be declared singly.

CtoP generates separate declarations for each name to guarantee the same order in the Pascal code:

int i, j, k;	**VAR** **i : INTEGER;** **j : INTEGER;** **k : INTEGER;**

5.7.4 Bit Fields

Packing of *struct*s is done by specifying field widths.	Packing of **RECORD**s is indicated by the keyword **PACKED**.
The width of an integer component may be specified in bits. This component should be *unsigned.*	Fields may be declared as a subrange $0..2^n-1$ where *n* is the number of bits in the field.
Some bit fields may be unnamed, to act as padding only.	All fields must be named so a dummy name must be generated for such fields.

5.7.5 Portability Problems

The use of bit fields in 'C' raises the question of packing and alignment strategies.

UCSD Pascal packs fields in the order in which they are defined. Fields may span byte boundaries. Fields may not span word boundaries.	

5.7.6 Sizes of Structures

The size of a structure includes any padding necessary for alignment considerations. e.g., *struct { double f; char c; }* may require padding to align a *double* following the *struct*. Any type may start on any 16 bit word boundary.	The size of a structure includes only internal padding for alignment.

It possible that structure sizes may differ between 'C' and Pascal.

5.8 Union Types

union {	**RECORD CASE INTEGER OF**
* int i;*	**0 : (i : INTEGER);**
* char c[2];*	**1 : (c : ARRAY[0..1] OF CHAR);**
}	**END**

For the equivalence of *union name {…}* and *union {…} name*, see **§5.7 Structure Types** above, as the same conversion rules apply.

5.8.1 Union Component Layout

union overlays single components, so *struct* is needed to group components together in each "variant".	Variant records allow components to be overlayed in memory.

struct overlay {	**TYPE**
* char tag;*	** S_IntChar = RECORD**
* union {*	** quantity : INTEGER;**
* struct {*	** part_id : PACKED ARRAY [0..1] OF CHAR;**
* int quantity;*	** END;**
* char part_id[2];*	** U_SIntChar = RECORD CASE INTEGER OF**
* } E;*	** 0 : (E : S_IntChar);**
* char buffer[4];*	** 1 : (buffer : PACKED ARRAY [0..3] OF CHAR);**
* } U;*	** END;**
};	** overlay = RECORD**
	** tag : CHAR;**
	** U : U_SIntChar;**
	** END;**

The programmer could then improve these declarations, provided that U and E are not accessed in the code, to:

```
TYPE
  overlay = RECORD
      tag : CHAR;
      CASE INTEGER OF
      0 : (quantity : INTEGER;
          part_id  : PACKED ARRAY[0..1] OF CHAR);
      1 : (buffer   : PACKED ARRAY[0..3] OF CHAR);
      END;
```

5.8.2 Sizes of Unions

The size of a union is the size of the largest component in that type. This rules applies for both languages.

5.9 Function Types

A function may return an object of any type except "array of …" or "function returning …".	A function may only return a scalar type.
A function returning *void* returns no value.	A function returning no value is declared **PROCEDURE**.

5.10 Void

void fn(…)	**PROCEDURE fn**(…)**;**

'C' programmers often omit *void*, but CtoP will attempt to spot that "function returning *void*" is intended (see **§9 Functions**).

5.11 Typedef Names

typedef is used to name a type: *typedef int *name;*	Types are named by declaring them: **TYPE name = ^INTEGER;**

5.11.1 Redefining Typedef Names

Typedef names follow the same scope and redefinition rules as variables.

5.11.2 Implementation Note

alpha (beta); is grammatically ambiguous in 'C'. If *alpha* is a *typedef*-name, then this is a declaration: *beta* is an *alpha.* Otherwise, *alpha* must be a function that takes *beta* as an argument. CtoP correctly detects and handles this case.

5.12 Type Equivalence

These are the rules that decide if two types are the same. The rules are important in the sense that operations between variables of two types may only be possible if these equivalence rules hold.

In both languages names declared as typedef names are synonymous for types.

The 'C' type equivalence rules are much more lax than Pascal. CtoP generates names for any equivalent types that are used more than once and do not yet have a name. Whenever CtoP encounters an anonymous type it compares that type with those it already knows about. This comparison is done using the 'C' equivalence rules modified to incorporate extra discrimination (described below). If a match is found and a name already exists that name is used else a new name is created and used.

In 'C', equivalent types may be interchanged and so are commonly unnamed. In Pascal, types are not equivalent even when they are textually the same. e.g., two variables of type **^INTEGER** may not be assigned unless **^INTEGER** is named (say **P_Int**) and the named type is used for both variables.

Because CtoP generates one name for all equivalent types, the problems of type equivalence disappear in the translated program.

5.12.1 Array Types

Two arrays of known size are equivalent if they have the same number of entries and their element types are the same.	Two ararys are equivalent if there were defined with the same name.
Two arrays, one whose size is omitted, are the same if the element types are equivalent.	A conformant arrays.

If an array definition used a *define* name to specify the number of element that name is carried over to the output by CtoP, ie

#define max_lines 24

int l[max_lines];

l :Array[max_lines] Of Integer;

Note that this usage creates an array with one more element than its 'C' equivalent. In comparing arrays for equivalence any *define* names used in their declaration are taken into account.

#define twenty_four 24
int xyz[twenty_four];

CtoP does not consider *l* and *xyz* to be equivalent, even though they have the same number of elements.

5.12.2 Enumeration, Structure and Union Types

'C' uses the same rules as Pascal: only named *struct*s etc can be equivalent, unless the variables are declared in the same declaration:

struct cell { int i; };

struct cell x, y;
struct cell z;
struct { int i; } a, b;
struct { int i; } c;

x, *y* and *z* are equivalent. *a* and *b* are equivalent.

CtoP will actually make all six variables equivalent, but this does not introduce any problems:

```
TYPE
 cell = RECORD
      i : INTEGER;
      END;
 …
VAR
 x : cell;
 y : cell;
 z : cell;
 a : cell;
 b : cell;
 c : cell;
```

Note that the named type has been used throughout. CtoP attempts to use type names supplied in the 'C' source, if possible.

5.12.3 Typedef Names

'C' defines *typedef*-names as being synonyms for the type they are declared as. Thus, in the following, all the variables have the same type:

typedef int ∗iptr;
typedef int scalar;

int ∗p;
scalar ∗q;
iptr r;

5.13 Type Names And Abstract Declarators

'C' allows anonymous types to appear in casts and as the argument of *sizeof*.

CtoP will generate a name for these types (see page 66).

Chapter 6

Type Conversions

'C' provides many implicit conversions between types. These lead many 'C' programmers to become very lax about types; *int* and *char* tend to be mixed when small positive values are being manipulated, pointers and arrays are similarly confused, and so on.

A cast expression may explicitly convert a value to another type.	There are a few built-in conversion functions.
Implicit type conversions occur in expressions.	Implicit type conversions never occur.
Implicit type conversions occur in parameters to function.	Implicit type conversions never occur.

Explicit conversions allow programmers to override data structure by imposing their own, e.g., when an array of characters (treated simply as bytes of data) is accessed under the template of a structure:

```
char data[n];          /* output buffer */
struct entry {
  int quantity;
  char part_id [2];
  };

 /* set 3rd entry to "4 of part Ai" */
 *(int *) &data[8] = 4;
 strcpy( (struct entry *) &data[8] -> part_id, "Ai");
```

Explanation: If *data* were overlayed by an "array of *struct entry*", then, since *sizeof(struct entry)* is *4* (generally), *&data[8]* is the address of the third *struct entry* (*data[0* through *3]* cover the first *struct entry*, *data[4* through *7]* cover the second and so on)

&data[8] is the address of the 8th character in *data*, and has type "pointer to *char*".

Cast to "pointer to *int*", it is dereferenced to set *quantity*.

Cast to "pointer to *struct entry*", it is dereferenced and the *part_id* set to the required value; *strcpy* is a standard 'C' routine to assign strings.

This sort of thing tends to happen because ‘C’ programmers don’t take the trouble to define their data structures explicitly.

In Pascal, of course, the programmer is required to be precise and to explicitly declare the overlaying implied in the above ‘C’ example.

See page 27 for an example of how *union*s can be used, and what Pascal code is implied.

6.1 Representation Changes

Signed integer types are assumed (by CtoP) to be held in 2’s complement form.

No assumptions are made (or are needed) about how floating-point types are held.

Information may be lost due to truncation when a larger type is assigned to a smaller one. Explicit casting may produce the same effect e.g., *(int) (char) i* will narrow *i*, then widen it. This sort of change is implementation dependent as it relies upon the underlying representation.

UCSD provides long integers (usually represented in BCD form) to hold values larger than **INTEGER** will allow. These may be used to represent ‘C’s *long int*s, provided bit manipulation is not required.

To be expanded…

6.2 Trivial Conversions

The differences between ‘C’ and Pascal, if any, do not affect CtoP.

6.3 Conversions To Integer Types

6.3.1 From Integer Types

Unsigned integer types cause two problems.

1. Converting from signed to unsigned is defined to map i to j in the range $0..2^n-1$, such that $j = i \bmod 2^n$, for n-bit, unsigned integers.

   Pascal subranges can only cover $0..2^n-1$ for $n+1$ bits.

2. Converting from unsigned to signed, the same congruency is used. Thus large positive integers will become negative.

   Again, the problem is the subrange available in Pascal.

CtoP will flag all such conversions.

To be expanded…

6.3.2 From Floating-point Types

The result of this conversion is implementation dependent in 'C' (although rounding is implied). CtoP leaves it up to the programmer to choose **ROUND** or **TRUNC**.

6.3.3 From Enumeration Types

Most compilers will perform implicit conversion.	Explicit conversion can be achieved using the built-in function **ORD**.

To be expanded…

6.3.4 From Pointer Types

Treated as an unsigned integer.	An illegal conversion.

To be expanded…

6.4 Conversions To Floating-point Types

UCSD Pascal allows either single or double precision to exist, but not both at the same time.

6.4.1 From Floating-point Types

Mixed use of *float* and *double* are flagged by CtoP.

6.4.2 From Integer Types

int expressions are converted to *double* in those contexts requiring a floating type.	An **INTEGER** expression is never implicitly converted to **REAL** type except in one context, addition. A method of explicitly converting an **INTEGER** to a **REAL** expression is to add **0.0**.

Pascal and 'C' have the same rules for integer to floating-point conversions.

6.5 Conversions To Structure And Union Types

A *struct* (or *union*) may only be assigned to another variable of the same type. This is the same in both languages.

6.6 Conversions To Enumeration Types

Performed using casts.	No conversion of enumerated types are available.

6.6.1 From Integer Types

Permissible provided an enumeration identifier exists with the given integer value.	Illegal.

6.7 Conversions To Pointer Types

To be expanded…

6.7.1 From Pointer Types

A "pointer to *x*" may be converted to the type "pointer to *y*".	Only the **NIL** pointer may be used in any pointer context.

6.7.2 From Integer Types

0 is assumed to be the null pointer.

Casts may be used to change type.	No conversions are possible.

6.7.3 From Array Types

See **§5.4 Pointer Types, §5.5 Array Types**.

6.7.4 From Function Types

Assigning a function name to a pointer to function causes an explicit conversion to occur.	Functions may not be assigned.

6.8 Conversions To Array And Function Types

No such conversions are possible in either language.

6.9 Conversions To The Void Type

This conversion simply discards the value of an expression. Every statement in 'C' is an expression whose value is discarded.

'C' expressions of the form *a=b* return *b* as the value. If this expression is used at the statement level this value is discarded. CtoP spots assignments and assignment operations at the statement level and simply creates an assignment. Those 'C' expressions that are not legal Pascal statements are handled by assignments to dummy variables.

6.10 The Casting Conversions

Any of the preceding conversions may be performed explicitly using a cast. See page 45 for more details on casts.	Apart from the built-in functions there is no other form of simple conversion.

6.11 The Assignment Conversions

In 'C' if the types of expressions on the left and right of an assignment do not agree, the following, implicit, conversions are performed:

Left Hand Side	**Right Hand Side**
any arithmetic type	any arithmetic type
any pointer type	the integer constant *0*
pointer to …	array of …
pointer to function	function

In Pascal the types of the left and right handside of assignment statements must be the same.

6.12 The Usual Unary Conversions

'C' performs some implicit conversions on operands. The aim is to reduce the number of cases that operators have to deal with:

Original Operand Type	**Converted Type**
char, *short*	*int*
unsigned char	*unsigned*
unsigned short	*unsigned*
float	*double*
array of …	pointer to …
function returning …	pointer to function returning …

6.13 The usual Binary Conversions

For binary operators, 'C' performs some conversions on their operands.

1. The unary conversions are applied.
2. If one operand is of type *T*, then the other is converted to that type, where *T* is checked to be, in order,

   a) *double*

   b) *unsigned long int*

   c) *long int*

   d) *unsigned*

   e) *long*

   f) *int*

   so both operands end up as large as the larger of the two.

Pascal will float integer expressions within an expression if one of the operands has type real.

6.14 The Function Argument Conversions

The unary conversions are performed on arguments before they are passed as actual arguments. See above.

6.15 Other Functions Conversions

See **§9.4 Adjustments To Parameter Types**.

Chapter 7

Expressions

7.1 Objects and Lvalues

For information only:

‘C’ has the concept of an object, lvalue and rvalue.

An object is a region of memory that can have values stored into or read from it.

A lvalue is an expression that refers to an object.

A rvalue is an expression that produces a value.

The difference between a lvalue and a rvalue is that lvalues can have the objects they point to changed, i.e., by assignment.

Pascal does not have these concepts. It does talk about variable-access and value-access.

At the level of the programmer reading source code these designations are not worth worrying about. Here we will give the rules and describe the differences regarding what can occur where.

7.2 Expressions and precedence

Some binary operators associate to the right.	All operators associate to the left.

The precedence of some operators differ between the two languages. CtoP will insert parenthesis where precedence differences exist across languages.

7.2.1 Kinds of expressions

‘C’ has a few oddities in this area. There are no constructs that cannot be handled by Pascal.

7.2.2 Precedence and associativity of operators

Order of precedence, highest to lowest:

Unary	**Unary**
Postfix ++ --	
Prefix ++ --	
sizeof	
casts	
~	**NOT**
!	
-	**-**
&	
*	
Binary	**Binary**
/ %	*** / DIV MOD**
+ -	**+ -**
<< >>	
< > <= >=	**= <> <= < >= >**
== *!=*	
&	
^	
|	
&&	**AND**
||	**OR**
? :	
= += -= *= %= ^= |=	
,	

7.2.3 Overflow

Overflow is not defined on signed values.	An error occurs if the right operator of **DIV** or **MOD** is zero.
Defined for unsigned values.	

Since we are converting working programs overflow should not be a problem. There may be situations where overflow occurs and the programmer has made use of this fact. Provided the underlying representation is the same in both languages this should be ok.

7.3 Primary expressions

7.3.1 Names

The name of a variable declared to be of arithmetic, pointer, enumeration, *struct* or union type is evaluated to an object of that type and is a lvalue.

The name of an array is not a lvalue. In those contexts where the usual unary conversions are applied the name evaluates to a pointer to the first element of that array.

The name of a function evaluates to that function. It is not a lvalue. In those contexts where the usual unary conversions are applied the function is converted to a pointer to that function.

Label names may not be used in expressions.

Typedef names may appear in cast expressions

The name of a variable declared to be of arithmetic, pointer, enumeration, **RECORD**, **BOOLEAN**, **CHAR** or **ARRAY** to an object of that type and is a lvalue.

7.3.2 Literals

See page 6.

7.3.3 Parenthesis

Reordering of expressions involving parenthesis by the compiler is considered reasonable.

Some compilers do coercions of expressions within parenthesis.

A programmer may use parenthesis to specify an order of evaluations. A compiler may not change this order.

‘C’s case is the most unsafe. Converting to Pascal therefore removes potential problems.

7.3.4 Subscripts

Arrays may only have one dimension.	Arrays may have more than one dimension.
a[i++, j]=0; is equivalent to: *i++; a[j]=0;*	**a[i, j]:=0;** a two dimensional array access.

This topic is more a cause of misunderstanding by the Pascal programmer when reading 'C' than a serious conversion problem. The same expression means two completely different things in the two languages.

7.3.5 Component selection

The syntax rules for accessing addresses of structures are different to those for actual structures.	The syntax is the same in both cases.
struct cell temp, ∗head;	**TYPE** **P_cell = ^cell;** … **VAR** **temp : cell;** **head : P_cell;**
temp.val = 1; *head->val = 2;*	**temp.val := 1;** **head^.val := 2;**
Functions may return structures whose fields are selected.	Functions may not return structures.

The programmer must add an extra argument and perform a procedure call with this argument.

7.3.6 Function calls

Functions may be called with missing, or extra arguments.	The number of arguments to functions and procedures must match those declared in the function header.
It is possible to call functions via pointers to functions.	Procedures and functions may be passed as parameters but it is not possible to point to them.
Parameters are widened at the point of call.	Parameters must be the same type, therefore no widening is necessary.

See page 37 for details of the widening operations.

Calls to functions via evaluation of a pointer to functions appears in the cross reference listing.

7.4 Unary operator expressions

7.4.1 Casts

It is possible to explicitly convert one type to another type.	A very restricted and predefined number of conversions are available.

It is possible to perform a similar operation in Pascal via variant records.

```
Var
    Two_Type :Record
        Case Integer Of
                    0 :(Int  :Integer);
          1 :(Ch   :^Char)
        End;
```

Here we have overlayed an integer with a pointer to char. By dot selecting the appropriate field we can access the same value as different types.

The types must have the same size; no truncation or widening is performed.

7.4.2 Sizeof

Can give a variable or a type as the parameter.	A UCSD extension allows a variable or a (?) type.
May be applied to an arbitrary expression.	May only be applied to names.

This is a compile time option. Thus *sizeof(i++)* does not cause *i* to be incremented.

7.4.3 Unary minus

For signed values the result is in the range *0-k*. For unsigned values the result is in the range *65536-k.*	As in 'C'.

7.4.4 Logical negation

Equivalent to *(X)==0.*	Converted to **((X)=0)**.
Returns *0* or *1*.	Returns **FALSE** or **TRUE**.

7.4.5 Bitwise negation

Performed on integer operands.	**NOT** causes the bitwise negation P-code to be generated in UCSD Pascal.

Flips each bit of its operand..

7.4.6 Address operator

Returns the address of a variable.	Can be simulated with a function call.

This operator occurs in two contexts:

a) Address is required for pointing to a data structure.

b) An address is passed as a parameter to a function call. Here it would be acting like a **VAR** parameter in Pascal or (a).

See page 62 for a discussion of how CtoP handles **VAR** parameters.

7.4.7 Indirection

Performs indirection through a pointer.	In most cases ^ is equivalent.

Usually used in the context of accessing parameters. In these situations passing the parameter by **VAR** in Pascal has the desired effect (ie an address has been passed).

7.4.8 Preincrement operator

Increments an integer or pointer and returns the incremented value.	Can be simulated with a function call in an expression context.
b=++a;	**a:=a+1; b:=a;**

7.4.9 Postincrement operator

Increments an integer or pointer and returns the old value.	Can be simulated with a function call in an expression context.
b=a++;	**b:=a; a:=a+1;**

7.4.10 Predecrement operator

Decrements an integer or pointer and returns the decremented value.	Can be simulated with a function call in an expression context.
b=--a;	**a:=a-1; b:=a;**

7.4.11 Postdecrement operator

Decrements an integer or pointer and returns the old value.	Can be simulated with a function call in an expression context.
b=a--;	**b:=a; a:=a-1;**

7.5 Binary operators

7.5.1 Multiplicative operators

'/' can be applied to any arithmetic type.	**DIV** for integer operands, '/' for reals.

7.5.2 Additive operands

Arithmetic may be performed on pointer types.	Arithmetic may not be performed on pointers.

See page 24 for a discussion of enumerated types.

7.5.3 Shift operator

To be simulated with function calls. Note that shifting *long ints* will not have the assumed effect in UCSD Pascal.

7.5.4 Inequality operator

May be performed on pointers. Use with enumerated types is dependent on the model employed to handle enumerated types.	May not be performed on pointers.

7.5.5 Equality operators

Problems arise in the area of unsigned conversions.

To be expanded…

7.5.6 Bitwise AND

Performed between integer operands.	Standard Pascal specifies that **AND** returns a result of **TRUE** or **FALSE** (represented by 1 and 0 respectively). UCSD Pascal **AND**s each bit of the two operands.

7.5.7 Bitwise OR

Performed between integer operands.	Standard Pascal specifies that **OR** returns a result of **TRUE** or **FALSE**. UCSD Pascal **OR**s each bit of the two operands.

7.5.8 Bitwise XOR

Performed between integer operands.	Not available in standard Pascal. Can be simulated by inline code in UCSD Pascal.

7.6 Logical operator expressions

Return the value *0* or *1*.	Standard Pascal defines that **FALSE** (0) or **TRUE** (1) should be returned. UCSD Pascal performs bitwise operations. **x AND TRUE** is guaranteed to return zero or one.

7.6.1 Logical AND operator

i && j	**(i<>0) AND (j<>0)** (see above)

7.6.2 Logical Or operator

i || *j*	**(i<>0) OR (j<>0)** (see above)

7.7 Conditional expressions

A ternary operator which provides an expression form of *if (…)…else…*	This operator is not available.

Only one of the *then* expression or the *else* expression is evaluated, and returned.

Substituting a Pascal function call may not have the equivalent effect.

z = a ? b++ : c++;	**z := IfThenElse(a, Inc(b), Inc(c));**

Because all arguments to the function call are evaluated both **b** and **c** will be incremented.

CtoP checks for expressions containing side effects. Conditional expressions free of side effects are output as function calls to **IfThenElse**. Conditional expressions containing side effects are output as follows:

```
(*
    FUNCTION F_a(q :BOOLEAN) :INTEGER;
    BEGIN
    IF q THEN
      F_a:=Inc(b)
    ELSE
      F_a:=Inc(c)
    END;
*)
z := F_a(a);
```

7.8 Assignment expressions

In ‘C’ assignment is a binary operator. This means that it may occur at the statement or expression level. Assignment operators are right associative (all other ‘C’ operatots are left associative). To be expanded out where possible. In other cases to be simulated by a function call.

7.8.1 Simple assignments

On assignment arrays are converted to pointers to the first element of that array.	On assignment the contents of the array on the right handside are copied into the memory locations occupied by the array on the left handside.
On assignment functions are converted to pointers to functions.	Functions or procedure may not be assigned.

CtoPwill correctly handle the array case. Assigning of functions or procedures results in an entry in the cross reference listing. See §? for details.

7.8.2 Compound assignment

i+=j;	**i:=i+j;**
a[l++]=k;*	**a[l]:=a[l]*k; l:=l+1;**

CtoP detects the precence of side effects on the left handside of the compound assignment and outputs a function form **TimesEqual(a[Postplus(l)], k)** of the expression. Where possible CtoP expands the compound assignment into its full form.

7.9 Sequential expressions

i=1, j=2;	**i:=1; j:=2;**
x=i++, j;	**i:=i+1; x:=j;**

At the statement level CtoP expands comma expressions into individual statements. At the expression level a function call is used.

7.10 Constant expressions

Compiler required to reduce constant expressions.	A constant expression may not be given where a constant is required.

Will be automatically handled by the CtoP preprocessor.

7.11 Order of evaluation

Boolean expressions must be short ciruited. *a && b* is defined as *if (a)* * b* *else* * 0* *a* || *b* is defined as *if (a)* * 1* *else* * b*	All subexpressions in a boolean expression must be evaluated.

Programmers may rely on short circuit evaluation in order to prevent out of range errors:

if (i < BUFSIZ && buf[i] != 1)…	**IF (i < BUFSIZ) AND (buf[i] <> 1) THEN** … This will cause an error if **i** is **>= BUFSIZ** since **buf[i]** will always be evaluated.

CtoP does not do anything about this situation.

7.12 Discarded values

An expression is a legal statement.	An expression on its own is not a legal statement.

Handled by assigning the expression to a dummy variable.

7.13 Compiler optimisations

We assume that any compiler optimisations are performed such that the original semantics are retained.

7.14 The Big Fake

Pointer arithmetic may be performed.	Pointer arithmetic may not be performed.

This feature is commonly used by 'C' programmers. The frequency of occurence coupled with incomprehension by Pascal programmers decided the design. CtoP would attempt to correctly translate this feature.

This is the one case where CtoP changes the structure of declarations and statements.

*int a[10], *p, i;*	**p : ARRAY[0..1] OF ^INTEGER;** **i :INTEGER;**
*i=*p++;*	**i:=p[Post_Inc(p)]^;**
i=++p;*	**i:=p[Pre_Inc(p)]^;**

The right handside expression in Pascal makes use of the order of evaluation of expressions.

p is an **ARRAY** of two pointers. The zeroth element is the pointer obtained after applying the increment/decrement operator. The first element is the current actual value of **p**. The functions **Post_Inc** and **Pre_Inc** increment the parameter passed, set up the array elements and return **0**.

The code is dirty. We are forced into using it by the frequent use of pointer arithemtic in 'C' and our desire to provide a good translation.

Chapter 8

Statements

Pascal contains all of the statement forms found in ‘C’, plus a few idiosyncrasies of its own.

8.1 General syntax

8.1.1 Semicolons

The ’*;*’ is part of the syntax of statements.	The ’**;**’ is a separator.

CtoP will attempt to remove redundant semicolons.

8.1.2 Control expressions

Expressions that control conditional or iterative statements are contained within parenthesis.	Pascal contains extra keywords.

This feature is just part of the surface syntax.

8.2 Expression statements

Expressions may be statements, *a=b* is an expression.	An expression is not a statement, **a:=b** is a statement.
(a<b) ? a : b	Can be simulated with a function call or **IF** … **THEN** … **ELSE**.

8.3 Labelled statements

See page 5 for details of the conversion of labels.

8.4 Compound statement

{	**BEGIN**
decl	**stmt**
stmt	**END**
}	

Declarations may only appear in the declaration part.

The declaration is moved to its appropriate Pascal context. The one situation where moving the declaration may have other effects is the case of frequent allocation and deallocation of large amounts of local storage. The storage consumed by the declaration in the compound statement is freed up upon leaving the block. Thus a series of blocks each with their own local storage would use the same actual storage in turn. By moving the storage to the procedure level this storage is no longer shared. See page 11 for details about the handling of nested declarations.

8.5 Conditional statements

See page 49 for a discussion on the evaluation of boolean expressions.

8.5.1 Dangling Else problem

'C' and Pascal resolve this problem the same way.

8.6 Iterative statements

8.6.1 While statement

while (expr)	**WHILE (expr) DO**
stmt	**stmt**

See page 49 for a discussion on the evaluation of boolean expressions.

8.6.2 Do statement

do	**REPEAT**
stmt	**stmt**
while (expr)	**UNTIL NOT (expr)**

CtoP will attempt to remove the **NOT** by reversing the boolean condition in (expr).

See page 49 for a discussion on the evaluation of boolean expressions. Converted into the Pascal repeat statement.

8.6.3 For Statement

The 'C' *for* statement:

```
for (expr1; expr2; expr3)
  stmt;
```

is equivalent to:

expr1;	**expr1;**
while (expr2)	**WHILE expr2 DO**
{	**BEGIN**
stmt;	**stmt;**
expr3;	**expr3**
}	**END;**

The 'C' for loop thus requires *expr2* to be evaluated on every iteration. The various vagaries of the 'C' for loop are be handled.

8.6.4 Multiple control variables

These are be correctly handled by CtoP.

8.7 Switch statement

The *case* labels are bound to the *switch* by semantic rules.	The case labels are bound to the **CASE** by syntax rules.
case labels bind to the closest surrounding *switch*.	Binding is automatic from the grammar.
case labels may prefix any statement inside the *switch* compound.	Case labels have a statement body bound to them

switch(x) *{* *case 1: y=2;* *if (b)* *case 2: y=4;* *}*	Illegal Pascal.
Flow of control is not influenced by the presence or absence of *case* labels.	Flow of control inside a compound statement belonging to a case label continues with the first statement after the **CASE** on reaching the end of that compound statement.
The label *default* specifies the code to be executed if the selector does not match any of the *case* labels. *break* causes the flow of control to goto the first statement after the *switch*.	If there is no label to match the selector value execution continues with the first statement after the **CASE**.

CtoP

- If a *break* is not the last statement before a *case* label a **GOTO** is inserted to cause the Pascal flow of control to enter the following arm of the **CASE**.
- Any *default* option is carried over into the output file. The possible side effects caused by evaluating the selector and fall through were considered too complex to be handled automatically.
- *case* labels not at the correct level in the Pascal grammar sense, are flagged.
- If the selector expression has type *char* the values of the *case* labels must also have type *char*. This will cause problems if the character escape feature of ‘C’ is used, see page 7. CtoP will output that value surrounded by a **CHR()** . It is upto the programmer to take appropriate action.

Recommendations

If use has been made of the fall through of *case* labels in 'C' the case_statement_list should be made into a procedure call, i.e.,

```
switch (c)
  {
  case 'a': z=3;
  case 'b': m=4;
  };
```

CtoP generates:

```
CASE c OF
 'a':BEGIN
    z:=3;
    GOTO 1
    END;
 'b':BEGIN
    1:
    m:=4
    END
 END;
```

Programmer turns this into:

```
CASE c OF
 'a':BEGIN
    z:=3;
    Body_of_B
    END;
 'b':Body_of_B
 END;
```

where **Body_of_B** is a procedure containing the code from the body of the case label **'b'**.

8.8 Break and Continue

8.8.1 Using break

{	**BEGIN**
…	…
break;	**GOTO 1;**
…	…
}	**END;**
	1:

8.8.2 Using continue

{	**BEGIN**
…	…
continue;	**GOTO 2;**
…	…
}	**2:**
	END;

8.9 Return statement

Causes the optional expression to be assigned to the function name and the function to return to the calling function.	Not available.

To be simulated by assigning to the function name and calling exit.

8.10 Goto statement

See page 5 for a discussion on labels. Converted to the equivalent Pascal.

8.11 Null statement

The ';' is a null statement in both languages.

Chapter 9

Functions

There are functions.	There are functions and procedures.

9.1 Function definitions

Functions may be called before they are defined.	Procedures and functions must be defined before use.

CtoP provides cross reference information to help the programmer order declarations.

To be expanded…

9.2 Function types

Unless explicitly specified a function always returns a given type.	Procedures do not return a value. Functions always return a value.
Functions may return anonymous types.	A function always returns a named type.

See **§10.4** for a discussion of how CtoP assigns names to anonymous types.

9.3 Formal parameter declarations

The types of formal parameters may be anonymous.	The type of a formal parameter must be a named type.
Parameters are always passed by value.	Parameters may be passed by **VAR** or by value.
Parameters may be omitted.	Parameters may not be omitted.
Extra parameters may be passed.	Extra parameters may not be given.

9.4 Adjustment to parameter types

It is suggested that Pascal programmers sit down before continuing this section.

'C' specifies that certain adjustments must be made to function arguments when passed and when accessed in the function body.

```
int f(c)                                    int f(c)
char c;                                     int c;
{                                           {
  int i;                                      int i;
  i = c;            …is equivalent to…        i = (int)(char) c;
}                                           }
```

Thus the argument passed is widened to an *int* after it is evaluated. Accesses to that argument return narrowed values. The widening operations are given in **§6.12**. Many compilers do not perform this widening and narrowing operation. Its effects only become noticeable if:

1. The quantity being widened is treated as signed.
2. A value outside of the range of the parameter is passed. The programmer relying on the top bits being truncated by the narrowing.

9.5 Parameter passing conventions

All parameters are passed by value.	Parameters may be passed by value or address.
swap(a, b) *int *a, *b;* *{* *int temp;* *temp=*a;* **a=*b;* **b=temp;* *}*	**PROCEDURE swap(VAR a, b :INTEGER);** **VAR** **temp :INTEGER;** **BEGIN** **temp:=a;** **a:=b;** **b:=temp** **END;**

Giving an array as a parameter causes a pointer to the first element of that array to be passed.	For a value parameters a copy of the array is made. For **VAR** parameters the address of the base address of the array is passed.
Arrays of any size may be passed as parameters to a given function.	Arrays passed as parameters must be the same type as that given in the formal parameter declaration, unless this is a conformant array.

CtoP will attempt to spot parameters that should be passed by **VAR**. If it is decided that a parameter is to be output as a **VAR**, occurrences of that variable in the body of the function will not appear with an accompanying pointer dereference.

Because of the duality of arrays and pointers in ‘C’ it is not always possible to determine if a formal parameter declared as a pointer to a type is simply that or a pointer to an array of that type.

```
do_something(z)
  char *z;
{
*z = 'a';
z[0] ='a';
}
```

CtoP looks at the context in which formal pointer parameters are used in order to determine if an array is intended.

If CtoP can establish that the parameter really is an array, the parameter declaration is altered to a conformant array and the parameter references in the function body are treated appropriately.

If CtoP cannot establish that the parameter is an array, but can establish that it is **VAR**, the declaration is altered from "pointer to T" to "**VAR** T" with a conformant array declaration in comments.

9.6 Agreement of formal and actual parameters

No checking is performed to ensure that the type of the parameters passed to functions match the type required by the formal.	The type of the actual parameter must be the same as the formal parameter.

CtoP assumes that any 'C' function returning a value is a function in the Pascal sense.

A cross reference list of calls to functions with non-matching arguments is provided.

To be expanded…

9.7 Function return types

A function may return any type except "array of …" or "function returning …".	A function may only return scalar types.

Functions returning non-scalar types appear in the cross reference. It is suggested that an extra parameter be added to the function and a value passed back through this parameter.

9.8 Agreement of actual and declared return type

Historically it has been possible to return/not return values from any function.	A procedure does not return a value. A function always returns a value.

'C' does specify that the result of not providing a return expression where one is required is undefined.

Chapter 10

Program structure

10.1 Introduction

Program execution starts with the first statement of a function called *main*.	Program execution starts with the first statement of the body belonging to the **PROGRAM** heading.
The smallest object that can be linked is a function.	The smallest object that can be linked is a unit of compilation.
A linker is usually used to bind all of the functions called by the main function and all the functions that call, etc into one code file.	A librarian or the operating system can be used to locate and use the required units.
Functions can be compiled without any reference to how they slot into the overall program.	Individual functions may not be compiled separately. A **UNIT** must be used to encapsulate the code.

10.2 Units or include files

The programmer converting a 'C' program into Pascal has two choices:

1. Create one large program and use include files to import all of the necessary source code.
2. Create units and use these units where needed.

Option 1. has the advantage of requiring the least amount of initial effort. However, a single monolithic program will be difficult to develop and maintain.

Creating a suite of units will require a larger initial effort in man power. However, independent units will be simpler to develop and maintain. With independent units it will also be possible for several people to work on different parts of the program simultaneously.

10.3 What belongs in a unit?

CtoP makes the assumption that all of the source code in a single file is a potential unit.

Some of the source files will include other files. A method of specifying which files should be considered as single units is provided. The convention is that those source files given as input in the main control file are regarded as potential units. Each of these input files is output to a separate file and has cross reference information associated with it.

The cross reference given for the output file is intended to provide the information needed to create a compilable unit.

A **UNIT** will need declarations from elsewhere and other units will need information to be exported from this unit.

CtoP automatically generates the **INTERFACE** and **USES** for each specified source **UNIT**.

CtoP also provides a list of all (external) identifiers referenced in a source unit that are not defined anywhere within the suite of 'C' programs. Typically, such identifiers will be library routines if the source of the library has not been supplied. This list is included in the **INTERFACE** section of the translated program, preceded by a noticeable comment to that effect.

10.4 Types

Typenames may be defined at the global level in every compilable file.	Because of the explicit importing of other units names must not clash. However, this explicit importing means that type definitions need not be redefined, the same definition can be shared.

CtoP gives names to anonymous types.

In order to prevent type name clashes and reduce multiplicity of type definitions all global type definitions and created types are output to a type unit. This type unit consists solely of type definitions in its interface part. This unit is then **USE**d by every other compilation unit. There is no runtime overhead incurred by placing type definitions in a single unit.

This TypeUnit also contains global variables generated by CtoP (such as **C_Null_String**).

Chapter 11

The runtime library

'C' does not have the built-in functions and procedures that are available in Pascal (such as **WRITE**, **CHR** and so on), but many routines are assumed to be available as part of a "standard library". These routines include:

- Character operations e.g., *isdigit*, *toupper*
- Strings operations e.g., *strcpy*
- Mathematical operations e.g., *abs*
- Storage management e.g., *malloc*
- Input/Output operations e.g., *printf*, *fopen*

The declarations of these facilities usually appear in header files which are included by the 'C' programs that use them.

Some of the functions may be implemented as macros, others will be actual functions. CtoP will expand out all the macro definitions.

If the source of any libraries used by the program is available, these may be used as the basis of a Pascal **UNIT** implementing the 'C' runtime library. The runtime library may be difficult to mimic since these routines usually make use of 'C's ability to pass arbitrary arguments and perform type conversions.

Some library routines access and set an external error code variable. The variable and its possible values are defined in the header file *errno.h.* These error codes are implementation dependent but the following are standard:

```
/* global error code variable: */
extern int errno;
/* for mathematical functions: */
#define EDOM …
/*      argument not in domain of function */
#define ERANGE …
/*      result is out of range */
```

11.1 Character Processing

11.1.1 Classification

int is…(c)	**FUNCTION is**…**(c : CHAR)**
char c;	**: INTEGER;**

e.g., isdigit returns a non-zero value (i.e., **TRUE**) if c is a digit (i.e., **c IN ['0'..'9']**).

Some of these routines assume the ASCII character set with *EOF* defined as a special character (*-1*) representing end-of-file.

11.1.2 Conversion

int to…(c)	**FUNCTION to**…**(c : CHAR)**
char c;	**: INTEGER;**

e.g., toint returns the "weight" of a (hexadecimal) digit: 0 for '0' … 9 for '9', 10 for 'A' or 'a' … 15 for 'F' or 'f'

11.2 String Processing

'C' has the convention that strings are stored as "array of *char*", with a null character (*'\0'*) as a terminator. e.g., the string *"abc"* is stored as *'a', 'b', 'c', '\0'*, which has 4 elements.

The null string, written *""*, is represented as an array with one element, the null character.

The common string handling routines include:

strcat(s1,s2)	appends *s2* to *s1* and returns *s1* (*char* ∗). *s1* must be large enough to hold the result string.
strcmp(s1,s2)	compare *s1* with *s2*, using standard, lexicographical, ordering, and return *0* if they are equal, *n<0* if *s1* is "less than" *s2*, else return *n>0*. Usually *-1* and *1* are returned on inequality.
strcpy(s1,s2)	copy *s2* to *s1*, and return *s1* (*char* ∗). Often used by 'C' programmers to move any block of (null character terminated) data from one address to another.
strlen(s)	return the number of characters (up to, but not including, the null character) in the string *s*.

Other functions may be provided to search for a character within a string, and so on, but usually 'C' programmers provide their own routines for anything more exotic than the

above.

11.3 Mathematical functions

These are generally self-explanatory, and most standard mathematical functions are provided in Pascal.

11.4 Storage Management

'C' does not have a heap in the Pascal sense, but provides, through some "standard" routines, a simple form of heap memory management.

A fixed, *extern*, data space is assumed which represents the heap. Storage may be requested from, and returned to, this space using the following routines:

```
char *malloc(size)
 unsigned size;

void free(ptr)
 char *ptr;
```

malloc returns a pointer to a contiguous area, *size* characters long. *free* returns the area pointed at by *ptr* to the storage pool.

struct cell { * struct cell *next;* * …* *};*	**TYPE cell = RECORD** ** next : ^ cell;** ** …** ** END;**
*struct cell *cp;*	**VAR cp : ^ cell;**
*cp = (struct cell *)* * malloc(sizeof(struct cell));*	**new(cp);**
free(cp);	**dispose(cp);**

Note that, in 'C', the pointer returned by *malloc* must be cast to the correct type, and that the size of the structure must be specified.

CtoP will simply convert such 'C' source into Pascal. (see page 42 on casts) The programmer must then modify the program to use the Pascal heap.*

11.5 Standard I/O

I/O in 'C' is handled by "standard" library routines. The header file *stdio.h* contains the declarations of all the routines and data structures used for input/output.

* or write a Pascal equivalent to *malloc* and *free* that simulates the heap with some global array.

End-of-file is defined to be a special "character" whose value is *-1*. *stdio.h* contains the macro definition:

#define EOF (-1)

Input/Output routines refer to "streams" (the equivalent of Pascal's file variables), which have type "pointer to *FILE*". The standard input stream (the file variable **INPUT** in Pascal) is called *stdin*, and the standard output stream (**OUTPUT**) is called *stdout*. A stream called *stderr* is also declared (usually associated with the terminal) to which error messages may be written.

11.5.1 Using stdin and stdout

int getchar()
int getc(stream)
*FILE *stream;*

reads the next character from *stdin*. *EOF* is returned when end-of-file is encountered. *getchar()* is equivalent to *getc(stdin)*.

int putchar(c)
char c;
int putc(c, stream)
*FILE *stream;*

write the character *c* to *stdout*, and return the (ASCII) value of that character. *putchar(c)* is equivalent to *putc(c, stdout)*.

*char *gets(s)*
*char *s;*

reads a string from *stdin*. *gets* reads up to newline or end-of-file, and returns the string made up of all the characters read up to that point. The null pointer is returned if an error occurs, else the address of *s* is returned.

int puts(s)
*char *s;*

writes the string *s* to *stdout*. Returns *EOF* if any error occurs.

c = getchar();	**read(c);**
putchar(c);	**write(c);**
if ((c=getchar()) == EOF) …	**IF EOF(INPUT) THEN** … **ELSE read(c);**

int printf(format, arg1, arg2, …)
*char *format;*

printf is a truly horrible product of 'C's flexible functions! *printf* writes its arguments to *stdout*, according to the formatting information in the string *format* (which also determines how many arguments there should be).

int scanf(format, ptr1, ptr2, …)
*char *format;*

scanf is the sister of *printf*, for reading from *stdin* according to *format*, storing items read in at locations addressed by its *ptr*-arguments.

11.5.2 File I/O

The 'C' standard library provides two levels of file access:

1. Via "streams", using *FILE* as mentioned above. Routines at this level operate on and/or produce "streams" which are "pointer to *FILE*". The actual *FILE* objects are allocated by routines such as *fopen*, and the programmer only declares pointers to these. A *FILE* will contain such information as current position in file and the "file descriptor" (see 2.)
2. Via "file descriptors". This is the low-level I/O. Most of the routines are very similar to those used with "streams" except that they take an integer "file descriptor" instead of a "stream". These "file descriptors" are indices into an internal table containing relevant information needed by the file system.

Only the high-level routines are briefly described here.

*FILE *fopen(pathname, type)*
*char *pathname, *type;*

type is a string specifying the manner of access required:

"r"	Open an existing file for reading.
"w"	Create a new file, or truncate an existing one, for writing.

"a"	Create a new file, or append to an existing one, for writing.
"r+"	Open an existing file for update (i.e., reading and writing).
"w+"	Create a new file, or truncate an existing one, for update.
"a+"	Create a new file, or append to an existing one, for update.

```
FILE *s1, *s2;                          VAR s1, s2 : TEXT;

s1 = fopen("xyz:abc.text", "r");        reset(s1, "xyz:abc.text");

s2 = fopen("v:fred.text", "w");         rewrite(s2, "v:fred.text");

fclose(s1);                             close(s1);

if (fclose(s2) == EOF)                  close(s2);
 puts("Error closing s2\n");            IF IOresult <> 0 THEN
                                         writeln('Error closing s2');
```

Most of the file I/O routines are similar to the *stdin/stdout* routines described above but have "*f*" preceding the routine name and take an extra (final) argument which is a "stream".

```
int f…(…, stream)
 FILE *stream;
```

There are a few exceptions to this; notably *fprintf* which has the "stream" as the first argument.

```
int fread(ptr, size_of_ptr, count, stream)
  char *ptr;
  unsigned size_of_ptr;
  int count;
  FILE *stream;
```

This reads *count* items of size *size_of_ptr* into the buffer pointed at by *ptr*. If the first argument passed has type "pointer to *T*", then the second argument should be *sizeof(T)*. *fread* returns *0* if no data was read or an error occurred, else it returns the number of items actually read (which may be less than *count*).

```
int fwrite(ptr, size_of_ptr, count, stream)
```

This is the matching output routine to *fread* above.

```
long ftell(stream)
 FILE *stream;
```

ftell returns the current file position for the given "stream". The result is suitable as a second argument to *fseek*, below.

int fseek(stream, offset, type)
*FILE *stream;*
long offset;
int type;

Sets the current file position for the given "stream" according to the value of *type*:

0	Absolute - file position set equal to *offset*
1	Relative (to current position) - file position set equal to current position + *offset*. *offset* may be positive or negative.
2	Relative (to end-of-file) - file position set to end-of-file + *offset*. Often used to extend a file (on output, with a positive *offset*)

Chapter 12

Cross reference information

12.1 Introduction

There are two aims in producing the cross reference information:

1. Help the programmer find his/her way about the original source code.
2. Highlight those areas where conversion work still has to be done. This information in turn breaks down into to areas:

    a) Where conversion still has to be done.

    b) Information needed to convert a particular statement that is scattered throughout the source.

The cross reference information occurs in two places:

1. In the output source file. This happens for those cases where the conversion can be done in a purely local context.
2. In the cross reference file. Information gets written here for those conversions that require extra global information to generate the correct equivalent Pascal.

12.2 Cross reference information in the Pascal output

12.2.1 Could be boolean

'C' does not have the data type boolean. Instead *int* is used. Zero signifies false, non-zero true. CtoP looks at the contexts in which variables occur. Variables which appear in a context requiring a **BOOLEAN** datatype are marked with a comment at the point of their declaration.

12.2.2 Casts

The cast conversion is output in its 'C' form.

12.3 The cross reference file

12.3.1 Functions called with incorrect arguments

The format of the cross reference information is:
Unit name - Information for functions in this unit

Function name Parameters
Called from unit name, function name, line number

Function name Parameters
Called from unit name, function name, line number

....
....

12.3.2 Functions assigned to pointers

The format of the cross reference information is:

Unit name - Information for functions in this unit

Function name Parameters
The format of the cross reference information is:

Unit name - Information for functions in this unit

Function name Name, function name where assigned to, line no

Function name Name, function name where assigned to, line no

....
....

12.3.3 Pointers to functions that are called

The format of the cross reference information is:

Unit name - Information for variables in this unit

Variable name Unit name, function name where assigned to, line no

Variable name Unit name, function name where assigned to, line no

....
....

12.3.4 Mixed use of float and double in expressions

Occurrences of the use of mixed precision real arithmetic in expressions will be listed. The format is:

Unit name - Information for variables in this unit

Function name, line number. If a global variable is used then the unit in which that variable is declared.

....
....

Contents

1 Introduction to CtoP 1

1.1 Using CtoP ... 1
1.2 Reading this manual ... 1
1.3 Which C? ... 1
1.4 An overview of C programming ... 2

2 Lexical Elements 3

2.1 Layout of the source code ... 3
2.2 The source character set ... 3
2.3 Comments ... 3
2.4 Tokens ... 4
2.5 Operators and separators ... 4
2.6 Identifiers ... 4
2.6.1 #define names ... 5
2.6.2 Labels ... 5
2.7 Reserved words ... 5
2.8 Constants ... 7
2.8.1 Integer constants ... 7
2.8.2 Floating-point constants ... 7
2.8.3 Character constants ... 7
2.8.4 String constants ... 7
2.8.5 Escape characters ... 8
2.8.6 Character escape codes ... 8
2.8.7 Numeric escape codes ... 8

3 The C preprocessor 10

4 Declarations 11

4.1 Organization of Declarations ... 11
4.2 Terminology ... 11
4.2.1 Scope ... 11
4.2.2 Visibility ... 13
4.2.3 Forward References ... 13
4.2.4 Overloading Of Names ... 14
4.2.5 Duplicate Declarations ... 14
4.2.6 Duplicate Visibility ... 14
4.2.7 Extent ... 16

4.2.8 Initial Values ... 16
4.2.9 External Names ... 16
4.3 Storage Class Specifiers ... 16
4.3.1 Default Storage Class Specifiers ... 18
4.4 Type Specifiers ... 18
4.4.1 Default Type Specifiers ... 18
4.4.2 Missing Declarators ... 19
4.5 Declarators ... 19
4.5.1 Simple Declarators ... 19
4.5.2 Pointer Declarators ... 19
4.5.3 Array Declarators ... 20
4.5.4 Function Declarators ... 20
4.5.5 Composition of Declarators ... 21
4.6 Initializers ... 21
4.6.1 Integers ... 21
4.6.2 Floating-point ... 21
4.6.3 Pointers ... 21
4.6.4 Arrays ... 21
4.6.5 Enumerations ... 21
4.6.6 Structures ... 21
4.6.7 Unions ... 22
4.6.8 Other Types ... 22
4.6.9 Eliding Braces ... 22
4.7 Implicit Declarations ... 22
4.8 External Names ... 22

5 Types 23

5.1 Storage Units ... 23
5.2 Integer Types ... 23
5.2.1 Signed Integer Types ... 24
5.2.2 Unsigned Integer Types ... 24
5.2.3 Character Type ... 25
5.2.4 Long Integers ... 25
5.3 Floating-Point Types ... 25
5.4 Pointer Types ... 26
5.4.1 Pointer Arithmetic ... 26
5.4.2 Some Problems With Pointers ... 26
5.5 Array Types ... 26
5.5.1 Arrays and Pointers ... 27
5.5.2 Multidimensional Arrays ... 27
5.5.3 Array Bounds ... 27
5.5.4 Operations ... 27
5.6 Enumeration Types ... 28
5.6.1 Detailed Semantics ... 28
5.7 Structure Types ... 28
5.7.1 Operations on Structures ... 28

5.7.2 Components ... 29
5.7.3 Structure Component Layout ... 29
5.7.4 Bit Fields ... 29
5.7.5 Portability Problems ... 30
5.7.6 Sizes of Structures ... 30
5.8 Union Types ... 30
5.8.1 Union Component Layout ... 30
5.8.2 Sizes of Unions ... 31
5.9 Function Types ... 31
5.10 Void ... 32
5.11 Typedef Names ... 32
5.11.1 Redefining Typedef Names ... 32
5.11.2 Implementation Note ... 32
5.12 Type Equivalence ... 32
5.12.1 Array Types ... 33
5.12.2 Enumeration, Structure and Union Types ... 33
5.12.3 Typedef Names ... 34
5.13 Type Names And Abstract Declarators ... 34

6 Type Conversions 35

6.1 Representation Changes ... 36
6.2 Trivial Conversions ... 36
6.3 Conversions To Integer Types ... 36
6.3.1 From Integer Types ... 36
6.3.2 From Floating-point Types ... 37
6.3.3 From Enumeration Types ... 37
6.3.4 From Pointer Types ... 37
6.4 Conversions To Floating-point Types ... 37
6.4.1 From Floating-point Types ... 37
6.4.2 From Integer Types ... 37
6.5 Conversions To Structure And Union Types ... 37
6.6 Conversions To Enumeration Types ... 38
6.6.1 From Integer Types ... 38
6.7 Conversions To Pointer Types ... 38
6.7.1 From Pointer Types ... 38
6.7.2 From Integer Types ... 38
6.7.3 From Array Types ... 38
6.7.4 From Function Types ... 39
6.8 Conversions To Array And Function Types ... 39
6.9 Conversions To The Void Type ... 39
6.10 The Casting Conversions ... 39
6.11 The Assignment Conversions ... 39
6.12 The Usual Unary Conversions ... 40
6.13 The usual Binary Conversions ... 40
6.14 The Function Argument Conversions ... 40
6.15 Other Functions Conversions ... 40

7 Expressions 41

7.1 Objects and Lvalues ... 41
7.2 Expressions and precedence ... 41
7.2.1 Kinds of expressions ... 41
7.2.2 Precedence and associativity of operators ... 42
7.2.3 Overflow ... 42
7.3 Primary expressions ... 43
7.3.1 Names ... 43
7.3.2 Literals ... 43
7.3.3 Parenthesis ... 43
7.3.4 Subscripts ... 44
7.3.5 Component selection ... 44
7.3.6 Function calls ... 44
7.4 Unary operator expressions ... 45
7.4.1 Casts ... 45
7.4.2 Sizeof ... 45
7.4.3 Unary minus ... 46
7.4.4 Logical negation ... 46
7.4.5 Bitwise negation ... 46
7.4.6 Address operator ... 46
7.4.7 Indirection ... 46
7.4.8 Preincrement operator ... 47
7.4.9 Postincrement operator ... 47
7.4.10 Predecrement operator ... 47
7.4.11 Postdecrement operator ... 47
7.5 Binary operators ... 48
7.5.1 Multiplicative operators ... 48
7.5.2 Additive operands ... 48
7.5.3 Shift operator ... 48
7.5.4 Inequality operator ... 48
7.5.5 Equality operators ... 48
7.5.6 Bitwise AND ... 49
7.5.7 Bitwise OR ... 49
7.5.8 Bitwise XOR ... 49
7.6 Logical operator expressions ... 49
7.6.1 Logical AND operator ... 50
7.6.2 Logical Or operator ... 50
7.7 Conditional expressions ... 50
7.8 Assignment expressions ... 51
7.8.1 Simple assignments ... 51
7.8.2 Compound assignment ... 51
7.9 Sequential expressions ... 51
7.10 Constant expressions ... 52
7.11 Order of evaluation ... 52
7.12 Discarded values ... 52
7.13 Compiler optimisations ... 53

7.14 The Big Fake ... 53

8 Statements 54

8.1 General syntax ... 54
8.1.1 Semicolons ... 54
8.1.2 Control expressions ... 54
8.2 Expression statements ... 54
8.3 Labelled statements ... 55
8.4 Compound statement ... 55
8.5 Conditional statements ... 55
8.5.1 Dangling Else problem ... 55
8.6 Iterative statements ... 55
8.6.1 While statement ... 55
8.6.2 Do statement ... 56
8.6.3 For Statement ... 56
8.6.4 Multiple control variables ... 56
8.7 Switch statement ... 56
8.8 Break and Continue ... 58
8.8.1 Using break ... 58
8.8.2 Using continue ... 59
8.9 Return statement ... 59
8.10 Goto statement ... 59
8.11 Null statement ... 59

9 Functions 60

9.1 Function definitions ... 60
9.2 Function types ... 60
9.3 Formal parameter declarations ... 60
9.4 Adjustment to parameter types ... 62
9.5 Parameter passing conventions ... 62
9.6 Agreement of formal and actual parameters ... 63
9.7 Function return types ... 64
9.8 Agreement of actual and declared return type ... 64

10 Program structure 65

10.1 Introduction ... 65
10.2 Units or include files ... 65
10.3 What belongs in a unit? ... 66
10.4 Types ... 66

11 The runtime library 67

11.1 Character Processing ... 68
11.1.1 Classification ... 68
11.1.2 Conversion ... 68
11.2 String Processing ... 68
11.3 Mathematical functions ... 69

11.4 Storage Management ... 69
11.5 Standard I/O ... 70
11.5.1 Using stdin and stdout ... 70
11.5.2 File I/O ... 71

12 Cross reference information 74

12.1 Introduction ... 74
12.2 Cross reference information in the Pascal output ... 74
12.2.1 Could be boolean ... 74
12.2.2 Casts ... 74
12.3 The cross reference file ... 75
12.3.1 Functions called with incorrect arguments ... 75
12.3.2 Functions assigned to pointers ... 75
12.3.3 Pointers to functions that are called ... 75
12.3.4 Mixed use of float and double in expressions ... 76