HP 9000 Computer Systems

HP C/HP-UX Programmer's Guide

Workstations and Servers

HP Part No. 92434-90011
Printed in U.S.A. May 1997

E0597

HEWLETT-PACKARD COMPANY
3000 Hanover Street
Palo Alto, California 94304 U.S.A.

Printing History

New editions are complete revisions of the manual. The dates on the title page change only when a new edition is published.

The software code printed alongside the data indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

First Edition	Nov 1986	
Second Edition	Nov 1987	MPE/XL: 31506A.00.02 HP-UX: 92453-01A.00.09
Update 1	Oct 1988	MPE/XL: 31506A.01.21 HP-UX: 92453-01A.03.04
Third Edition	August 1992	MPE/iX: 31506A.04.01 HP-UX: 92453-01A.09.17
Fourth Edition	January 1994	MPE/iX: 31506A.04.01 HP-UX: 92453-01A.09.61
Fifth Edition	June 1996	HP C/HP-UX A.10.32
Sixth Edition	May 1997	HP C/HP-UX A.10.33

You may send any suggestions for improvements in this manual to:

Languages Information Engineering Manager
Hewlett-Packard Company
Mailstop 42UD
11000 Wolfe Road
Cupertino CA 95014-9804

Preface

The *HP C Programmer's Guide* contains a detailed discussion of selected C topics for the HP 9000 Series computer systems. This manual is intended for experienced programmers who are familiar with HP systems, data processing concepts, and the C programming language. The manual does not discuss every feature of C. For more information, refer to the *HP C/HP-UX Reference Manual*.

This manual is organized as follows:

Chapter 1 **Introduction to HP C**
Introduces HP C.

Chapter 2 **Storage and Alignment Comparisons**
Compares HP C storage and alignment on different systems.

Chapter 3 **Calling Other Languages**
Describes how to call other languages from HP C.

Chapter 4 **Optimizing HP C Programs**
Describes how to use the optimizer.

Chapter 5 **Programming for Portability**
Describes how to make new programs easily transportable among HP systems.

Chapter 6 **Migrating C Programs to HP-UX**
Discusses issues for migrating C language programs from VAX computers and HP 9000 Series 300/400 and 500 computers to HP 9000 workstations and servers.

Chapter 7 **Using C Programming Tools**
Discusses various C programming tools.

Additional Documentation

Refer to the following materials for further information on C language programming:

American National Standard for Information Systems—Programming Language—C, ANSI/ISO 9899-1990.

HP FORTRAN/9000 Programmer's Guide—This manual explains how to call C programs from FORTRAN on HP-UX.

HP Pascal/HP-UX Programmer's Guide—This manual describes how to call C programs from Pascal on HP-UX systems.

COBOL/HP-UX Operating Manual—This manual provides information on calling C subprograms from COBOL programs on HP-UX. It also explains how to call COBOL subprograms from C.

HP-UX Reference—For HP-UX 10.30 the manpages are available in Instant Information under the title *HP-UX Reference* and via the `man` command. For HP-UX 10.20 the manpages are available in LaserROM and via the `man` command. They document commands, system calls, file formats, device files, and other HP-UX related topics.

HP-UX Linker and Libraries Online User Guide—This online help describes programming in general on HP-UX. For example, it covers linking, loading, shared libraries, and several other HP-UX programming features.

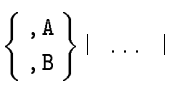
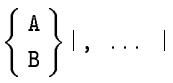
HP-UX Floating-Point Guide—This manual describes the IEEE floating-point standard, the HP-UX math libraries on HP 9000 systems, performance tuning related to floating-point routines, and floating-point coding techniques that can affect application results.

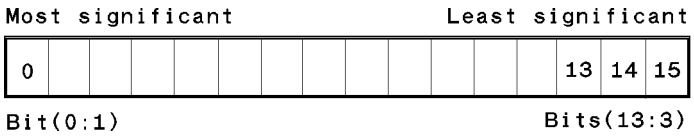
Conventions

NOTATION	DESCRIPTION
`UPPERCASE`	Within syntax descriptions, characters in uppercase must be entered in exactly the order shown, though you can enter them in either uppercase or lowercase. For example: `SHOWJOB` Valid entries: `showjob` `ShowJob` `SHOWJOB` Invalid entries: `shojwob` `ShoJob` `SHOW_JOB`
italics	Within syntax descriptions, a word in italics represents a formal parameter or argument that you must replace with an actual value. In the following example, you must replace *filename* with the name of the file you want to release: `RELEASE` *filename*
punctuation	Within syntax descriptions, punctuation characters (other than brackets, braces, vertical parallel lines, and ellipses) must be entered exactly as shown.
`{ }`	Within syntax descriptions, braces enclose required elements. When several elements within braces are stacked, you must select one. In the following example, you must select `ON` or `OFF`: `SETMSG { ON / OFF }`

NOTATION	DESCRIPTION
`[ ]`	Within syntax descriptions, brackets enclose optional elements. In the following example, brackets around `,TEMP` indicate that the parameter and its delimiter are optional: `PURGE {`*filename*`} [,TEMP]` When several elements with brackets are stacked, you can select any one of the elements or none. In the following example, you can select *devicename* or *deviceclass* or neither: `SHOWDEV` [*devicename* / *deviceclass*]

`[ ... ]`	Within syntax descriptions, a horizontal ellipsis enclosed in brackets indicates that you can repeatedly select elements that appear within the immediately preceding pair of brackets or braces. In the following example, you can select *itemname* and its delimiter zero or more times. Each instance of *itemname* must be preceded by a comma:
	`[,`*itemname*`][ ... ]`
	If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, the comma cannot precede the first instance of *itemname*:
	`[`*itemname*`][, ... ]`

NOTATION	DESCRIPTION
| ... |	Within syntax descriptions, a horizontal ellipsis enclosed in parallel vertical lines indicates that you can select more than one element that appears within the immediately preceding pair of brackets or braces. However, each element can be selected only one time. In the following example, you must select `,A` or `,B` or `,A,B` or `,B,A`:  If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, you must select `A` or `B` or `A,B` or `B,A` (the first element is not preceded by a comma): 
...	Within examples, horizontal or vertical ellipses indicate where portions of the example are omitted.
␣	Within syntax descriptions, the space symbol ␣ shows a required blank. In the following example, you must separate *modifier* and *variable* with a blank: `SET[(`*modifier*`)]␣(`*variable*`);`
KEY	The symbol KEY indicates a key on the terminal's keyboard. For example, RETURN indicates the carriage return key.
CTRL *char*	CTRL *char* indicates a control character. For example, CTRL `Y` means you have to press the Y key while holding down the CTRL key.

NOTATION	DESCRIPTION
base prefixes	The prefixes **%**, **#**, and **$** specify the numerical base of the value that follows: **%***num* specifies an octal number. **#***num* specifies a decimal number. **$***num* specifies a hexadecimal number. When no base is specified, decimal is assumed.
Bit (*bit:length*)	When a parameter contains more than one piece of data within its bit field, the different data fields are described in the format Bit (*bit:length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, `Bits (13:3)` indicates bits 13, 14, and 15: 
`computer font`	Denotes information displayed by the computer (for example, `login`:), file names (for example, `/usr/include/stdio.h`), and command names (for example, `vi`).
`underlining`	Denotes text you must type explicitly: `cc -c prog.c -lm`

Contents

Figures

Tables

1

Introduction to HP C

HP C is Hewlett-Packard's version of the C programming language that is implemented on HP 9000 workstations and servers. HP C/HP-UX is highly compatible with the C compiler implemented on the HP 9000 Series 300/400 and CCS/C, Corporate Computer Systems C compiler for the HP 3000. Some system and hardware-specific differences do exist. These are documented in the HP C Reference Manual for your system. Also, Chapter 2 of this manual, "Storage and Alignment Comparisons," provides system-specific information.

HP C Online Help

Online help for HP C is available for HP 9000 workstation and server users. The online help can be accessed from an X Windows display device. Several methods of invoking the HP C online help are listed below. Note that error messages are documented in the online help.

Accessing HP C Help with the +help Option

You may access HP C online help with the command line:

```
cc +help
```

Accessing HP C Help with the Front Panel

To access HP C online help if HP C and the help system are installed on your workstation:

1. Click on the `?` icon on the front panel.
2. The "Welcome to Help Manager" menu appears. Click on the HP C icon.

Accessing HP C Help with the dthelpview Command

If HP C is installed on another system or you are not running the help system, enter the following command from the system where HP C is installed:

```
/usr/dt/bin/dthelpview -h c
```

2

Storage and Alignment Comparisons

This chapter focuses on the different ways that internal data storage is allocated on various platforms and discusses the `HP_ALIGN` pragma which you can use to overcome these differences.

The storage and alignment rules of HP C on HP 9000 workstations and servers are compared with those of other systems. Note that the storage and alignment rules on the HP 3000 Series 900 are the same as those on the HP 9000 workstations and servers.

Data storage refers to the size of data types. **Data alignment** refers to the way a system or language aligns data structures in memory. Data type alignment and storage differences can cause problems when moving data between systems that have different alignment and storage schemes. These differences become apparent when data within a structure is exchanged between systems using files or inter-process communication.

The storage and alignment rules for the following systems are compared:

- HP C on the HP 9000 workstations and servers
- HP C on the HP 9000 Series 300/400.
- HP Apollo Series 3000/4000.
- HP Apollo Series 10000.
- CCS/C on the HP 1000.
- VAX/VMS C.

Data Type Size and Alignments

This section discusses storage sizes and alignment modes for the HP 9000 and HP Apollo systems as well as the VAX/VMS C, CCS/1000, and CCS/C 3000.

In all, there are a total of seven possible alignment modes which can be grouped into five categories as described in Table 2-1.

Table 2-1. The Alignment Modes

Alignment Mode	Description
`HPUX_WORD`, `DOMAIN_WORD`	`HPUX_WORD` is the native alignment for HP 9000 Series 300 and 400. `DOMAIN_WORD` is the native alignment for HP Apollo Series 3000 and 4000. The most restricted alignment boundary for a structure member is 2 bytes.
`HPUX_NATURAL`, `DOMAIN_NATURAL`	`HPUX_NATURAL` is the native alignment for HP 9000 workstations and servers and HP 3000 Series 900 and, therefore, is the default alignment mode. `DOMAIN_NATURAL` is the native alignment for HP Apollo Series 10000. The alignment of a structure member is related to its size (except for long double and long pointers), and the most restricted alignment boundary is 8 bytes.
`HPUX_NATURAL_S500`	`HPUX_NATURAL_S500` is the native alignment for HP 9000 Series 500. The alignment of a structure member is related to its size, and the most restricted alignment boundary is 4 bytes.
`NATURAL`	`NATURAL` is an architecture-independent alignment mode for HP Series 300, 400, workstations and servers, and HP Apollo Series 3000, 4000, and 10000. In the `NATURAL` mode, alignment of a structure member is related to its size, the most restricted alignment boundary being 8 bytes. The difference between `HPUX_NATURAL` and `NATURAL` are a 1-byte versus 2-byte minimum structure alignment and size, and the bit-field rules. This alignment mode is recommended when portability is an issue, since this mode enables data to be shared among the greatest number of HP-UX and Domain (HP Apollo) systems.
`NOPADDING`	This mode does not arise from a particular architecture. The most restricted alignment is 1 byte. `NOPADDING` alignment causes all structure and union members and typedefs to be packed on a byte boundary, and ensures that there will be no full byte padding inside the structure. Bit-field members either are byte-aligned or aligned immediately following a previous bit-field member, except in rare cases described in the section "Alignments of Bit-Fields" below.

Note With the exception of bit-fields, `DOMAIN_WORD` structure alignment is the same as `HPUX_WORD` structure alignment, and `DOMAIN_NATURAL` structure alignment is the same as `HP_NATURAL` structure alignment.

The alignment modes listed above can be controlled using the `HP_ALIGN` compiler pragma. See the section titled "The HP_ALIGN Pragma" in this chapter for a detailed description of this pragma. The `NATURAL` alignment mode should be used whenever possible. This mode enables data to be shared among the greatest number of HP-UX and Domain (HP Apollo) systems.

Alignment Rules

This discussion of alignment rules divides them into sections on scalar types, arrays, structures and unions, bit-fields, and typedefs.

Alignment of Scalar Types

Scalar types are integral types, floating types, and pointer types. Alignment of scalar types that are not part of a structure, union, or typedef declaration are not affected by the alignment mode. Therefore, they are aligned the same way in all alignment modes.

Table 2-2. Aligning Scalar Types

Data Type	Size (bytes)	Alignment (bytes)
`char, signed char, unsigned char, char enum`	1	1
`short, unsigned short, signed short, short enum`	2	2
`int, signed int, unsigned int, int enum`	4	4
`long, signed long, unsigned long, long enum`	4	4
`enum`	4	4
`long long`	8	8
`pointer`	4	4
`long pointer`	8	4
`float`	4	4
`double`	8	8
`long double`	16*	8

*8 bytes on `DOMAIN`

Note

Except for the `HPUX_NATURAL` and `DOMAIN_NATURAL` modes, the alignment of scalar types inside a structure or union may differ. (See the section "Alignment of Structures and Unions" below.) Also, a type that is defined via a typedef to any of the scalar types below may have a different alignment. (See the section "Alignment of Typedefs.")

Alignment of Arrays

An array is aligned according to its element type. For example, a `double` array is aligned on an 8-byte boundary; and a `float` array within a struct is aligned on a 4-byte boundary.

Alignment of array elements is not affected by the alignment mode, unless the array itself is a member of a structure or union. An array that is a member of a structure or union is aligned according to the rules for structure or union member alignment (see the section "Alignment of Structures and Unions" below for more information.)

An array's size is computed as:

(size of array element type) × (number of elements)

For instance, the array declared below is 400 bytes (4 × 100) long:

```
int arr[100];
```

The size of the array element type is 4 bytes and the number of elements is 100.

Alignment of Structures and Unions

In a structure, each member is allocated sequentially at the next alignment boundary corresponding to its type. Therefore, the structure might be padded internally if its members' types have different alignment requirements. In a union, all members are allocated starting at the same memory location. Both structures and unions can have padding at the end, in order to make the size a multiple of the alignment.

Note These rules are *not* true if the member type has been previously declared under another alignment mode. The member type will retain its original alignment, overriding other modes in effect. See the section "The HP_ALIGN Pragma" below for information on controlling alignment of structures and unions.

Table 2-3 lists the alignments for structure and union members.

Table 2-3. Aligning Structure or Union Members

Data Type	Size (bytes)	HPUX_WORD DOMAIN_WORD	HPUX_NATURAL DOMAIN_NATURAL	HPUX_ NATURAL_S500	NATURAL
char, signed char, unsigned char, char enum	1	1	1	1	1
short, unsigned short, signed short, short enum	2	2	2	2	2
int, signed int, unsigned int, int enum	4	2	4	4	4
long, signed long, unsigned long, long enum	4	2	4	4	4
enum	4	2	4	4	4
long long	8	2	8	4	8
pointer	4	2	4	4	4
long pointer	8	2	4	4	4
float	4	2	4	4	4
double	8	2	8	4	8
long double	16	2	8	4	8
arrays	Follows alignment of array type inside a structure or union.				
struct, union	Follows alignment of its most restricted member.				

Note In NOPADDING alignment mode, the alignment boundary is 1 byte in all cases except where bitfields are used.

HPUX_WORD/DOMAIN_WORD Alignments

For `HPUX_WORD` and `DOMAIN_WORD` alignments, all structure and union types are 2-byte aligned. Member types larger than 2 bytes are aligned on a 2-byte

boundary. Padding is performed as necessary to reach a resulting structure or union size which is a multiple of 2 bytes.

For example:

```
struct st {
   char c;
   long l;
   char d;
   short b;
   int i[2];
} s;
```

Compiling with the `+m` option to show the offsets of the identifiers, you will get the following output. Offsets are given as "byte-offset" @ "bit-offset" in hexadecimal.

```
Identifier          Class       Type              Address
----------          -----       -----------       -----------
s                     ext def     struct st
  c                   member      char              0x0 @ 0x0
  l                   member      long int          0x2 @ 0x0
  d                   member      char              0x6 @ 0x0
  b                   member      short int         0x8 @ 0x0
  i                   member      ints [2]          0xa @ 0x0
```

The resulting size of the structure is 18 bytes, with the alignment of 2 bytes, as illustrated in Figure 2-1

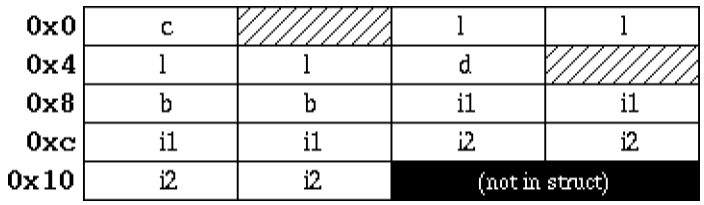


Figure 2-1. Example of HPUX_WORD/DOMAIN_WORD Alignment for Structure s

HPUX_NATURAL/DOMAIN_NATURAL Alignments

For `HPUX_NATURAL` and `DOMAIN_NATURAL` alignments, the alignment of structure and union types is the same as the strictest alignment of any member. Therefore, they may be aligned on 1-, 2-, 4-, or 8-byte boundaries. Padding is performed as necessary so that the size of the object is a multiple of the alignment size.

For example, the declaration shown in the previous section will now be aligned:

```
Identifier          Class       Type              Address
----------          --------    -----------       -----------
s                     ext def     struct st
  c                   member      char              0x0 @ 0x0
  l                   member      long int          0x4 @ 0x0
  d                   member      char              0x8 @ 0x0
  b                   member      short int         0xa @ 0x0
  i                   member      ints [2]          0xc @ 0x0
```

In this case, the size of the structure is 20 bytes, and the entire structure is aligned on a 4-byte boundary since the strictest alignment is 4 (from the `int` and `long` types), as illustrated in Figure 2-2.

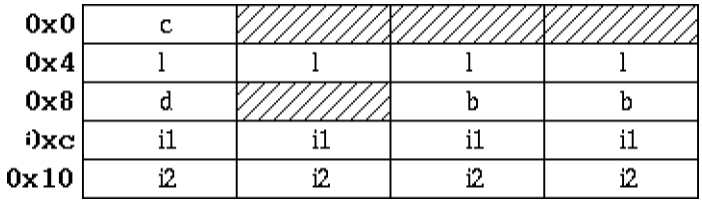


Figure 2-2.
Example of HPUX_NATURAL/DOMAIN_NATURAL Alignment for Structure s

HPUX_NATURAL_S500 Alignments

For `HPUX_NATURAL_S500` alignments, series 500 computers align structures on 2- or 4-byte boundaries, according to the strictest alignment of its members. As with the other alignment modes, padding is done to a multiple of the alignment size.

For example, the following code:

```
struct {
   char c;
   double d;
} s1;
```

compiled with the `+m` option produces:

```
Identifier          Class       Type              Address
----------          -------     -------           -----------
s1                    ext def     struct
  c                   member      char              0x0 @ 0x0
  d                   member      double            0x4 @ 0x0
```

The entire structure is 4-byte aligned, with a resulting size of 12 bytes.

NATURAL Alignments

For `NATURAL` alignments, structures and unions are aligned on 2-, 4-, or 8-byte boundaries, according to the strictest alignment of its members. Padding is done to a multiple of the alignment size.

NOPADDING Alignments

For `NOPADDING` alignments, structure or union members are byte aligned; therefore, struct and union types are byte aligned. This alignment mode does not cause compressed packing where there are zero bits of padding. It only ensures that there will be no full bytes of padding in the structure or union, unless bit-fields are used. There may be bit paddings or even a full byte of padding between members if there are bit-fields. Refer to the section "Alignment of Bit-Fields" for more information.

Consider the following code fragment:

```
#pragma HP_ALIGN NOPADDING
typedef struct s {
   char c;
   short s;
} s1;

s1 arr[4];
```

The size of `s1` is 3 bytes, with 1-byte alignment. Therefore, the size of `arr` is 12 bytes, with 1-byte alignment. There is no padding between the individual array elements; they are all packed on a byte boundary (see Figure 2-3).

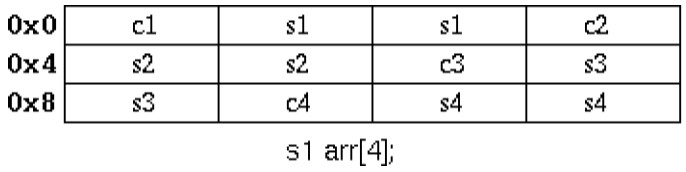


Figure 2-3. Example of NOPADDING Alignment for Structure s1

Note that if a member of a structure or union has been declared previously under a different alignment mode, it will retain its original alignment which may not be byte alignment. The `NOPADDING` alignment will not override the alignment of the member, so there may be some padding done within the structure, and the structure may be greater than byte aligned.

Refer to the section "Aligning Structures Between Architectures" below for examples on on structure alignment for different systems.

Alignment of Bit-Fields

The alignment modes for bit-fields are grouped differently than they are for the other types. The three groups are:

- `HPUX_NATURAL/HPUX_NATURAL_S500`
- `DOMAIN_WORD/DOMAIN_NATURAL/NATURAL/NOPADDING`
- `HPUX_WORD` (combination of the previous two)

HPUX_NATURAL/HPUX_NATURAL_S500 Alignments

For `HPUX_NATURAL` and `HPUX_NATURAL_S500` alignments, no bit-field can cross a "natural" boundary. A bit-field that immediately follows another bit-field is packed into adjacent bits, unless the second bit-field crosses a natural boundary according to its type. For example:

```
struct {
  int   a:5;
```

```
        int   b:15;
        int   c:17;
        char   :0;
        char  d:5;
        char  e:5;
      } foo;
```

when compiled with the `+m` option produces:

```
    Identifier          Class       Type             Address
    ----------          -----       ------           -----------
    foo                 ext def     struct
      a                   member      int              0x0 @ 0x0
      b                   member      int              0x0 @ 0x5
      c                   member      int              0x4 @ 0x0
<NULL_SYMBOL>             member      char             0x7 @ 0x0
      d                   member      char             0x7 @ 0x0
      e                   member      char             0x8 @ 0x0
```

The size of the structure is 12 bytes, with 4-byte alignment as illustrated in Figure 2-4.

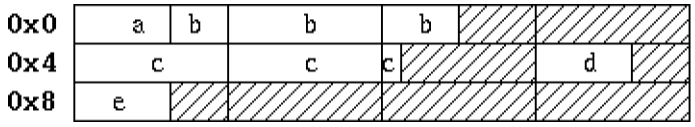


Figure 2-4.
Example of HPUX_NATURAL/HPUX_NATURAL_S500 Alignment for Structure foo

Since `b` (being an int type) does not cross any word boundaries, `a` and `b` are adjacent. `c` starts on the next word because it would cross a word boundary if it started right after `b`. The zero length bit-field forces no further bit-field to be placed between the previous bit-field, if any, and the next boundary described by the zero-length bit field's type. Thus, if we are at bit 5 and see a zero length bit-field of type int, then the next member will start at the next word boundary (bits 5-31 will be empty). However, if we are at bit 5 and see a zero length bit-field of type char, then the next member will start at least at

the next byte (bits 5-7 will be empty), depending on whether the next member can start at a byte-boundary.

DOMAIN_WORD/DOMAIN_NATURAL/NATURAL and NOPADDING Alignments

For `DOMAIN_WORD`, `DOMAIN_NATURAL`, `NATURAL`, and `NOPADDING` alignments:

- All integral types are treated identically; that is, the packing for `char a:17` (this is legal) is the same as for `int a:17`.
- Bit-fields can cross "natural" boundaries, unlike for HPUX_NATURAL. That is, for `int a:30; int b:7;`, `b` will start at bit 30.
- No bit-field can cross more than one 2-byte boundary. Thus, for `int a:14; int b:18;`, `b` will start at bit 16. If it started at bit 14, it would illegally cross both the 2- and 4-byte boundaries.
- The use of any type and size of bit-field alone will only cause the entire structure to have 2-byte alignment (1-byte for NOPADDING).

Note `NOPADDING` of bit-fields follows the `DOMAIN` alignment scheme. This may result in a full byte of padding between two bit-fields.

For example:

```
struct {
   char c;
   int i:31;  <-- At offset 2 bytes.
} bar;
```

The above structure `bar` will align the bit-field at offset 2 bytes, so that there is a full byte of padding between `c` and `i`, even with `NOPADDING` alignment mode (see Figure 2-5.)

Figure 2-5. Example of NATURAL Alignment for Structure bar

HPUX_WORD Alignments

For HPUX_WORD alignments:

- Alignment for char and short bit-fields is identical to that of HPUX_NATURAL.
- Alignment for any other bit-fields (int, long long, enum, for example) is identical to DOMAIN bit-field alignment.

Note that alignment of a char or short bit-field may not be the same as alignment of a char or short enum bit-field under the same circumstances.

For example:

```
#pragma HP_ALIGN HPUX_WORD

char enum b {a};
struct s {
   int           int_bit :30;
   char          char_bit :5;
};
struct t {
   int           int_bit :30;
   char enum b   char_enum_bit: 5;
};

int main()
{
   struct s basic_str;
   struct t enum_str;
}
```

Compilation with the +m option gives the following map:

```
Identifier          Class       Type             Address
----------          -----       --------         -----------
basic_str           auto        struct s         SP-48
  int_bit             member      int              0x0 @ 0x0
  char_bit            member      char             0x4 @ 0x0
enum_str            auto        struct t         SP-42
  int_bit             member      int              0x0 @ 0x0
  char_enum_bit       member      enum             0x3 @ 0x6
```

Both structures have a resulting size of 6 bytes, with 2-byte alignment as shown in Figure 2-6.

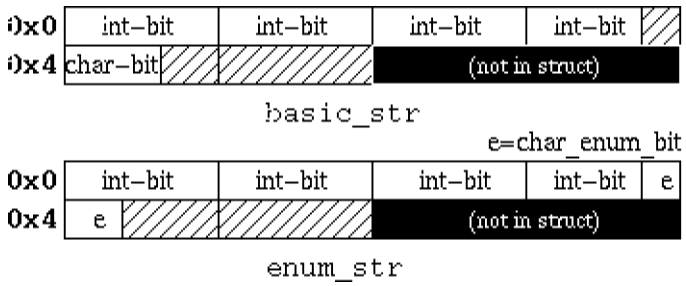


Figure 2-6. Example of Structures basic_str and enum_str

Notice that char_bit follows the `HPUX_NATURAL` alignment scheme, but char_enum_bit follows the `DOMAIN_WORD` alignment scheme, even though the length of their bit-field types are equivalent.

Alignment of Typedefs

Alignment for typedefs is slightly different than alignment for structures. Within a structure, the member itself is affected by the alignment mode. However, with a typedef, the alignment of the type that the typedef name is derived from is affected, not the typedef name itself. The typedef name is then associated with the derived type.

When a typedef is seen, a new type is created by:

1. Taking the innermost type from which the typedef name is derived (which may be another derived type).
2. Setting its alignment to what it would be if it were used inside a structure or union declaration.
3. Creating a derived type from that new type, associating it with the typedef name.

Let us start with a simple example of a declaration under `NOPADDING`:

```
typedef int my_int;
```

Since an int will be 1-byte aligned inside a structure under `NOPADDING`, `my_int` will be 1-byte aligned.

Consider a pointer typedef with `NOPADDING` alignment:

```
typedef int **my_double_ptr;
```

`my_double_ptr` is derived from an integer type; therefore, a new integer type of 1-byte alignment is created. `my_double_ptr` is defined to be a 4-byte aligned pointer to another 4-byte aligned pointer which points to a byte-aligned int.

Consider another example, this time with `HPUX_WORD`:

```
typedef int *my_ptr;
typedef my_ptr *my_double_ptr;
```

In the first typedef, `my_ptr` will be a 4-byte aligned pointer to a 2-byte aligned int. The second typedef will create another type for `my_ptr` which is now 2-byte aligned, since `my_double_ptr` is derived from `my_ptr`. So `my_double_ptr` is a 4-byte aligned pointer to a 2-byte aligned pointer which points to a 2-byte aligned int.

Similar declarations inside a structure will not have the same resulting alignment. Consider the following declaration:

```
#pragma HP_ALIGN NOPADDING

typedef int **my_double_ptr;

struct s {
   int **p;
};
```

In the above example, `my_double_ptr` is a 4-byte aligned pointer type pointing to another 4-byte aligned pointer which points to a 1-byte aligned int. However, `struct s` member `p` is a 1-byte aligned pointer which points to a 4-byte aligned pointer which points to 4-byte aligned int. Inside a structure, the member itself is affected by the alignment mode. However, with a typedef, the typedef name is not directly affected. The innermost type from which the typedef name is derived is affected by the alignment mode.

The HP_ALIGN Pragma

The HP_ALIGN pragma controls data storage allocation and alignment of structures, unions, and type definitions, using typedefs. It enables you to control the alignment mode when allocating storage space for data. It is especially important when used to control the allocation of binary data that is transmitted among machines having different hardware architectures.

The HP_ALIGN pragma takes a parameter indicating which alignment mode to use. Not all modes are available on all HP platforms; the NATURAL alignment mode is the most widely available on HP-UX. This mode is the recommended standard.

The syntax for the HP_ALIGN pragma is:

```
#pragma HP_ALIGN align_mode [PUSH]

#pragma HP_ALIGN POP
```

where *align_mode* is one of the following:

HPUX_WORD	This is the Series 300/400 default alignment mode.
HPUX_NATURAL_S500	This is the Series 500 default alignment mode.
HPUX_NATURAL	This is the HP 9000 workstations and servers and HP 3000 Series 900 systems default alignment mode.
NATURAL	This mode provides a consistent alignment scheme across HP architectures.
DOMAIN_WORD	This is the default word alignment mode on HP Apollo architecture.
DOMAIN_NATURAL	This is the default natural alignment mode on HP Apollo architecture.
NOPADDING	This causes all structures and union members that are not bit-fields to be packed on a byte boundary. It does not cause compressed packing where there are zero bits of padding. It only insures that there will be no full bytes of padding in the structure or union.

Note The above alignment modes are only available on HP-UX systems.

The HP_ALIGN pragma affects struct and union definitions as well as typedef declarations. It causes data objects that are later declared using these types to have the size and alignment as specified by the pragma.

The alignment pragma in effect at the time of data type declaration has significance. The alignment pragma in effect at the time of data type declaration has precedence over the alignment pragma in effect when space for a data object of the previously declared type is allocated.

Using the HP_ALIGN Pragma

The HP_ALIGN pragma allows you to control data storage allocation and alignment of structures, unions, and typedefs.

Note The basic scalar types, array types, enumeration types, and pointer types are *not* affected by the HP_ALIGN pragma. The pragma only affects struct or union types and typedefs - No other types are affected by specifying the HP_ALIGN pragma.

The HP_ALIGN pragma takes a parameter that specifies the alignment mode for example:

```
#pragma HP_ALIGN HPUX_NATURAL
```

There is also an optional parameter PUSH, which saves the current alignment mode before setting the specified mode as the new alignment mode. For example, in the following sequence:

```
#pragma HP_ALIGN NOPADDING PUSH
    /* decls following */
```

the current alignment mode is saved on the stack. It is then set to the new alignment mode, NOPADDING.

The PUSHed alignment mode can be retrieved later by doing a

```
#pragma HP_ALIGN POP
```

If the last alignment mode PUSHed on the stack was NOPADDING, the current alignment mode would now be NOPADDING.

Problems Sometimes Encountered with the HP_ALIGN Pragma

If only one alignment mode is used throughout the entire file, this pragma is straightforward to use and to understand. However, when a different mode is introduced in the middle of the file, you should be aware of its implications and effects.

The key to understanding HP_ALIGN is the following concept: typedefs and struct or union types retain their original alignment mode throughout the entire file. Therefore, when a type with one alignment is used in a different alignment mode, it will still keep its original alignment.

This feature may lead to confusion when you have a typedef, structure or union of one alignment nested inside a typedef, structure or union of another alignment.

Here are some examples of the most common misunderstandings.

Example 1: Using Typedefs. The alignment pragma will affect typedef, struct, and union types. Therefore, in the following declaration:

```
#pragma HP_ALIGN HPUX_WORD
typedef int int32;
```

int32 is not equivalent to int. To illustrate:

```
#pragma HP_ALIGN HPUX_WORD

typedef int  int32;

void routine (int *x);

int main()
{
   int   *ok;
   int32 *bad;

   routine(ok);
   routine(bad);  /* warning */
```

```
}
```

Compiling this with `-Aa -c` will give two warnings:

```
warning 604: Pointers are not assignment-compatible.
warning 563: Argument #1 is not the correct type.
```

These warnings occur because the actual pointer value of `bad` may not be as strictly aligned as the pointer type routine expects. This may lead to run-time bus errors in the called function if it dereferences the misaligned pointer.

Example 2: Using Combination of Different Alignment Modes. In the `WORD` alignment modes, the members of a structure whose sizes are larger than 2 bytes are aligned on a 2-byte boundary. However, this is only true if those member types are scalar or have been previously declared under the same alignment mode. If the member type is a typedef, struct, or union type which has been declared previously under a different alignment mode, it will retain its original alignment, regardless of current alignment mode in effect. For example:

```
typedef int my_int;

#pragma HP_ALIGN HPUX_WORD

struct st {
   char c;
   my_int i;
 };

int main()
{
   char c;
   struct st foo;
}
```

Although the size of `my_int` is greater than 2 bytes, because it was declared previously under `HPUX_NATURAL` with the alignment of 4 bytes it will be aligned on a 4-byte boundary, causing the entire struct `st` to be aligned on a 4-byte boundary. Compiling with the `+m` option to show the offsets of the identifiers (offsets given as "byte-offset @ bit-offset" in hexadecimal), you will get the following output:

```
                                main

Identifier          Class       Type              Address
----------          -----       ---------         -----------
c                   auto        char              SP-48
foo                 auto        struct st         SP-44
  c                   member      char              0x0 @ 0x0
  i                   member      int               0x4 @ 0x0
```

The resulting size of `foo` is 8 bytes, with 4-byte alignment.

If you change the type of member `i` in `struct st` to be a simple int type, then you will get the following result:

```
                                main

Identifier          Class       Type              Address
----------          -----       ------            -----------
c                   auto        char              SP-40
foo                 auto        struct st         SP-38
  c                   member      char              0x0 @ 0x0
  i                   member      int               0x2 @ 0x0
```

This time, the resulting size of `foo` is 6 bytes, with 2-byte alignment.

Example 3: Incorrect Use of Typedefs and Alignments. Often, you might mix typedefs and alignments without being aware of the actual alignment of the data types.

What may appear to be correct usages of these data types may turn out to be causes for misaligned pointers and run-time bus errors, among other things. For example, consider the following program.

```
<my_include.h>
 typedef unsigned short ushort;
 extern int get_index(void);
 extern ushort get_value(void);

<my_prog.c>
 #include "my_include.h"

 #pragma HP_ALIGN NOPADDING PUSH
 struct s {
    ushort member1;
    ushort member2;
 };
 #pragma HP_ALIGN POP

 char myBuffer[100];

 int main()
 {
    struct s *my_struct;
    int index = get_index();
    int value = get_value();
    int not_done = 1;

    while (not_done)  {
       my_struct = (struct s*)&myBuffer[index];
       my_struct->member1 = value;
     .
     .
     .

    }
 }
```

This code is not written safely. Although `struct s` is declared under `NOPADDING` alignment mode, it has 2-byte alignment due to the typedef for `ushort`. However, a pointer to `struct s` can be assigned an address that can point to anywhere in the char array (including odd addresses). If the function `get_index` always returns an even number, you will not run into any problems,

because it will always be 2-byte aligned. However, if the index happens to be an odd number, `&myBuffer[index]` will be an odd address. Dereferencing that pointer to store into a 2-byte aligned member will result in a run-time bus error.

Below are some examples of what you can do to avoid such behavior.

- Compile with `+u1` option, which forces all pointer dereferences to assume that data is aligned on 1-byte boundaries. However, this will have a negative impact on performance.
- Put the typedef inside the `NOPADDING` alignment. However, if you use `ushort` in contexts where it must have 2-byte alignment, this may not be what you want.
- Declare `struct s` with the basic type `unsigned short` rather than the typedef `ushort`.
- Make sure that the pointer will always be 2-byte aligned by returning an even index into the char array.
- Declare another typedef for `ushort` under the `NOPADDING` alignment:

  ```
  typedef ushort ushort_1
  ```

  and use the new type `ushort_1` inside `struct s`.

As mentioned above, the `HP_ALIGN` pragma must have a global scope; it must be outside of any function or enclosing structure or union. For example, suppose you have the following sequence of pragmas:

```
#pragma HP_ALIGN HPUX_WORD PUSH

struct string_1 {
        char *c_string;
        int  counter;
       };

#pragma HP_ALIGN HPUX_NATURAL PUSH

struct car {
        long double  car_speed;
        char         *car_type;
```

```
        };

#pragma HP_ALIGN POP

struct bus {
         int bus_number;
         char bus_color;
        };

#pragma HP_ALIGN POP
```

Variables declared of type `struct string_1`, are aligned according to the `HPUX_WORD` alignment mode. Variables declared of type `struct car`, are aligned according to the `HPUX_NATURAL` alignment mode. Variables declared of type `struct bus` are aligned according to `HPUX_WORD`.

Accessing Non-Natively Aligned Data with Pointers

Be careful when using pointers to access non-natively aligned data types within structures and unions. Alignment information is significant, as pointers may be dereferenced with either 8-bit, 16-bit, or 32-bit machine instructions. Dereferencing a pointer with an incompatible machine instruction usually results in a run-time error.

HP C permanently changes the size and alignment information of typedefs defined within the scope of an `HP_ALIGN` pragma. It makes data objects, such as pointers, declared by using typedefs, compatible with similar objects defined within the scope of the pragma.

For example, a pointer to an integer type declared with a typedef that is affected by the `HP_ALIGN` pragma will be dereferenced safely when it points to an integer object whose alignment is the same as that specified in the pragma.

The typedef alignment information is persistent outside the scope of the `HP_ALIGN` pragma. An object declared with a typedef will have the same storage and alignment as all other objects declared with the same typedef, regardless of the location of other `HP_ALIGN` pragma statements in the program.

There is a slight performance penalty for using non-native data alignments. The compiler generates slower but safe code for dereferencing non-natively aligned data. It generates more efficient code for natively aligned data.

The following program generates a run-time error because a pointer that expects word-aligned data is used to access a half-word aligned item:

```
#pragma HP_ALIGN HPUX_WORD

struct t1 { char a; int b;} non_native_rec;

#pragma HP_ALIGN POP

main ()
{
        int i;
        int *p = &non_native_rec.b;
        i  = *p;  /* assignment causes run-time bus error */
}
```

The following program works as expected because the pointer has the same alignment as the structure:

```
#pragma HP_ALIGN HPUX_WORD

struct t1 { char a; int b;} non_native_rec;
typedef int non_native_int;

#pragma HP_ALIGN POP

main ()
{
        int i;
        non_native_int *p = &non_native_rec.b;
        i = *p;
}
```

An alternative to using the `HP_ALIGN` pragma and typedefs to control non-natively aligned pointers is to use the `+u`*bytes* compiler option of HP C/HP-UX. The `+u`*bytes* forces all pointer dereferences to assume that data is aligned on 8-bit, 16-bit, or 32-bit addresses. The value of *bytes* can be 1

(8-bit), 2 (16-bit), or 4 (32-bit). This option can be used when accessing non-natively aligned data with pointers that would otherwise be natively aligned. This option can be useful with code that generates the compiler warning message

```
#565 - "address operator applied to non natively aligned member."
```

and aborts with a run-time error.

The `+u`*bytes* option affects all pointer dereferences within the source file. It can have a noticeable, negative impact on performance.

Note The HP C/iX implementation of the `+u` option omits the *bytes* parameter.

Defining Platform Independent Data Structures

One way to avoid trouble caused by differences in data alignment is to define structures so they are aligned the same on different systems. To do this, use **padding bytes**—that is, dummy variables to align fields the same way on different architectures.

For example, use:

```
struct {
   char c1;
   char dum1;
   char dum2;
   char dum3;
   int  i1;
};
```

instead of:

```
struct {
   char c1;
   int i1;
};
```

Aligning Structures Between Architectures

Differences in data type alignment can cause problems when porting code or data between systems that have different alignment schemes. For example, if you write a C program on the Series 300/400 that writes records to a file, then read the file using the same program on HP 9000 workstations and servers, it may not work properly because the data may fall on different byte boundaries within the file because of alignment differences.

Three methods can be used for aligning data within structures so that it can be shared between different architectures.

- Use only ASCII formatted data. This is the safest method, but has negative performance and space implications.
- Use the `HP_ALIGN` pragma, which is available on most HP-UX HP C compilers. It forces a particular alignment scheme, regardless of the architecture on which it is used. See "The HP_ALIGN Pragma" section for a detailed description of this pragma.
- Define platform independent data structures using explicit padding.

To illustrate the portability problem raised by different alignments, consider the following example.

```
#include <stdio.h>
struct char_int
  {
    char field1;
    int  field2;
  };
main (void)
  {
  FILE *fp;
  struct char_int s;
         ⋮
  fp = fopen("myfile", "w");
  fwrite(&s, sizeof(s), 1, fp);
         ⋮
  }
```

The alignment for the struct that is written to `myfile` in the above example is shown in the following diagram.

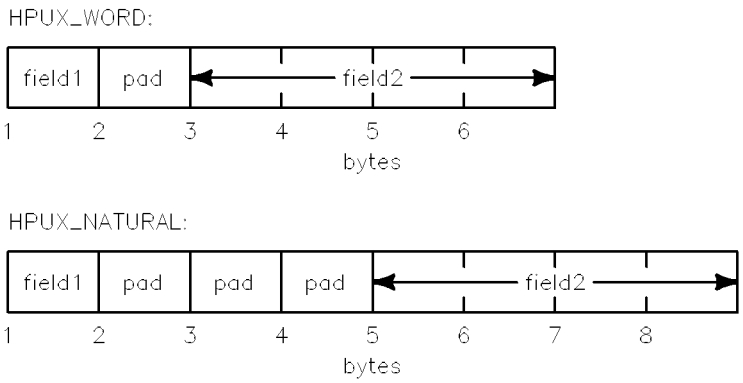


Figure 2-7.
Comparison of HPUX_WORD and HPUX_NATURAL Byte Alignments

In the `HPUX_WORD` alignment mode, six bytes are written to `myfile`. The integer `field2` begins on the third byte. In the `HPUX_NATURAL` alignment mode, eight bytes are written to `myfile`. The integer `field2` begins on the fifth byte.

Examples of Structure Alignment on Different Systems

The code fragment in Figure 2-8 can be used to illustrate the alignment on various systems.

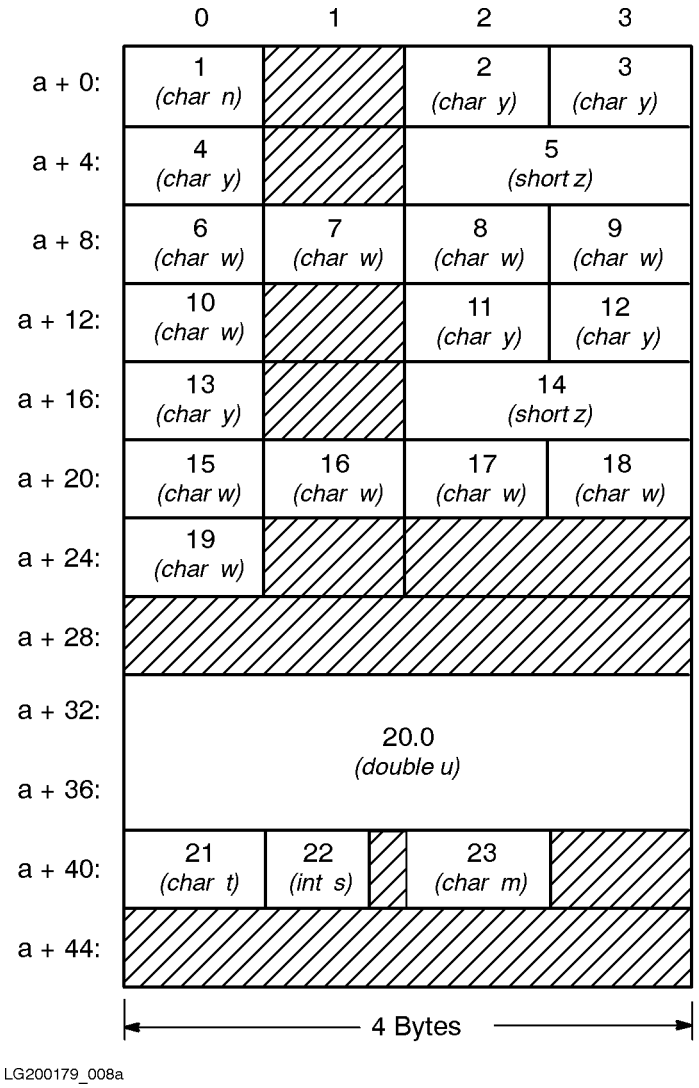
```
struct x {
   char y[3];
   short z;
   char w[5];
};

struct q {
   char n;
   struct x v[2];
   double u;
   char t;
   int s:6;
   char m;
} a = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
       20.0,21,22,23};
```

Figure 2-8. Code Fragment for Comparing Storage and Alignment

HP C/HP-UX 9000 Workstations and Servers and HP C/iX

Figure 2-9 shows how the data in Figure 2-8 is stored in memory when using HP C on the HP 9000 workstations and servers and HP 3000 Series 900. The values are shown above the variable names. Shaded cells indicate padding bytes.



LG200179_008a

Figure 2-9.
Storage with HP C on the HP 9000 workstations and servers and HP 3000 Series 900

The `struct q` is aligned on an 8-byte boundary because the most restrictive data type within the structure is the `double u`.

Table 2-4 shows the padding for the example code fragment:

Table 2-4.
Padding on HP 9000 Workstations and Servers and HP 3000 Series 900

Padding Location	Reason for Padding
`a+1`	The most restrictive type of the structure `x` is `short`; therefore, the structure is 2-byte aligned.
`a+5`	Aligns the `short z` on a 2-byte boundary.
`a+13`	Fills out the `struct x` to a 2-byte boundary.
`a+17`	Aligns the `short z` on a 2-byte boundary.
`a+25`	Fills out the structure to a 2-byte boundary.
`a+26` through `a+31`	Aligns the `double u` on an 8-byte boundary. The bit-field `s` begins immediately after the previous item at `a+41`. Two bits of padding is necessary to align the next byte properly.
`a+43` through `a+47`	Fills out the `struct q` to an 8-byte boundary.

HP C on the Series 300/400

The differences between HP C on the HP 9000 Series 300/400 and HP C on the HP 9000 workstations and servers and HP 3000 Series 900 are:

- On the Series 300/400, a structure is aligned on a 2-byte boundary. On the HP 9000 workstations and servers and HP 3000 Series 900, it is aligned according to the most restrictive data type within the structure.
- On the Series 300/400, the double data type is 2-byte aligned within structures. It is 8-byte aligned on the HP 9000 workstations and servers and HP 3000 Series 900.
- On the Series 300/400, the long double, available in ANSI mode only, is 2-byte aligned within structures. The long double is 8-byte aligned on the HP 9000 workstations and servers and HP 3000 Series 900.

- On the Series 300/400, the enumerated data type is 2-byte aligned in a structure, array, or union. The enumerated type is always 4-byte aligned on the HP 9000 workstations and servers and HP 3000 Series 900, unless a sized enumeration is used.

When the sample code fragment is compiled and run, the data is stored as shown in Figure 2-10:

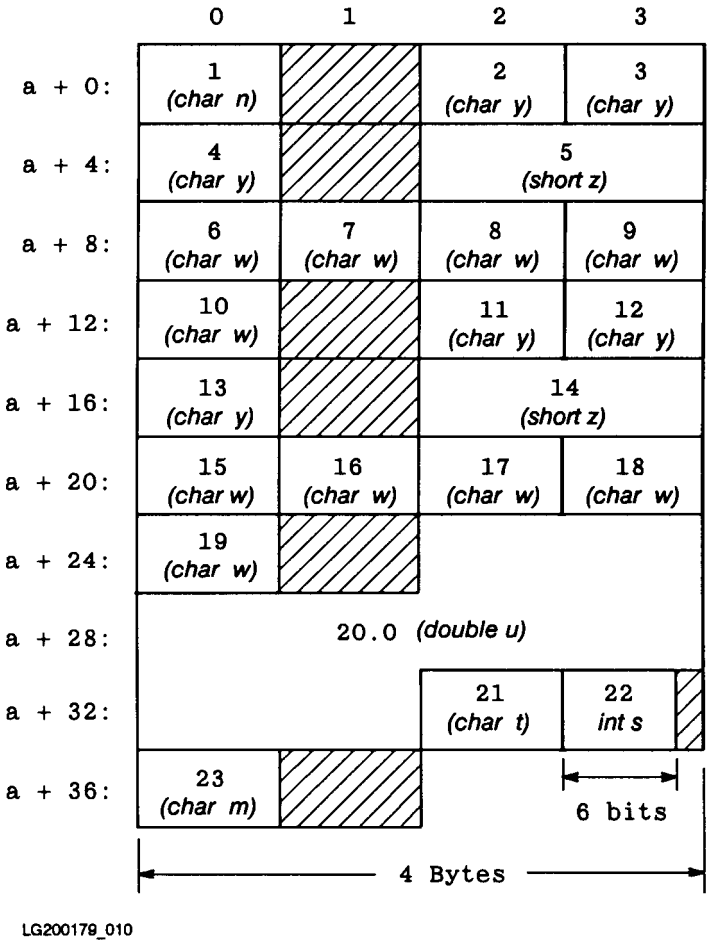


Figure 2-10. Storage with HP C on the HP 9000 Series 300/400

Table 2-5 shows the padding for the example code fragment:

Table 2-5. Padding on the HP 9000 Series 300/400

Padding Location	Reason For Padding
`a+1`	Within structures align `short` on a 2-byte boundary.
`a+5`	Aligns the `short z` on a 2-byte boundary.
`a+14`	Structures within structures are aligned on a 2-byte boundary.
`a+17`	Aligns the `short z` on a 2-byte boundary.
`a+25`	Doubles are 2-byte aligned within structures.
`a+37`	Pads `a` to a 2-byte boundary.

CCS/C on the HP 1000 and HP 3000

Figure 2-11 shows how the members of the structure defined in Figure 2-8 are aligned in memory when using CCS/C on the HP 1000 or HP 3000:

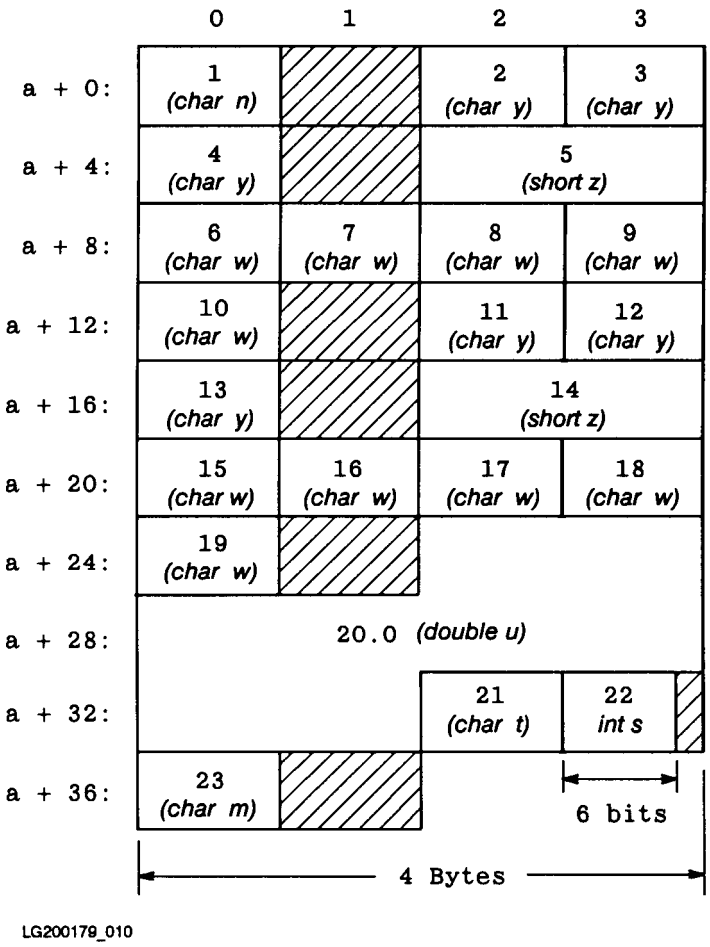


Figure 2-11. Storage with CCS/C

Note All data types and structures are 2-byte aligned when using CCS/C on the HP 1000 or HP 3000.

Table 2-6 shows the padding for the example code fragment:

Table 2-6. Padding with CCS/C

Padding Location	Reason for Padding
`a+1`	Aligns the structure on a 2-byte boundary.
`a+5`	Aligns the `short z` on a 2-byte boundary.
`a+13`	Fills out the `struct x` to a 2-byte boundary. (Aligns the character on a 2-byte boundary.)
`a+17`	Aligns the `short z` on a 2-byte boundary.
`a+25`	Fills out the structure to a 2-byte boundary and aligns the `double u` on a 2-byte boundary.
`a+37`	Pads `a` to a 2-byte boundary.

VAX/VMS C

The differences between HP C and VAX/VMS C are:

- In HP C workstations and servers, the `double` type is 8-byte aligned; in VAX/VMS C, the `double` type is 4-byte aligned.
- In HP C, bit-fields are packed from left to right. In VAX/VMS C, the fields are packed from right to left.
- HP C uses big-endian data storage with the most significant byte on the left. VAX/VMS C uses little-endian data storage with the most significant byte on the right. (See the `swab` function in the *HP-UX Reference* manual for information about converting from little-endian to big-endian.)

In VAX/VMS C, the data from the program in Figure 2-8 is stored as shown in Figure 2-12:

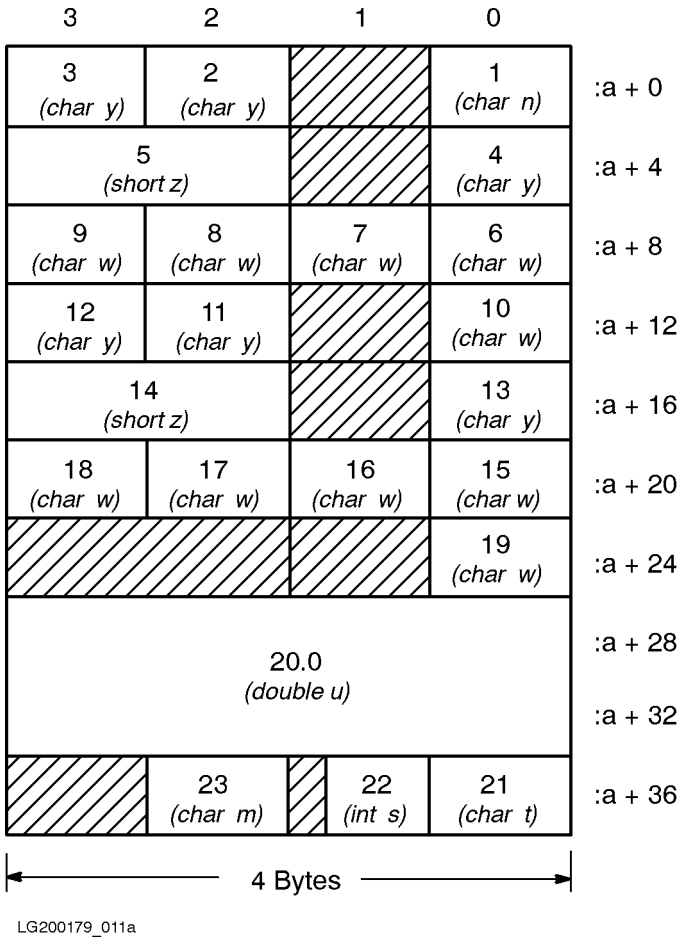


Figure 2-12. Storage on VAX/VMS C

Table 2-7 shows the padding for the example code fragment:

Table 2-7. Padding on VAX/VMS C

Padding Location	Reason for Padding
a+1	The most restrictive type of any `struct x` member is `short`; therefore, `struct x` is 2-byte aligned.
a+5	Aligns the `short z` on a 2-byte boundary.
a+13	Fills out the `struct x` to a 2-byte boundary.
a+17	Needed for alignment of the `short z`.
a+25 through a+27	Fills out the structure to a 2-byte boundary and aligns the `double u` on a 4-byte boundary.
a+37	Aligns the `char m` on a byte boundary.
a+39	Fills out the structure to a 4-byte boundary.

3

Calling Other Languages

This chapter describes how to call routines written in other HP languages from HP C programs.

Invoking routines or accessing data defined or declared in another programming language from HP C can be tricky. Here are some common problems:

- Mismatched data types for parameters and return values.
- Different language storage layouts for aggregates (arrays, records, variants, structures, unions, equivalences, and commons).
- Different formats for strings among HP C, HP Pascal, and HP FORTRAN 77.
- Different language values for true, false, eof, and nil.
- Different language bit level justification of objects smaller than 32 bits (right-justification or most significant bit or byte last versus left-justification or most significant bit or byte first).

The topics listed above are described in detail in this chapter. For additional information, refer to Chapter 2, "Storage and Alignment Comparisons," in this manual. Also, the following manuals have chapters on calling other languages:

- *HP Pascal/HP-UX Programmer's Guide*
- *HP FORTRAN/9000 Programmer's Guide*
- *COBOL/HP-UX Operating Manual*

Comparing HP C and HP Pascal

Table 3-1 summarizes the differences in storage allocation between HP C and HP Pascal. The footnote numbers refer to notes located in a section immediately following the table.

Table 3-1. HP C versus HP Pascal Storage Allocation

HP C Type	HP C Description	Corresponding HP Pascal Type	HP Pascal Description
`char, signed char`	1 byte, byte aligned	`--`	1 byte, byte aligned; Subrange: -128 .. 127
`unsigned char`	1 byte, byte aligned	`char`	1 byte, byte aligned; Subrange: 0 .. 255
`short`	2 bytes, 2-byte aligned	`shortint`	Subrange: -32768..32767
`unsigned short`	2 bytes, 2-byte aligned	`--`	Subrange: 0 .. 65535
`int`	4 bytes, 4-byte aligned	`integer`	4 bytes, 4-byte aligned; Subrange: -2147483648 .. 2147483647
`unsigned int`	4 bytes, 4-byte aligned	`--`	4 bytes, 4-byte aligned; Subrange: 0 .. 4294967295
`long`	4 bytes, 4-byte aligned	`integer`	Subrange: -2147483648 .. 2147483647
`unsigned long`	4 bytes, 4-byte aligned	`--`	4 bytes, 4-byte aligned; Subrange: 0 .. 4294967295
(See Note 1)	`--`	`longint`	8 bytes, 4-byte aligned
`float`	4 bytes, 4-byte aligned	`real`	4 bytes, 4-byte aligned

Table 3-1. HP C versus HP Pascal Storage Allocation (continued)

HP C Type	HP C Description	Corresponding HP Pascal Type	HP Pascal Description
`double`	8 bytes, 8-byte aligned	`longreal`	8 bytes, 8-byte aligned
`long double`	16 bytes, 16-byte aligned	`--`	`--`
`enum`	4 bytes, 4-byte aligned	enumeration or `integer` (*See Note 2*)	1 byte if fewer than 257 elements; 2 bytes if between 257 and 65536; otherwise, 4 bytes. 1, 2, or 4-byte aligned.
`char enum`	1 byte, 1-byte aligned	`--`	1 byte, 1-byte aligned, subrange: -128..127
`short enum`	2 bytes, 2-byte aligned	`short int`	subrange: -32768..32767
`int enum`	4 bytes, 4-byte aligned	`integer`	4 bytes, 4-byte aligned, subrange: -2,147,483,648.. 2,147,483,647
`long enum`	4 bytes, 4-byte aligned	`integer`	4 bytes, 4-byte aligned, subrange: -2,147,483,648.. 2,147,483,647
`array` [*n*] `of` *type*	Size is number of elements times element size. Align according to element type.	`ARRAY` [0 .. *n*-1] OF *type* (*See Note 3*)	Size is the number of elements times element size. Align according to element type.

Table 3-1. HP C versus HP Pascal Storage Allocation (continued)

HP C Type	HP C Description	Corresponding HP Pascal Type	HP Pascal Description
`array` [*n*] `of char`	[*n*] bytes, byte aligned	`PACKED ARRAY` [0 .. *n*-1] `OF CHAR` or not `PACKED` (*See Note 4*)	[*n*] bytes, byte aligned
`struct` (*See Note 5*)	Pascal string descriptors may be emulated using C structures, see the note for an example.	`STRING` [*n*]	Size 4+[*n*]+1 bytes, 4-byte aligned.
Pointer to string descriptor structure (*See Note 6*)	Pascal `VAR` parameters may be emulated using C pointers to string descriptor structures. (*See Note 6*).	`STRING`	--
`char *`	Pointer to a null terminated array of characters	pointer to character array	(*See Note 7*)
`struct`	Size of elements plus padding, aligned according to largest type	`record`	(*See Note 8*)
`union`	Size of elements plus padding, aligned according to largest type	(untagged) variant record (*See Note 9*)	(*See Note 8*)
signed bit-fields	--	`packed record` (*See Note 10*)	--
unsigned bit-fields	--	`packed record` (*See Note 11*)	--

Table 3-1. HP C versus HP Pascal Storage Allocation (continued)

HP C Type	HP C Description	Corresponding HP Pascal Type	HP Pascal Description
`void`	--	Used when calling an HP Pascal procedure (*See Note 12*)	--
pointer	4 bytes, 4-byte aligned	pointer to corresponding type	4 bytes, 4-byte aligned
long pointer	8 bytes, 8-byte aligned	`$ExtnAddr$` pointer or `$ExtnAddr$` VAR parameter	8 bytes, 4-byte aligned
`char`	1 byte, 1-byte aligned	`boolean` (*See Note 13*)	1 byte, 1 byte aligned
`void` function parameter	4 bytes, 4-byte aligned	`PROCEDURE` *parameter*	4 bytes, 4-byte aligned
function parameter	4 bytes, 4-byte aligned	`FUNCTION` *parameter*	4 bytes, 4-byte aligned
`struct` of 1-bit fields	(*See Note 14*)	`set`	--
--	Pascal files may be read by C programs with some effort. (*See Note 15*)	`file`	external record oriented file
pointer to `void` function	--	`procedure`	--
pointer to function	--	`function`	--

Notes on HP C and HP Pascal

1. The `longint` type in HP Pascal is a 64-bit signed integer. A corresponding HP C type could be any structure or array of 2 words; however, HP C cannot directly operate on such an object.

2. By default, HP C enumerations are allocated 4 bytes of storage, while HP Pascal enumerations use the following scheme:

   1 byte, if fewer than 257 elements.
   2 bytes, if between 257 and 65536 elements.
   4 bytes, otherwise.

   If the default enumeration specifier is modified with a `char` or `short` type specifier, 1 or 2 bytes of storage are allocated. See Table 3-1 for a description of the sized enumerated types.

   This is important if the items are packed. For example, a 25-element enumeration in HP Pascal can use 1 byte and be on a byte boundary, so you must use the HP C type `char` or a sized `enum` declaration `char enum`.

3. HP C always indexes arrays from zero, while HP Pascal arrays can have lower bounds of any user-defined scalar value. This is only important when passing an array using an index to subscript the array. When passing the subscript between HP C and HP Pascal, you must adjust the subscript accordingly. HP C always passes a pointer to the first element of an array. To pass an array by value, enclose the array in a `struct` and pass the `struct`.

4. HP C `char` arrays are packed one character per byte, as are HP Pascal arrays (even if PACKED is not used). HP Pascal permits certain string operations with a packed array of `char` when the lower bound is one.

5. The HP Pascal type `STRING` [*n*] uses a string descriptor that consists of the following: a word containing the current length of the string, *n* bytes for the characters, and an extra byte allocated by the HP Pascal compiler. Thus, the HP Pascal type `STRING[10]` corresponds to the following HP C structure:

```
typedef struct {
   int cur_len;                    /*  4 bytes  */
   char chars [10];                /* 10 bytes  */
```

```
    char extra_byte;                /*  1 byte   */
} STRING_10;
```

which is initialized like this:

```
STRING_10 this_string = {
   0,                                /* The current length */
   {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, /* The 10 bytes       */
   0                                 /* The null byte      */
};
```

Both the C structure and the Pascal string are 4-byte aligned.

6. HP Pascal also has a generic string type in which the maximum length is unknown at compile time. Objects of this type have the same structure as in Note 5 above; the objects are *only* used as VAR formal parameters.

7. A variable of this type is a pointer to a character array if the string is null-terminated; HP Pascal will not handle the null byte in any special way. An HP C parameter of type "pointer to `char`" corresponds to an HP Pascal VAR parameter of type "packed array of `char`." However, the type definition of that VAR parameter must have the bounds specified.

8. The size is equal to the size of all members plus any padding needed for the alignment. (See Chapter 2 for details on alignment.) The alignment is that of the member with the strictest alignment requirement.

9. A union corresponds directly to an untagged HP Pascal variant record. For example, the HP C union:

```
typedef union {
   int i;
   float r;
   unsigned char c;
} UNIONTYPE;
```

corresponds to:

```
TYPE
   UNIONTYPE = RECORD CASE INTEGER OF
      1 : (i : INTEGER);
      2 : (r : REAL);
      3 : (c : CHAR);
```

```
END;
```

The tagged HP Pascal variant record:

```
TYPE
   TAGGED_UNIONTYPE = RECORD CASE tag : INTEGER OF
      1 : (i : INTEGER);
      2 : (r : REAL);
END;
```

corresponds to this HP C structure:

```
typedef struct {
   int tag;
   union {
      int i;
      float r;
   };
} TAGGED_UNIONTYPE;
```

10. HP Pascal subranges with a negative value as their lower bound have enough bits allocated to contain the upper bound, with an extra bit for the sign. Thus, the HP C structure:

```
typedef struct {
   int b1 : 1;
   int b2 : 2;
   int b3 : 3;
   int b4 : 4;
   int b5 : 5;
   int b6 : 6;
   int b7 : 7;
} BITS;
```

corresponds to the following untagged HP Pascal record:

```
TYPE
   BITS = PACKED RECORD
      b1 : BOOLEAN;
      b2 : -2 ..  1;
      b3 : -4 ..  3;
      b4 : -8 ..  7;
```

```
      b5 : -16 .. 15;
      b6 : -32 .. 31;
      b7 : -64 .. 63;
   END;
```

11. Unsigned bit-fields map onto HP Pascal packed record fields whose types are the appropriate subranges. For example, the HP C structure:

```
typedef struct {
   unsigned int b1 : 1;
   unsigned int b2 : 2;
   unsigned int b3 : 3;
   unsigned int b4 : 4;
   unsigned int b5 : 5;
   unsigned int b6 : 6;
   unsigned int b7 : 7;
} BITS;
```

    corresponds to this untagged HP Pascal record:

```
TYPE
   BITS = PACKED RECORD
      b1 : 0 ..   1;
      b2 : 0 ..   3;
      b3 : 0 ..   7;
      b4 : 0 ..  15;
      b5 : 0 ..  31;
      b6 : 0 ..  63;
      b7 : 0 .. 127;
      END;
```

12. The type `void`, when applied to a function declaration, corresponds to an HP Pascal procedure.

13. HP Pascal allocates one byte for `Boolean` variables, and only accesses the rightmost bit to determine its value. HP Pascal uses a 1 to represent *true* and zero for *false*; HP C interprets any nonzero value as *true* and interprets zero as *false*.

14. HP Pascal sets are packed arrays of unsigned bits. For example, given the HP Pascal set:

```
TYPE
    SET_10 = SET OF 0 .. 9;
VAR s: SET_10;
```

the corresponding HP C `struct` would be:

```
typedef struct {
     unsigned int b0 : 1;
     unsigned int b1 : 1;
     unsigned int b2 : 1;
     unsigned int b3 : 1;
     unsigned int b4 : 1;
     unsigned int b5 : 1;
     unsigned int b6 : 1;
     unsigned int b7 : 1;
     unsigned int b8 : 1;
     unsigned int b9 : 1;
} SET_10;

SET_10 s;
```

Also, the following operation in HP Pascal:

```
s := s + [9];
```

has the following corresponding HP C code:

```
s.b9 = 1;
```

15. HP C and HP Pascal file types and I/O operations do not correspond.

Passing Parameters Between HP C and HP Pascal

This section describes additional information on parameter passing.

1. All HP C parameters are passed by value except arrays and functions, which are always passed as pointers. Reference parameters to HP Pascal can be implemented in two ways: first, by passing the address of an object by applying the address operator `&` to the variable; second, by declaring a variable to be a pointer to such a type, assigning an address to the pointer variable, and passing the pointer.

If an HP Pascal procedure or function has a parameter that is an array by value, it can be called from HP C by passing a `struct` that contains an array of the corresponding type.

2. Be careful when passing strings to HP Pascal. If the routine expects a packed array of `char`, be sure to pass a `char` array. If the routine expects a user-defined string, pass the structure declared in Note 5 above.

   The examples below are HP Pascal and HP C source files that show the parameter passing rules. The HP Pascal file contains 2 subroutines, `pass_char_arrays()` and `pass_a_string()`. The HP C file contains the main line routine that calls these two subroutines and displays the results. The HP C program is annotated with the expected results.

   The following is the HP Pascal procedure called from HP C:

```
$subprogram$
program p;
const len = 10;
type
  pac_10  = packed array [1..10] of char;
  string_10 = string [len];

  function pass_char_arrays (a: pac_10;
          var b: pac_10;
              c: string_10;
          var d: string_10) : integer;
      var
          i : integer;
          ret_val : integer;
      begin
          ret_val := 0;
          for i := 1 to len - 1 do
          begin
              if ( a[i] <> 'a' ) then
                  ret_val := 1;
              a[i]    := 'z';
              if ( b[i] <> 'b' ) then
                  ret_val := 256;
              b[i]    := 'y';
          end;
```

```
            for i := 1 to strlen (c) do
            begin
                if ( c[i] <> 'c' ) then
                    ret_val := 65536;
                c[i] := 'x';
            end;

            for i := 1 to strlen (d) do
            begin
                if ( d[i] <> 'd' ) then
                    ret_val := maxint;
                d[i] := 'w';
            end;
            pass_char_arrays := ret_val;
        end;

function pass_a_string (var a: string) : integer;
  var
      i       : integer;
      ret_val : integer;
  begin
      ret_val := 0;
      for i := 1 to strlen (a) do
      begin
          if (a[i] <> 'x' ) then
              ret_val := maxint;
          a[i] := 'q';
      end;
      pass_a_string := ret_val;
  end;

begin
end.
```

Figure 3-1. HP Pascal Procedure

The following HP C main program calls the HP Pascal procedure:

```
#include <stdio.h>
#include <string.h>
static struct string_10 {
   int cur_len;
   char chars[10];
};
/* a Pascal routine */
extern int pass_char_arrays (/* pac10,
                                var pac10,
                                string_10,
                                var string[10] */);
main(void)
{
   static struct string_10 a, b, c, d;
   int ret_val;
   strcpy (a.chars, "aaaaaaaaa");
   strcpy (b.chars, "bbbbbbbbb");
   strcpy (c.chars, "ccccccccc");
   c.cur_len = strlen (c.chars);
   strcpy (d.chars, "ddddddddd");
   d.cur_len = 5;
   ret_val = pass_char_arrays (a.chars, b.chars, &c, &d);

   printf ("a: %s\n", a.chars);           /* prints aaaaaaaaa */
   printf ("b: %s\n", b.chars);           /* prints yyyyyyyyy */
   printf ("c: %s\n", c.chars); /* value parm prints xxxxxxxxx */
   printf ("d: %s\n", d.chars);           /* prints wwwwwdddd */
   printf ("return mask: %d\n", ret_val); /* print 0 */

   ret_val = pass_a_string (&c);

   printf ("c: %s\n", c.chars);           /* prints qqqqqqqqq */
   printf ("return mask: %d\n", ret_val); /* print 0 */
   return 0;
}
```

Figure 3-2. Calling a Pascal Procedure from HP C

The program produces the following output:

```
a: aaaaaaaaa
b: yyyyyyyyy
c: xxxxxxxxx
d: wwwwwdddd
return mask: 0
c: qqqqqqqqq
return mask: 0
```

The routine `pass_a_string()` expects a generic string (described in Note 6 above), so you must pass an extra argument. The extra argument consists of a value parameter containing the maximum length of the `char` array.

3. HP Pascal routines do not maintain a null byte at the end of HP C strings. HP Pascal determines the current length of the string by maintaining the length in a 4-byte integer preceding the character data. When an HP Pascal procedure or function (that takes as a parameter a string by reference) is called, the following code is necessary if the Pascal routine modifies the string:

```
pass_a_string (a, temp);  /* From note 2 above */
a.chars[a.cur_len] = '\0';
```

4. In non-ANSI mode, HP C promotes most `float` (32-bit) arguments to `double` (64-bit). Therefore, all arithmetic using objects defined as `float` is actually using `double` code. `Float` code is only used when the `float` objects are stored.

   In ANSI mode where function prototypes have been declared with a `float` parameter, no automatic promotion is performed. If the prototype is within the current scope, `floats` will not be automatically promoted.

   To call an HP Pascal routine that expects an argument of type `REAL` (32-bits), you may either declare a function prototype in ANSI mode, use the `+r` command line option in non-ANSI mode to always pass `floats` as `floats`, or declare the actual parameter as a `struct` with a `float` as its only field, such as:

```
typedef struct {float f;} PASCAL_REAL_ARG;
```

5. HP Pascal global data can usually only be accessed by HP C if the data is declared at the outermost level. HP Pascal stores the names of the objects in lowercase letters.

   For example, the HP Pascal global:

```
PROGRAM example;

VAR
   PASCAL_GLOBAL: INTEGER;

BEGIN END.
```

   is accessed by HP C with this declaration:

```
extern int pascal_global;
```

The Pascal compiler directives `$GLOBAL$` and `$EXTERNAL$` can be used to share global data between HP Pascal and HP C.

The `$EXTERNAL$` directive should be used to reference C globals from a Pascal *subprogram*.

The `$GLOBAL$` directive should be used to make Pascal globals visible to other languages such as HP C. It should be used if it is necessary to share globals when calling C functions from a Pascal *program*.

Linking HP Pascal Routines on HP-UX

When calling HP Pascal routines, you must include the HP Pascal run-time libraries by adding the following option to the `cc` command line:

```
-lcl
```

Additionally, the `-lm` option may be necessary if the Pascal routines use the Pascal predefined math functions.

For details on linking external libraries, see the `-l` option of the `cc(1)` and `ld(1)` commands in the *HP-UX Reference* manual.

Comparing HP C and HP FORTRAN 77

Table 3-2 shows the differences in storage allocation between HP C and HP FORTRAN 77. The notes the table refers to are located after the table in the section called "Notes on HP C and HP FORTRAN 77."

Table 3-2. HP C versus HP FORTRAN 77 Storage

HP C Type	HP C Description	HP FORTRAN 77 Type	HP FORTRAN 77 Description
`char`, `signed char`, `char enum`	1 byte, byte aligned	--	1 byte, 1-byte aligned
`unsigned char`	1 byte, byte aligned	`CHARACTER*1`	1 byte, 1-byte aligned
`short`, `short enum`	2 bytes, 2-byte aligned	`INTEGER*2`	2 bytes, 2-byte aligned
`unsigned short`	2 bytes, 2-byte aligned	--	--
`int`, `int enum`	4 bytes, 4-byte aligned	`INTEGER*4` or `INTEGER`	4 bytes, 4-byte aligned
`unsigned int`	4 bytes, 4-byte aligned	--	--
`long`, `long enum`	4 bytes, 4-byte aligned	`INTEGER*4` or `INTEGER`	4 bytes, 4-byte aligned
`unsigned long`	4 bytes, 4-byte aligned	--	--
`float`	4 bytes, 4-byte aligned	`REAL` or `REAL*4`	4 bytes, 4-byte aligned
`double`	8 bytes, 8-byte aligned	`REAL*8` or `DOUBLE PRECISION`	8 bytes, 8-byte aligned
`long double`	16 bytes, 16-byte aligned	`REAL*16`	16 bytes, 16-byte aligned
(See Note 1)	8 bytes, 4-byte aligned	`COMPLEX` or `COMPLEX*8`	8 bytes, 4-byte aligned
(See Note 2)	16 bytes, 8-byte aligned	`DOUBLE COMPLEX` or `COMPLEX*16`	16 bytes, 8-byte aligned

Table 3-2. HP C versus HP FORTRAN 77 Storage (continued)

HP C Type	HP C Description	HP FORTRAN 77 Type	HP FORTRAN 77 Description
`enum`	4 bytes, 4-byte aligned	`INTEGER*4` or `INTEGER`	4 bytes, 4-byte aligned
pointer to *type* long pointer to *type*	--	Not available	--
string (`char *`)	--	`CHARACTER*n` (*See Note 3*)	--
`char` array	--	`CHARACTER*1` array (*See Notes 4 & 5*)	--
(*See Note 5*)	--	Hollerith array	--
`arrays`	Size is number of elements times element size. Aligned according to element type.	(*See Note 4*)	Size is number of elements times element size. Aligned according to element type.
`struct`	(*See Note 6*)	`STRUCTURE`	Used to declare FORTRAN 77 record structures.
`union`	(*See Note 6*)	`UNION`	Used to declare FORTRAN 77 union types.
`short` (Used for logical test)	2 bytes, 2-byte aligned	`LOGICAL*2` (*See Note 7*)	2 bytes, 2-byte aligned
`int` (Used for logical test)	4 bytes, 4-byte aligned	`LOGICAL*4` (*See Note 7*)	4 bytes, 4-byte aligned
`void`	-- -	Used when calling a SUBROUTINE	--
function	--	Used when calling a FUNCTION	--

Notes on HP C and HP FORTRAN 77

1. The following HP C structure is equivalent to the HP FORTRAN 77 type listed in the table:

```
struct complex {
   float real_part;
   float imag_part;
};
```

2. The following HP C structure is equivalent to the HP FORTRAN 77 type listed in the table:

```
struct double_complex {
   double real_part;
   double imag_part;
};
```

3. HP FORTRAN 77 passes strings as parameters using string descriptors corresponding to the following HP C declarations:

```
   char *char_string;  /* points to string */
   int len;            /* length of string */
```

4. HP C stores arrays in row-major order, whereas HP FORTRAN 77 stores arrays in column-major order. The lower bound for HP C is always zero; for HP FORTRAN 77, the default lower bound is 1.

5. HP C terminates character strings with a null byte, while HP FORTRAN 77 does not.

6. The size is equal to the size of all members plus any padding needed for the alignment. (See Chapter 2 for details on alignment.) The alignment is that of the member with the strictest alignment requirement.

7. HP C and HP FORTRAN 77 do not share a common definition of true or false. In HP FORTRAN 77, logical values are determined by the low-order bit of the high-order byte. If this bit is 1, the logical value is `.TRUE.`, and if the bit is zero, the logical value is `.FALSE.`. HP C interprets nonzero value as *true* and interprets zero as *false*.

Mixing C and FORTRAN File I/O

A FORTRAN unit cannot be passed to a C routine to perform I/O on the associated file. Nor can a C file pointer be used by a FORTRAN routine. However, a file created by a program written in either language can be used by a program of the other language if the file is declared and opened within the latter program. C accesses the file using I/O subroutines and intrinsics. This method of file access can also be used from FORTRAN instead of FORTRAN I/O.

Be aware that HP FORTRAN 77 on HP 9000 workstations and servers using HP-UX uses the unbuffered I/O system calls `read` and `write` (described in the *HP-UX Reference* manual) for all terminal I/O, magnetic tape I/O, and direct access I/O. It uses the system calls `fread` and `fwrite` for all other I/O. This can cause problems in programs that mix C and FORTRAN I/O. In particular, C programs that use `stdio(3S)` output procedures such as `printf` and `fwrite` and FORTRAN output statements must flush `stdio` buffers (by calling the `libc` function `fflush`) if they are in use before returning to FORTRAN output or the I/O may be asynchronous (if the library is using `write`).

Mixing FORTRAN direct, terminal, or tape READ statements with `stdio fread` input results in the FORTRAN READ commencing from the beginning of the next block after the contents of the buffer, not from the current position of the input cursor in the `fread` buffer. The same situation in reverse may occur by mixing `read` with a FORTRAN sequential disc read. You can avoid these problems by using only the `read` and `write` calls in the C program that the FORTRAN I/O library uses.

Passing Parameters Between HP C and HP FORTRAN 77

All parameters in HP FORTRAN 77 are passed by reference. This means that all arguments in an HP C call to an HP FORTRAN 77 routine must be pointers. In addition, all parameters in an HP C routine called from HP FORTRAN 77 must be pointers, unless the HP FORTRAN 77 code uses the `$ALIAS` directive to define the parameters as value parameters. Refer to the example called "HP FORTRAN 77 Nested Structure" later in this chapter.

Passing string variables of any length must be done by: building and passing a two-parameter descriptor (defined in Note 3 above), initializing the string appropriately, and by passing two arguments. The two arguments are the

pointer to the characters and the value of the length word. This is shown below:

```
char *chars = "Big Mitt";
int len;
  ⋮
len = strlen(chars);

pass_c_string (chars, len);
   ⋮
```

Linking HP FORTRAN 77 Routines on HP-UX

When calling HP FORTRAN 77 routines on an HP-UX system, you have to include the HP FORTRAN 77 run-time libraries by adding the option:

```
-lcl
```

to the `cc` command line.

For details on linking external libraries, see the `-l` option of the `cc(1)` and `ld(1)` commands in the *HP-UX Reference* manual.

Comparing Structured Data Type Declarations

This section shows how to declare a nested structure in HP C, HP Pascal, and HP FORTRAN 77.

HP C Nested Structure

```
struct x {
   char y [3];
   short z;
   char w [5];
};

struct q {
   char n;
   struct x v [2];
```

```
   double u;
   char t;
} a;

struct u{
   union {
      int x;
      char y[4];
   } uval;
};
```

HP Pascal Nested Structure

```
TYPE
   x = RECORD
       y : PACKED ARRAY [1 .. 3] OF CHAR;
       z : SHORTINT;
       w : PACKED ARRAY [1 .. 5] OF CHAR;
       END;

   q = RECORD
       n : CHAR;
       v : PACKED ARRAY [1 .. 2] OF x;
       u : LONGREAL;
       t : CHAR;
       END;

   u = RECORD
        CASE
        Boolean OF
          TRUE : (x : INTEGER);
          FALSE: (y : ARRAY[1..4] of CHAR);
       END;
VAR a:q;
```

HP FORTRAN 77 Nested Structure

```
program main
structure /x/
        character*3 y
```

```
        integer*2 z
        character*5 w
end structure

structure /q/
        character n
        record /x/ v(2)
        real*8 u
        character t
end structure


structure /u/
        union
                map
                    integer*4 x
                end map
                map
                    character*4 y
                end map
        end union
end structure
```

4

Optimizing HP C Programs

This chapter discusses the following:

- When and how to use the optimizer.
- The four levels of optimization.
- Profile-based optimization.

The HP C optimizer transforms programs so machine resources are used more efficiently. The optimizer can dramatically improve application run-time speed. HP C performs only minimal optimizations unless you specify otherwise. You activate additional optimizations using HP C command line options.

There are four major levels of optimization: levels 1, 2, 3, and 4. Level 4 optimization can produce the fastest executable code. Level 4 is a superset of the other levels.

Additional parameters enable you to control the size of the executable program, compile time, and aggressiveness of the optimizations performed.

Compile time memory and CPU usage increase with each higher level of optimization due to the increasingly complex analysis that must be performed. You can control the trade-offs between compile-time penalties and code performance by choosing the level of optimization you desire.

Generally, the optimizer is not used during code development. It is used when compiling production-level code for benchmarking and general use.

Summary of Major Optimization Levels

The HP C major optimization options are summarized in Table 4-1.

Table 4-1. HP C Major Optimization Options

Option	Description	Benefits
+O0 (default)	Constant folding and simple register assignment.	Compiles fastest.
+O1	Level 0 optimizations plus instruction scheduling and optimizations that can be performed on small sections of code.	Produces faster programs than level 0. Compiles faster than level 2.
+O2 or -O	Level 1 optimizations, plus optimizations performed over entire functions in a single file. Optimizes loops in order to reduce pipeline stalls. Performs scalar replacement, and analysis of data-flow, memory usage, loops and expressions.	Can produce faster run-time code than level 1 if programs use loops extensively. Compiles faster than level 3. Loop-oriented floating point intensive applications may see run times reduced by 50%. Operating system and interactive applications that use the already optimized system libraries can achieve 30% to 50% additional improvement.
+O3	Level 2 optimizations, plus full optimization across all subprograms within a single file. Includes subprogram inlining.	Can produce faster run-time code than level 2 on code that frequently calls small functions. Links faster than level 4.
+O4	Level 3 optimizations, plus full optimizations across the entire application program. Includes global and static variable optimization and inlining across the entire program. Optimizations are performed at link-time.	Produces faster run-time code than level 3 if programs use many global variables or if there are many opportunities for inlining procedure calls.

Supporting Optimization Options

Table 4-2 shows optimization options that support the core optimization levels. These optimizations are performed only when specifically invoked. They are available at all optimization levels.

Table 4-2. Other Supporting Optimizations

Option	Description	Benefits
+ESfic	Replaces millicode calls with inline code.	Run-time code is faster because fast indirect calls are used instead of millicode calls.
+ESlit	Places string literals and constants defined with the ANSI C `const` type qualifier into read-only data storage.	Reduces memory requirements and improves run-time speed in multi-user applications. Can improve data-cache utilization.
+I, +P	Enables all profile-based optimizations. Uses execution profile data to identify the most frequently executed code paths. Repositions functions, basic blocks, and aids other optimizations according to these frequently executed paths.	Improves code locality and cache hit rates. Improves efficiency of other optimizations. Benefits most applications, especially large applications with multiple compilation units. May be used at any optimization level.

Enabling Basic Optimization

To enable **basic optimizations**, use the -O option (equivalent to +O2), as follows:

```
cc -O sourcefile.c
```

Basic optimizations do not change the behavior of ANSI C standard-conforming code. They improve run-time execution time but only increase compile time and link time by a moderate amount.

Enabling Different Levels of Optimization

There may be times when you want more or less optimization than what is provided with the basic `-O` option.

Level 1 Optimization

To enable level 1 optimization, use the `+O1` option, as follows:

```
cc +O1 sourcefile.c
```

Level 1 optimization compiles quickly, but still provides some run-time speedup.

Level 2 Optimization

To enable level 2 optimization, use the `+O2` option, as follows:

```
cc +O2 sourcefile.c
```

Level 2 (equivalent to `-O`) takes more time to compile, but produces greatly improved run-time speed.

Level 3 Optimization

To enable level 3 optimization, use the `+O3` option, as follows:

```
cc +O3 sourcefile.c
```

Level 3 does full optimization of all subprograms within a single file.

Level 4 Optimization

To enable level 4 optimization, use the `+O4` option, as follows:

```
cc +O4 sourcefile.c
```

Level 4 can potentially produce the greatest improvements in speed by performing optimizations across multiple object files. Level 4 does optimizations at link time, so compiles will be faster, but links will be longer.

Depending on the size and number of the modules, compiling at level 4 can consume a large amount of virtual memory. Level 4 may consume roughly

1.25 megabytes per 1000 lines of noncommented source. When you use level 4 on a large application, it is a good idea to increase the system swap space. For information on increasing system swap space, see the book *HP System Administration Tasks*.

Changing the Aggressiveness of Optimizations

At each level of optimization, you can control the aggressiveness of the optimizations performed.

Use the `+Oconservative` option at optimization level 2, 3, or 4 if you are not sure if your code conforms to standards. This option provides more safety.

Use the `+Oaggressive` option at optimization level 2, 3, or 4 for best performance when you are willing to risk changes to the behavior of your programs. Using the `+Oaggressive` option can cause your program to have compilation or run-time problems that require troubleshooting.

Enabling Only Conservative Optimizations

You can enable **conservative optimizations** at the second, third, or fourth optimization levels by using the `+Oconservative` option, as follows:

```
cc +O2 +Oconservative sourcefile.c
```

or:

```
cc +O3 +Oconservative sourcefile.c
```

or:

```
cc +O4 +Oconservative sourcefile.c
```

Conservative optimizations are optimizations that do not change the behavior of code, in most cases, even if the code does not conform to standards.

Use the conservative optimizations provided with level 2, 3, and 4 when your code is non-ANSI.

Enabling Aggressive Optimizations

To enable **aggressive optimizations** at the second, third, or fourth optimization levels, use the `+Oaggressive` option as follows:

```
cc +O2 +Oaggressive sourcefile.c
```

or:

```
cc +O3 +Oaggressive sourcefile.c
```

or:

```
cc +O4 +Oaggressive sourcefile.c
```

Aggressive optimizations are new optimizations or are optimizations that can change the behavior of programs. These optimizations may do any of the following:

- convert certain library calls to millicode and inline instructions
- cause the inlined routines `strcpy()`, `sqrt()`, `fabs()`, and `alloca()` to not return the routine's completion status in `ERRNO`
- alter exception handling and asynchronous interrupt handling as a result of instruction scheduling optimization
- cause less precise floating-point results
- cause programs that perform comparisons between pointers to shared memory and pointers to private memory to run incorrectly

Use aggressive optimizations with stable, well-structured, ANSI-conforming code. These types of optimizations give you faster code, but are riskier than the default optimizations.

Removing Compilation Time Limits When Optimizing

You can remove optimization time restrictions at the second, third, or fourth optimization levels by using the `+Onolimit` option as follows:

```
cc +O2 +Onolimit sourcefile.c
```

or:

```
cc +O3 +Onolimit sourcefile.c
```

or:

```
cc +O4 +Onolimit sourcefile.c
```

By default, the optimizer limits the amount of time spent optimizing large programs at levels 2, 3, and 4. Use this option if longer compile times are acceptable because you want additional optimizations to be performed.

Limiting the Size of Optimized Code

You can disable optimizations that expand code size at the second, third, and fourth optimization levels by using the `+Osize` suboption, as follows:

```
cc +O2 +Osize sourcefile.c
```

or:

```
cc +O3 +Osize sourcefile.c
```

or:

```
cc +O4 +Osize sourcefile.c
```

Most optimizations improve execution speed and decrease executable code size. A few optimizations significantly increase code size to gain execution speed. The `+Osize` option disables these code-expanding optimizations.

Use this option if you have limited main memory, swap space, or disk space.

Specifying Maximum Optimization

To get maximum optimization, use:

`cc +Oall` *Performs maximum optimization.*

Use `+Oall` with stable, well-structured, ANSI-conforming code. These types of optimizations give you the fastest code, but are *riskier* than the default optimizations.

You can use `+Oall` at optimization levels 2, 3, and 4. The default is `+Onoall`.

The `+Oall` option by itself (specified without the `+O2, +O3,` or `+O4` options) combines the `+O4 +Oaggressive +Onolimit` options. This combination performs aggressive optimizations with unrestricted compile time at the highest level of optimization.

Combining Optimization Parameters

Optimization parameters that affect code size, compile-time, and the aggressiveness of the optimizations performed can be combined with a level of optimization.

For example, to specify conservative optimizations at level 2 and disable code-expanding optimizations, use:

```
cc +O2 +Oconservative +Osize sourcefile.c
```

`+Olimit` and `+Osize` can be used with either `+Oaggressive` or `+Oconservative`.

You cannot use `+Oaggressive` with `+Oconservative`.

Summary of Optimization Parameters

The HP C optimization parameters are summarized in Table 4-3.

Table 4-3. HP C Optimization Parameters

Option	What It Does	Level of Opt
`+O[no]aggressive`	The `+O[no]aggressive` option enables optimizations that can result in significant performance improvement, but that can change a program's behavior. These optimizations include newly released optimizations and the optimizations invoked by the following advanced optimization options:[1] ■ `+Osignedpointers` ■ `+Oregionsched` ■ `+Oentrysched` ■ `+Onofltacc` ■ `+Olibcalls` ■ `+Onoinitcheck` ■ `+Ovectorize` The default is `+Onoaggressive`.	2, 3, 4
`+O[no]all`	The `+Oall` option performs maximum optimization, including aggressive optimizations and optimizations that can significantly increase compile time and memory usage. The default is `+Onoall`.	4

1 See Table 4-4 and the following section for details about advanced optimization options.

Table 4-3. HP C Optimization Parameters (continued)

Option	What It Does	Level of Opt
`+O[no]conservative`	The `+O[no]conservative` option causes the optimizer to make conservative assumptions about the code when optimizing it. Use `+Oconservative` when conservative assumptions are necessary due to the coding style, as with non-standard conforming programs. The `+Oconservative` option relaxes the optimizer's assumptions about the target program. The default is `+Onoconservative`.	2, 3, 4
`+O[no]info`	`+Oinfo` displays informational messages about the optimization process. This option supports the core optimization levels, and therefore, can be used at levels 0-4. The default is `+Onoinfo`.	0, 1, 2, 3, 4
`+O[no]limit`	The `+Olimit` option suppresses optimizations that significantly increase compile-time or that can consume a lot of memory. The `+Onolimit` option allows optimizations to be performed regardless of their effect on compile-time or memory usage. The default is `+Olimit`.	2, 3, 4
`+O[no]size`	The `+Osize` option suppresses optimizations that significantly increase code size. The `+Onosize` option does not prevent optimizations that can increase code size. The default is `+Onosize`.	2, 3, 4

Profile-Based Optimization

Profile-based optimization (PBO) is a set of performance-improving code transformations based on the run-time characteristics of your application.

There are three steps involved in performing this optimization:

1. *Instrumentation* - Insert data collection code into the object program.
2. *Data Collection* - Run the program with representative data to collect execution profile statistics.
3. *Optimization* - Generate optimized code based on the profile data.

Invoke profile-based optimization through HP C by using any level of optimization and the `+I` and `+P` options on the `cc` command line.

Compile times will be fast and link times will be slow when using PBO because code generation happens at link time.

Instrumenting the Code

To instrument your program, use the `+I` option as follows:

`cc -Aa +I -O -c sample.c`	*Compile for instrumentation.*
`cc -o sample.exe +I -O sample.o`	*Link to make instrumented executable.*

The first command line uses the -O option to perform level 2 optimization and instruments the code. The `-c` option in the first command line suppresses linking and creates an intermediate object file called `sample.o`. The `.o` file can be used later in the optimization phase, avoiding a second compile.

The second command line uses the `-o` option to link `sample.o` into `sample.exe`. The `+I` option instruments `sample.exe` with data collection code. Note that instrumented programs run slower than non-instrumented programs. Only use instrumented code to collect statistics for profile-based optimization.

Collecting Data for Profiling

To collect execution profile statistics, run your *instrumented* program with representative data as follows:

```
sample.exe < input.file1    Collect execution profile data.
sample.exe < input.file2
```

This step creates and logs the profile statistics to a file, by default called `flow.data`. The data collection file is a structured file that may be used to store the statistics from multiple test runs of different programs that you may have instrumented.

Performing Profile-Based Optimization

To optimize the program based on the previously collected run-time profile statistics, relink the program as follows:

```
cc -o sample.exe +P -O sample.o
```

An alternative to this procedure is to recompile the source file in the optimization step:

```
cc -o sample.exe +I -O sample.c      instrumentation
sample.exe < input.file1             data collection
cc -o sample.exe +P -O sample.c      optimization
```

Maintaining Profile Data Files

Profile-based optimization stores execution profile data in a disk file. By default, this file is called `flow.data` and is located in your current working directory.

You can override the default name of the profile data file. This is useful when working on large programs or on projects with many different program files.

The `FLOW_DATA` environment variable can be used to specify the name of the profile data file with either the `+I` or `+P` options. The `+df` command line option can be used to specify the name of the profile data file when used with the `+P` option.

The `+df` option takes precedence over the `FLOW_DATA` environment variable.

In the following example, the `FLOW_DATA` environment variable is used to override the `flow.data` file name. The profile data is stored instead in `/users/profiles/prog.data`.

```
%setenv FLOW_DATA /users/profiles/prog.data
%cc -Aa -c +I +03 sample.c
%cc -o sample.exe +I +03 sample.o
%sample.exe < input.file1
%cc -o sample.exe +P +03 sample.o
```

In the next example, the `+df` option is used to override the `flow.data` file name with the name `/users/profiles/prog.data`.

```
%cc -Aa -c +I +03 sample.c
%cc -o sample.exe +I +03 sample.o
%sample.exe < input.file1
%mv flow.data /users/profile/prog.data
%cc -o sample.exe +df /users/profiles/prog.data +P +03 sample.o
```

Maintaining Instrumented and Optimized Program Files

You can maintain both instrumented and optimized versions of a program. You might keep an instrumented version of the program on hand for development use, and several optimized versions on hand for performance testing and program distribution.

Care must be taken when maintaining different versions of the executable file because the *instrumented* program file name is used as the *key identifier* when *storing* execution profile data in the data file.

The optimizer must know what this *key identifier* name is in order to find the execution profile data. By default, the *key identifier* name used to *retrieve* the profile data is the *instrumented* program file name used to run the program for data collection.

When you optimize a program file and the optimized program file name is different from the instrumented program file name, you must use the `+pgm` option. Specify the instrumented program file name with this option. The optimizer uses this value as the *key identifier* to retrieve execution profile data.

In the following example, the instrumented program file name is `sample.inst`. The optimized program file name is `sample.opt`. The `+pgm` name option is used to pass the instrumented program name to the optimizer:

```
%cc -Aa -c +I +03 sample.c
%cc -o sample.inst +I +03 sample.o
%sample.inst < input.file1
%cc -o sample.opt +P +03 +pgm sample.inst sample.o
```

Profile-Based Optimization Notes

When using profile-based optimization, please note the following:

- Because the linker performs code generation for profile-based optimization, linking object files compiled with `+I` and `+P` takes more time than linking ordinary object files. However, compile-times will be relatively fast. This is because the compiler is only generating the intermediate code.
- Profile-based optimization has a greater impact on application performance at each higher level of optimization.
- Profile-based optimization should be enabled during the final stages of application development. To obtain the best performance, re-profile and re-optimize your application after making source code changes.
- If you use level-4 or profile-based optimization and do not use `+DA` to generate code for a specific version of PA-RISC, note that code generation occurs at link time. Therefore, the system on which you link, rather than compile, determines the object code generated.
- If you use level-4 or profile-based optimization and do not use `+DS` to specify instruction scheduling, note that instruction scheduling occurs at link time. Therefore, the system on which you link, rather than compile, determines the implementation of instruction scheduling.

For more information on profile-based optimization, see the *HP-UX Linker and Libraries Online User Guide*.

Controlling Specific Optimizer Features

Most of the time, specifying optimization level 1, 2, 3, or 4 should provide you with the control over the optimizer that you need. Additional parameters are provided when you require a finer level of control.

At each level, you can turn on and off specific optimizations using the `+O[no]`*optimization* option. The *optimization* parameter is the name of a specific optimization technique described below. The optional prefix [no] disables the specified optimization.

The following section describes the optimizations that can be turned on or off, their defaults, and the optimization levels at which they may be used. The options listed in Table 4-4 are described below.

Table 4-4. HP C Advanced Optimization Options

Option	Option
`+O[no]dataprefetch`	`+O[no]entrysched`
`+O[no]fail_safe`	`+O[no]fastaccess`
`+O[no]fltacc`	`+O[no]global_ptrs_unique`
`+O[no]initcheck`	`+O[no]inline`
`+Oinline_budget`	`+O[no]libcalls`
`+O[no]loop_transform`	`+O[no]loop_unroll`
`+O[no]moveflops`	`+O[no]parallel`
`+O[no]parallel_env`	`+O[no]parmsoverlap`
`+O[no]pipeline`	`+O[no]procelim`
`+O[no]ptrs_ansi`	`+O[no]ptrs_strongly_typed`
`+O[no]ptrs_to_globals`	`+O[no]regionsched`
`+O[no]regreassoc`	`+O[no]sideeffects`
`+O[no]signedpointers`	`+O[no]static_prediction`
`+O[no]vectorize`	`+O[no]volatile`
`+O[no]whole_program_mode`	

`+O[no]dataprefetch`

Optimization level(s): 2, 3, 4

Default: `+Onodataprefetch`

When `+Odataprefetch` is enabled, the optimizer will insert instructions within innermost loops to explicitly prefetch data from memory into the data cache. Data prefetch instructions will be inserted only for data structures referenced within innermost loops using simple loop varying addresses (that is, in a simple arithmetic progression). It is only available for PA-RISC 2.0 targets.

The math library contains special prefetching versions of vector routines. If you have a PA-RISC 2.0 application that contains operations on arrays larger than 1 megabyte in size, using `+Ovectorize` in conjunction with `+Odataprefetch` may improve performance substantially.

Use this option for applications that have high data cache miss overhead.

`+O[no]entrysched`

Optimization level(s): 1, 2, 3, 4

Default: `+Onoentrysched`

The `+Oentrysched` option optimizes instruction scheduling on a procedure's entry and exit sequences. Enabling this option can speed up an application. The option has undefined behavior for applications which handle asynchronous interrupts. The option affects unwinding in the entry and exit regions.

At optimization level `+O2` and higher (using dataflow information), save and restore operations become more efficient.

This option can change the behavior of programs that perform exception-handling or that handle asynchronous interrupts. The behavior of `setjmp()` and `longjmp()` is not affected.

`+O[no]fail_safe`

Optimization level(s): 1, 2, 3

Default: `+Ofail_safe`

The `+Ofail_safe` option allows compilations with internal optimization errors to continue by issuing a warning message and restarting the compilation at `+O0`.

You can use `+Onofail_safe` at optimization levels 1, 2, 3, or 4 when you want the internal optimization errors to abort your build.

This option is disabled when compiling for parallelization.

`+O[no]fastaccess`

Optimization level(s): 0, 1, 2, 3, 4

Default: `+Onofastaccess` at optimization levels 0, 1, 2 and 3, `+Ofastaccess` at optimization level 4

The `+Ofastaccess` option optimizes for fast access to global data items.

Use `+Ofastaccess` to improve execution speed at the expense of longer compile times.

`+O[no]fltacc`

Optimization level(s): 2, 3, 4

The `+Onofltacc` option allows the compiler to perform floating-point optimizations that are algebraically correct but that may result in numerical differences. For example, this option may change the order of expression evaluation as such: If `a`, `b`, and `c` are floating-point variables, the expressions `(a + b) + c` and `a + (b + c)` may give slightly different results due to roundoff. In general, these differences will be insignificant.

The `+Onofltacc` option also enables the optimizer to generate fused multiply-add (FMA) instructions, the `FMPYFADD` and `FMPYNFADD`. These instructions improve performance but occasionally produce results that may differ from results produced by code without FMA instructions. In general, the differences are slight. FMA instructions are only available on PA-RISC 2.0 systems.

Specifying `+Ofltacc` disables the generation of FMA instructions as well as some other floating-point optimizations. Use `+Ofltacc` if it is important that the compiler evaluate floating-point expressions as it does in unoptimized code. The `+Ofltacc` option does not allow any optimizations that change the order of expression evaluation and therefore may affect the result.

If you are optimizing code at level 2 or higher and do not specify `+Onofltacc` or `+Ofltacc`, the optimizer will use FMA instructions, but will not perform floating-point optimizations that involve expression reordering or other optimizations that potentially impact numerical stability.

The list below identifies the different actions taken by the optimizer according to whether you specify `+Ofltacc`, `+Onofltacc`, or neither option.

Optimization Options	Expression Reordering?	FMA?
+O2	No	Yes
+O2 +Ofltacc	No	No
+O2 +Onofltacc	Yes	Yes

`+O[no]global_ptrs_unique [=`*name1,name2, ...nameN*`]`

Optimization level(s): 2, 3, 4

Default: `+Onoglobal_ptrs_unique`

Use this option to identify unique global pointers, so that the optimizer can generate more efficient code in the presence of unique pointers, for example by using copy propagation and common sub-expression elimination. A global pointer is unique if it does not alias with any variable in the entire program.

This option supports a comma-separated list of unique global pointer variable names.

Refer to your online *HP C Online Reference* for examples.

`+O[no]initcheck`

Optimization level(s): 2, 3, 4

Default: unspecified

The initialization checking feature of the optimizer has three possible states: on, off, or unspecified. When on (`+Oinitcheck`), the optimizer initializes to zero any local, scalar, non-static variables that are uninitialized with respect to at least one path leading to a use of the variable.

When off (`+Onoinitcheck`), the optimizer issues warning messages when it discovers definitely uninitialized variables, but does not initialize them.

When unspecified, the optimizer initializes to zero any local, scalar, non-static variables that are definitely uninitialized with respect to all paths leading to a use of the variable.

Use `+Oinitcheck` to look for variables in a program that may not be initialized.

`+O[no]inline[=`*name1, name2, ...nameN*`]`

Optimization level(s): 3, 4

Default: `+Oinline`

When `+Oinline` is specified without a *name* list, any function can be inlined. For inlining to be successful, follow prototype definition for function calls in the appropriate header file.

When specified with a *name* list, the named functions are important candidates for inlining. For example, saying

```
+Oinline=foo,bar +Onoinline
```

indicates that inlining be strongly considered for `foo` and `bar`; all other routines will not be considered for inlining, since `+Onoinline` is given.

When this option is disabled with a name list, the compiler will not consider the specified routines as candidates for inlining. For example, saying

```
+Onoinline=baz,x
```

indicates that inlining should not be considered for `baz` and `x`; all other routines will be considered for inlining, since `+Oinline` is the default.

The `+Onoinline` disables inlining for all functions or a specific list of functions.

Use this option when you need to precisely control which subprograms are inlined. Use of this option can be guided by knowledge of the frequency with which certain routines are called and may be warranted by code size concerns.

`+Oinline_budget[=`*n*`]`

Optimization level(s): 3, 4

Default: `+Oinline_budget=100`

where *n* is an integer in the range 1 - 1000000 that specifies the level of aggressiveness, as follows:

n = 100
Default level of inlining.

n > 100
More aggressive inlining. The optimizer is less restricted by compilation time and code size when searching for eligible routines to inline.

n = 1
Only inline if it reduces code size.

The `+Onolimit` and `+Osize` options also affect inlining. Specifying the `+Onolimit` option has the same effect as specifying `+Oinline_budget=200`. The `+Osize` option has the same effect as `+Oinline_budget=1`.

Note, however, that the `+Oinline_budget` option takes precedence over both of these options. This means that you can override the effect of `+Onolimit` or `+Osize` option on inlining by specifying the `+Oinline_budget` option on the same compile line.

`+O[no]libcalls`

Optimization level(s): 0, 1, 2, 3, 4

Default: `+Onolibcalls`

Use the `+Olibcalls` option to increase the runtime performance of code which calls standard library routines in simple contexts. The `+Olibcalls` option expands the following library calls inline:

- `strcpy()`
- `sqrt()`
- `fabs()`
- `alloca()`

Inlining will take place only if the function call follows the prototype definition the appropriate header file. Fast subprogram linkage is also emitted to tuned millicode versions of the math library functions `sin, cos, tan, atan 2, log, pow, asin, acos, atan, exp,` and `log10`. (See the *HP-UX Floating-Point Guide* for the most up-to-date listing of the math library functions.) The calling code must not expect to access `ERRNO` after the function's return.

A single call to `printf()` may be replaced by a series of calls to `putchar()`. Calls to `sprintf()` and `strlen()` may be optimized more effectively, including elimination of some calls producing unused results. Calls to `setjmp()` and `longjmp()` may be replaced by their equivalents `_setjmp()` and `_longjmp()`, which do not manipulate the process's signal mask.

Use `+Olibcalls` to improve the performance of selected library routines only when you are not performing error checking for these routines.

Using `+Olibcalls` with `+Ofltacc` will give different floating point calculation results than those given using `+Ofltacc` without `+Olibcalls`.

The `+Olibcalls` option replaces the obsolete `-J` option.

`+O[no]loop_transform`

Optimization level(s): 3, 4

Default: `+Oloop_transform`

The `+O[no]loop_transform` option enables [disables] transformation of eligible loops for improved cache performance. The most important transformation is the reordering of nested loops to make the inner loop unit stride, resulting in fewer cache misses.

`+Onoloop_transform` may be a helpful option if you experience any problem while using `+Oparallel`.

`+O[no]loop_unroll[=unroll factor]`

Optimization level(s): 2, 3, 4

Default: `+Oloop_unroll`

The `+Oloop_unroll` option turns on loop unrolling. When you use `+Oloop_unroll`, you can also use the unroll factor to control the code expansion. The default unroll factor is `4`, that is, four copies of the loop body. By experimenting with different factors, you may improve the performance of your program.

`+O[no]moveflops`

Optimization level(s): 2, 3, 4

Default: `+Omoveflops`

Allows [or disallows] moving conditional floating point instructions out of loops. The +Onomoveflops option replaces the obsolete `+OE` option. The behavior of floating-point exception handling may be altered by this option.

Use `+Onomoveflops` if floating-point traps are enabled and you do not want the behavior of floating-point exceptions to be altered by the relocation of floating-point instructions.

`+O[no]parallel`

Optimization level(s): 3, 4

Default: `+Onoparallel`

When a program is compiled with the `+Oparallel` option, the compiler looks for opportunities for parallel execution in loops and generates parallel code to execute the loop on the number of processors set by the `MP_NUMBER_OF_THREADS` environment variable discussed in the section "Parallel Execution" at the end of this chapter.

The `+Oparallel` option should not be used for programs that make explicit calls to the kernel threads library `/usr/lib/libpthread.sl`.

`+O[no]parallel` (continued)

`+Onoloop_transform` and `+Onoinline` may be helpful options if you experience any problem while using `+Oparallel`.

You may use `+Oparallel` at optimization levels 3 and 4. The default is `+Onoparallel` at levels 0-4. `+Oparallel` disables `+Ofailsafe`.

Parallelization is incompatible with the `prof` tool, so the `-p` option is disabled by `+Oparallel`. Parallelization is compatible with `gprof`. Special `*crt0.o` startup files are required for programs compiled for a parallel environment. The parallel runtime library, `libmp.a`, must be linked in.

For additional information, see the section "Parallel Execution" at the end of this chapter.

Note At the HP-UX 10.20 release, if a program made of multiple files had any of its files compiled with the `+Oparallel` option, then the remaining files had to be compiled with either the `+Oparallel` or `+O[no]parallel_env` option. The `+Oparallel_env` option ensured a consistent execution environment for all files in the program, including those not to be compiled for parallel execution. At the HP-UX 10.30 release, it is no longer necessary or permissible to use the `+O[no]parallel_env` option.

`+O[no]parmsoverlap`

Optimization level(s): 2, 3, 4

Default: `+Oparmsoverlap`

The `+Oparmsoverlap` option optimizes with the assumption that the actual arguments of function calls overlap in memory.

The `+Onoparmsoverlap` option replaces the obsolete `+Om1` option.

Use `+Onoparmsoverlap` if C programs have been literally translated from FORTRAN programs.

`+O[no]pipeline`

Optimization level(s): 2, 3, 4

Default: `+Opipeline`

Enables [or disables] software pipelining. The `+Onopipeline` option replaces the obsolete `+Os` option.

Use `+Onopipeline` to conserve code space.

`+O[no]procelim`

Optimization level(s): 0, 1, 2, 3, 4

Default: `+Onoprocelim` at levels 0-3, `+Oprocelim` at level 4

When `+Oprocelim` is specified, procedures that are not referenced by the application are eliminated from the output executable file. The `+Oprocelim` option reduces the size of the executable file, especially when optimizing at levels 3 and 4, at which inlining may have removed all of the calls to some routines.

When `+Onoprocelim` is specified, procedures that are not referenced by the application are not eliminated from the output executable file.

The default is `+Onoprocelim` at levels 0-3, and `+Oprocelim` at level 4.

If the `+Oall` option is enabled, the `+Oprocelim` option is enabled.

`+O[no]ptrs_ansi`

Optimization level(s): 2, 3, 4

Default: `+Onoptrs_ansi`

Use `+Optrs_ansi` to make the following two assumptions, which the more aggressive `+Optrs_strongly_typed` does not make:

- An `int *p` is assumed to point to an int field of a struct or union.
- `char *` is assumed to point to any type of object.

When both are specified, `+Optrs_ansi` takes precedence over `+Optrs_strongly_typed`.

For more information about type aliasing see the section "Aliasing Options" later in this chapter.

`+O[no]ptrs_strongly_typed`

Optimization level(s): 2, 3, 4

Default: `+Onoptrs_strongly_typed`

Use `+Optrs_strongly_typed` when pointers are type-safe. The optimizer can use this information to generate more efficient code.

Type-safe (that is, strongly-typed) pointers are pointers to a specific type that only point to objects of that type, and not to objects of any other type. For example, a pointer declared as a pointer to an `int` is considered type-safe if that pointer points to an object only of type `int`, but not to objects of any other type.

Based on the type-safe concept, a set of groups are built based on object types. A given group includes all the objects of the same type.

The term *type-inferred aliasing* is a concept which means any pointer of a type in a given group (of objects of the same type) can only point to any object from the same group; it can not point to a typed object from any other group.

For more information about type aliasing see the section "Aliasing Options" later in this chapter.

Type casting to a different type violates type-inferring aliasing rules. See Example 2 below.

Dynamic casting is allowed. See Example 3 below.

`+O[no]ptrs_strongly_typed` (continued)

For finer detail, see the "[NO]PTRS_STRONGLY_TYPED pragma" section later in this chapter..

Example 1: How Data Types Interact

The optimizer generally spills all global data from registers to memory *before* any modification to global variables or any loads through pointers. However, you can instruct the optimizer on how data types interact so that it can generate more efficient code.

If you have the following:

```
1  int *p;
2  float *q;
3  int a,b,c;
4  float d,e,f;
5  foo()
6  {
7    for (i=1;i<10;i++) {
8               d=e
9              *p=..
10              e=d+f;
11              f=*q;
12   }
13 }
```

With `+Onoptrs_strongly_typed` turned on, the pointers `p` and `q` will be assumed to be disjoint because the types they point to are different types. Without type-inferred aliasing, `*p` is assumed to invalidate all the definitions. So, the use of `d` and `f` on line 10 have to be loaded from memory. With type-inferred aliasing, the optimizer can propagate the copy of `d` and `f` and thus avoid two loads and two stores.

This option can be used for any application involving the use of pointers, where those pointers are type safe. To specify when a subset of types are type-safe, use the [NO]PTRS_STRONGLY_TYPED pragma. The compiler issues warnings for any incompatible pointer assignments that may violate the type-inferred aliasing rules discussed in "Aliasing Options" later in this chapter.

`+O[no]ptrs_strongly_typed` (continued)

Example 2: Unsafe Type Cast

Any type cast to a different type violates type-inferred aliasing rules. Do not use `+Optrs_strongly_typed` with code that has these "unsafe" type casts. Use the [NO]PTRS_STRONGLY_TYPED pragma to prevent the application of type-inferred aliasing to the unsafe type casts.

```
struct foo{
  int a;
  int b;
} *P;

struct bar {
  float a;
  int b;
  float c;
} *q;

P = (struct foo *) q;
  /* Incompatible pointer assignment
  through type cast */
```

Example 3: Generally Applying Type Aliasing

Dynamic cast is allowed with `+Optrs_strongly_typed` or `+Optrs_ansi`. A pointer dereference is called dynamic cast if a cast is applied on the pointer to a different type.

In the example below, type-inferred aliasing is applied on `P` generally, not just to the particular dereference. Type-aliasing will be applied to any other dereferences of P.

```
struct s {
  short int a;
  short int b;
  int c;
} *P
* (int *)P = 0;
```

For more information about type aliasing see the section "Aliasing Options" at the end of this chapter.

`+O[no]ptrs_to_globals[=`*name1, name2, ...nameN*`]`

Optimization level(s): 2, 3, 4

Default: `+Optrs_to_globals`

By default global variables are conservatively assumed to be modified anywhere in the program. Use this option to specify which global variables are not modified through pointers, so that the optimizer can make your program run more efficiently by incorporating copy propagation and common sub-expression elimination.

This option can be used to specify all global variables as not modified via pointers, or to specify a comma-separated list of global variables as not modified via pointers.

Note that the *on* state for this option disables some optimizations, such as aggressive optimizations on the program's global symbols.

For example, use the command line option `+Onoptrs_to_globals=a,b,c` to specify global variables `a`, `b`, and `c` as not being accessed through pointers. No pointer can access these global variables. The optimizer will perform copy propagation and constant folding because storing to `*p` will not modify `a` or `b`.

```
int a, b, c;
float *p;
foo()
{
   a = 10;
   b = 20;
  *p = 1.0;
   c = a + b;
}
```

If all global variables are unique, use the following option without listing the global variables:

```
+Onoptrs_to_globals
```

`+O[no]ptrs_to_globals` (continued)

In the example below, the address of `b` is taken. This means `b` can be accessed indirectly through the pointer. You can still use `+Onoptrs_to_globals` as: `+Onoptrs_to_globals +Optrs_to_globals=b`.

```
int b,c;
int *p;

p=&b;

foo()
```

For more information about type aliasing see the section "Aliasing Options" at the end of this chapter.

`+O[no]regionsched`

Optimization level(s): 2, 3, 4

Default: `+Onoregionsched`

Applies aggressive scheduling techniques to move instructions across branches. This option is incompatible with the linker `-z` option. If used with `-z`, it may cause a `SIGSEGV` error at run-time.

Use `+Oregionsched` to improve application run-time speed. Compilation time may increase.

`+O[no]regreassoc`

Optimization level(s): 2, 3, 4

Default: `+Oregreassoc`

If disabled, this option turns off register reassociation.

Use `+Onoregreassoc` to disable register reassociation if this optimization hinders the optimized application performance.

`+O[no]sideeffects=[`*name1, name2, ...nameN*`]`

Optimization level(s): 2, 3, 4

Default: assume all subprograms have side effects

Assume that subprograms specified in the *name* list might modify global variables. Therefore, when `+Osideeffects` is enabled the optimizer limits global variable optimization.

The default is to assume that all subprograms have side effects unless the optimizer can determine that there are none.

Use `+Onosideeffects` if you know that the named functions do not modify global variables and you wish to achieve the best possible performance.

`+O[no]signedpointers`

Optimization level(s): 0, 1, 2, 3, 4

Default: `+Onosignedpointers`

Perform [or do not perform] optimizations related to treating pointers as signed quantities. Applications that allocate shared memory and that compare a pointer to shared memory with a pointer to private memory may run incorrectly if this optimization is enabled.

Use `+Osignedpointers` to improve application run-time speed.

`+O[no]static_prediction`

Optimization level(s): 0, 1, 2, 3, 4

Default: `+Onostatic_prediction`

`+Ostatic_prediction` turns on static branch prediction for PA-RISC 2.0 targets.

PA-RISC 2.0 has two means of predicting which way conditional branches will go: dynamic branch prediction and static branch prediction. Dynamic branch prediction uses a hardware history mechanism to predict future executions of a branch from its last three executions. It is transparent and quite effective unless the hardware buffers involved are overwhelmed by a large program with poor locality.

With static branch prediction on, each branch is predicted based on implicit hints encoded in the branch instruction itself; the dynamic branch prediction is not used.

Static branch prediction's role is to handle large codes with poor locality for which the small dynamic hardware facility will prove inadequate.

Use `+Ostatic_prediction` to better optimize large programs with poor instruction locality, such as operating system and database code.

Use this option only when using PBO, as an amplifier to `+P`. It is allowed but silently ignored with `+I`, so makefiles need not change between the `+I` and `+P` phases.

`+O[no]vectorize`

Optimization level(s): 0, 1, 2, 3, 4

Default: `+Onovectorize`

`+Ovectorize` allows the compiler to replace certain loops with calls to vector routines.

Use `+Ovectorize` to increase the execution speed of loops.

When `+Onovectorize` is specified, loops are not replaced with calls to vector routines.

Because the `+Ovectorize` option may change the order of operations in an application, it may also change the results of those operations slightly. See the *HP-UX Floating-Point Guide* for details.

The math library contains special prefetching versions of vector routines. If you have a PA2.0 application that contains operations on very large arrays (larger than 1 megabyte in size), using `+Ovectorize` in conjunction with `+Odataprefetch` may improve performance substantially.

You may use `+Ovectorize` at levels 3 and 4. `+Onovectorize` is also included as part of `+Oaggressive` and `+Oall`.

This option is only valid for PA-RISC 1.1 and 2.0 systems.

`+O[no]volatile`

Optimization level(s): 1, 2, 3, 4

Default: `+Onovolatile`

The `+Ovolatile` option implies that memory references to global variables cannot be removed during optimization.

The `+Onovolatile` option implies that all globals are not of `volatile` class. This means that references to global variables *can be removed* during optimization.

The `+Ovolatile` option replaces the obsolete `+OV` option.

Use this option to control the `volatile` semantics for all global variables.

`+O[no]whole_program_mode`

Optimization level(s): 4

Default: `+Onowhole_program_mode`

The `+Owhole_program_mode` option enables the assertion that only the files that are compiled with this option directly reference any global variables and procedures that are defined in these files. In other words, this option asserts that there are no unseen accesses to the globals.

When this assertion is in effect, the optimizer can hold global variables in registers longer and delete inlined or cloned global procedures.

All files compiled with `+Owhole_program_mode` must also be compiled with `+O4`. If any of the files were compiled with `+O4` but were not compiled with `+Owhole_program_mode`, the linker disables the assertion for all files in the program.

The default, `+Onowhole_program_mode`, disables the assertion.

Use this option to increase performance speed, but only when you are certain that only the files compiled with `+Owhole_program_mode` directly access any globals that are defined in these files.

Using Advanced Optimization Options

Several advanced optimization options can be specified on the same command line. For example, the following command line specifies aggressive level 3 optimizations with unrestricted compile time, disables software pipelining, and disables moving conditional floating-point instructions out of a loop:

```
cc +O3 +Oaggressive +Onolimit +Onomoveflops +Onopipeline \
   sourcefile.c
```

Specify the level of optimization first (`+O1`, `+O2`, `+O3`, or `+O4`), followed by any +O[no]*optimization* options.

Level 1 Optimization Modules

The level 1 optimization modules are:

- Branch optimization.
- Dead code elimination.
- Faster register allocation.
- Instruction scheduler.
- Peephole optimization.

The examples in this section are shown at the source code level wherever possible. Transformations that cannot be shown at the source level are shown in assembly language. See Table 4-5 for descriptions of the assembly language instructions used.

Branch Optimization

The branch optimization module traverses the procedure and transforms branch instruction sequences into more efficient sequences where possible. Examples of possible transformations are:

- Deleting branches whose target is the fall-through instruction; that is, the target is two instructions away.
- When the target of a branch is an unconditional branch, changing the target of the first branch to be the target of the second unconditional branch.

- Transforming an unconditional branch at the bottom of a loop, branching to a conditional branch at the top of the loop, into a conditional branch at the bottom of the loop.
- Changing an unconditional branch to the exit of a procedure into an exit sequence where possible.
- Changing conditional or unconditional branch instructions that branch over a single instruction into a conditional nullification in the following instruction.
- Looking for conditional branches over unconditional branches, where the sense of the first branch could be inverted and the second branch deleted. These result from null `then` clauses and from `then` clauses that only contain `goto` statements. For example, the code:

```
            if(a) {
              :
               statement 1
            } else {
               goto L1;
            }
            statement 2
        L1:
```

becomes:

```
            if(!a) {
                goto L1;
            }
            statement 1
            statement 2
        L1:
```

Dead Code Elimination

The dead code elimination module removes unreachable code that is never executed.

For example, the code:

```
        if(0) {
           a = 1;
```

```
} else {
   a = 2;
```

becomes:

```
a = 2;
```

Faster Register Allocation

The faster register allocation module, used with unoptimized code, analyzes register use faster than the coloring register allocator (a level 2 module).

This module performs the following:

- Inserts entry and exit code.
- Generates code for operations such as multiplication and division.
- Eliminates unnecessary copy instructions.
- Allocates actual registers to the dummy registers in instructions.

Instruction Scheduler

The instruction scheduler module performs the following:

- Reorders the instructions in a basic block to improve memory pipelining. For example, where possible, a load instruction is separated from the use of the loaded register.
- Where possible, follows a branch instruction with an instruction that can be executed as the branch occurs.
- Schedules floating-point instructions.

For example, the code:

```
LDW     -52(0,30),r1
ADDI    3,r1,r31    ;interlock with load of r1
LDI     10,r19
```

becomes:

```
LDW     -52(0,sp),r1
LDI     10,r19
ADDI    3,r1,r31    ;use of r1 is now separated from load
```

Table 4-5. Descriptions of Assembly Language Instructions

Instruction	Description
`LDW` *offset(sr, base), target*	Loads a word from memory into register *target*.
`ADDI` *const, reg, target*	Adds the constant *const* to the contents of register *reg* and puts the result in register *target*.
`LDI` *const, target*	Loads the constant *const* into register *target*.
`LDO` *const(reg),target*	Adds the constant *const* to the contents of register *reg* and puts the result in register *target*.
`AND` *reg1, reg2, target*	Performs a bitwise AND of the contents of registers *reg1* and *reg2* and puts the result in register *target*.
`COMIB` *cond const, reg, lab*	Compares the constant *const* to the contents of register *reg* and branches to label *lab* if the condition *cond* is true.
`BB` *cond reg,num,lab*	Tests the bit number *num* in the contents of register *reg* and branches to label *lab* if the condition *cond* is true.
`COPY` *reg, target*	Copies the contents of register *reg* to register *target*.
`STW` *reg, offset(sr, base)*	Store the word in register *reg* to memory.

Peephole Optimizations

The peephole optimization process involves looking at small windows of machine code for optimization opportunities. Wherever possible, the peephole optimizer replaces assembly language instruction sequences with faster (usually shorter) sequences, and removes redundant register loads and stores.

For example, the code:

```
LDI     32,r3
AND     r1,r3,r2
COMIB,= 0,r2,L1
```

becomes:

```
BB,>=   r1, 26, L1
```

Level 2 Optimization Modules

Level 2 performs optimizations within each procedure. At level 2, the optimizer performs all optimizations performed at the prior level, with the following additions:

- FMAC synthesis.
- Coloring register allocation.
- Induction variable elimination and strength reduction.
- Local and global common subexpression elimination.
- Advanced constant folding and propagation. (Simple constant folding is done by level 0 optimization.)
- Loop invariant code motion.
- Store/copy optimization.
- Unused definition elimination.
- Software pipelining.
- Register reassociation.
- Loop unrolling.

The examples in this section are shown at the source code level wherever possible. Transformations that cannot be shown at the source level are shown in assembly language.

Coloring Register Allocation

The name of this optimization comes from the similarity to map coloring algorithms in graph theory. This optimization determines when and how long commonly used variables and expressions occupy a register. It minimizes the number of references to memory (loads and stores) a code segment makes. This can improve run-time speed.

You can help the optimizer understand when certain variables are heavily used within a function by declaring these variables with the `register` qualifier. The

first 10 `register` qualified variables encountered in the source are honored. You should pick the ten most important variables to be most effective.

The coloring register allocator may override your choices and promote to a register a variable not declared `register` over one that is, based on estimated speed improvements.

The following code shows the type of optimization the coloring register allocation module performs. The code:

```
LDI     2,r104
COPY    r104,r103
LDO     5(r103),r106
COPY    r106,r105
LDO     10(r105),r107
```

becomes:

```
LDI     2,r25
LDO     5(r25),r26
LDO     10(r26),r31
```

Induction Variables and Strength Reduction

The induction variables and strength reduction module removes expressions that are linear functions of a loop counter and replaces each of them with a variable that contains the value of the function. Variables of the same linear function are computed only once. This module also simplifies the function by replacing multiplication instructions with addition instructions wherever possible.

For example, the code:

```
for (i=0; i<25; i++) {
     r[i] = i * k;
}
```

becomes:

```
t1 = 0;
for (i=0; i<25; i++) {
     r[i] = t1;
     t1 += k;
```

```
}
```

Local and Global Common Subexpression Elimination

The common subexpression elimination module identifies expressions that appear more than once and have the same result, computes the result, and substitutes the result for each occurrence of the expression. The types of subexpression include instructions that load values from memory, as well as arithmetic evaluation.

For example, the code:

```
a = x + y + z;
b = x + y + w;
```

becomes:

```
t1 = x + y;
a = t1 + z;
b = t1 + w;
```

Constant Folding and Propagation

Constant folding computes the value of a constant expression at compile time. For example:

```
A = 10;
B = A + 5;
C = 4 * B;
```

can be replaced by:

```
A = 10;
B = 15;
C = 60;
```

Loop Invariant Code Motion

The loop invariant code motion module recognizes instructions inside a loop whose results do not change and moves them outside the loop. This ensures that the invariant code is only executed once.

For example, the code:

```
x = z;
for(i=0; i<10; i++)
{
     a[i] = 4 * x + i;
}
```

becomes:

```
x = z;
t1 = 4 * x;
for(i=0; i<10; i++)
{
     a[i] = t1 + i;
}
```

Store/Copy Optimization

Where possible, the store/copy optimization module substitutes registers for memory locations, by replacing store instructions with copy instructions and deleting load instructions.

For example, the following HP C code:

```
a = x + 23;     where a is a local variable
return a;
```

produces the following code for the unoptimized case:

```
LDO     23(r26),r1
STW     r1,-52(0,sp)
LDW     -52(0,sp),ret0
```

and this code for the optimized case:

```
LDO     23(r26),ret0
```

Unused Definition Elimination

The unused definition elimination module removes unused memory location and register definitions. These definitions are often a result of transformations made by other optimization modules.

For example, the function:

```
f(int x)
{
     int a,b,c:

     a = 1;
     b = 2;
     c = x * b;
     return c;
}
```

becomes:

```
f(int x)
{
     int a,b,c;

     b = 2;
     c = x * b;
     return c;
}
```

Software Pipelining

Software pipelining is a code transformation that optimizes program loops. It rearranges the order in which instructions are executed in a loop. It generates code that overlaps operations from different loop iterations. Software pipelining is useful for loops that contain arithmetic operations on floats and doubles.

The goal of this optimization is to avoid CPU stalls due to memory or hardware pipeline latencies. The software pipelining transformation adds code before and after the loop to achieve a high degree of optimization within the loop.

Example

The following pseudo-code fragment shows a loop before and after the software pipelining optimization. Four significant things happen:

- A portion of the first iteration of the loop is performed before the loop.
- A portion of the last iteration of the loop is performed after the loop.
- The loop is unrolled twice.
- Operations from different loop iterations are interleaved with each other.

The following is a C `for` loop:

```
#define SIZ 10000
float x[SIZ], y[SIZ]; \*Software pipelining works with*\
int i;                \*floats and doubles.           *\
init();
for (i = 0;i<= SIZ;i++);
        {
        x[i] =x[i] / y[i] + 4.00
        }
```

When this loop is compiled with software pipelining, the optimization can be expressed in pseudo-code as follows:

```
R1 = 0;                 Initialize array index.
R2 = 4.0;               Load constant value.
R3 = Y[0];              Load first Y value.
R4 = X[0];              Load first X value.
R5 = R4 / R3;           Perform division on first element:
                        n = X[0] / Y[0].

do {                    Begin loop.
      R6 = R1;          Save current array index.
      R1++;             Increment array index.
      R7 = X[R1];       Load current X value.
      R8 = Y[R1];       Load current Y value.
      R9 = R5 + R2;     Perform addition on prior row:
                        X[i] = n + 4.0.
```

```
        R10 = R7 / R8;      Perform division on current row:
                            m = X[i+1] / Y[i+1].
        X[R6] = R9;         Save result of operations on prior row.

        R6 = R1;            Save current array index.
        R1++;               Increment array index.
        R4 = X[R1];         Load next X value.
        R3 = Y[R1];         Load next Y value.
        R11 = R10 + R2;     Perform addition on current row:
                            X[i+1] = m + 4
        R5 = R4 / R3;       Perform division on next row:
                            n = X[i+2] / Y[i+2]
        X[R6] = R11         Save result of operations on current row.
} while (R1 <= 100);        End loop.

R9 = R5 + R2;               Perform addition on last row:
                            X[i+2] = n + 4
X[R6] = R9;                 Save result of operations on last row.
```

This transformation stores intermediate results of the division instructions in unique registers (noted as *n* and *m*). These registers are not referenced until several instructions after the division operations. This decreases the possibility that the long latency period of the division instructions will stall the instruction pipeline and cause processing delays.

Prerequisites of Pipelining

Software pipelining is attempted on a loop that meets the following criteria:

- It is the innermost loop.
- There are no branches or function calls within the loop.
- The loop is of moderate size.

This optimization produces slightly larger program files and increases compile time. It is most beneficial in programs containing loops that are executed a large number of times. This optimization is not recommended for loops that are executed only a small number of times.

Use the `+Onopipeline` option with the `+O2`, `+O3`, or `+O4` option to suppress software pipelining if program size is more important than execution speed. This will perform level two optimization, but disable software pipelining.

Register Reassociation

Array references often require one or more instructions to compute the virtual memory address of the array element specified by the subscript expression. The register reassociation optimization implemented in the PA-RISC compilers tries to reduce the cost of computing the virtual memory address expression for array references found in loops.

Within loops, the virtual memory address expression can be rearranged and separated into a loop varying term and a loop invariant term. Loop varying terms are those items whose values may change from one iteration of the loop to another. **Loop invariant terms** are those items whose values are constant throughout all iterations of the loop. The **loop varying term** corresponds to the difference in the virtual memory address associated with a particular array reference from one iteration of the loop to the next.

The register reassociation optimization dedicates a register to track the value of the virtual memory address expression for one or more array references in a loop and updates the register appropriately in each iteration of a loop.

The register is initialized outside the loop to the loop invariant portion of the virtual memory address expression and the register is incremented or decremented within the loop by the loop variant portion of the virtual memory address expression. On PA-RISC, the update of such a dedicated register can often be performed for "free" using the base-register modification capability of load and store instructions.

The net result is that array references in loops are converted into equivalent but more efficient pointer dereferences.

For example:

```
int a[10][20][30];

void example (void)
{
        int i, j, k;
```

```
          for (k = 0; k < 10; k++)
             for (j = 0; j < 10; j++)
                 for (i = 0; i < 10; i++)
                 {
                     a[i][j][k] = 1;
                 }
  }
```

after register reassociation is applied to the innermost loop becomes:

```
int a[10][20][30];

void example (void)
{
  int i, j, k;
  register int (*p)[20][30];

  for (k = 0; k < 10; k++)
    for (j = 0; j < 10; j++)
      for (p = (int (*)[20][30]) &a[0][j][k], i = 0; i < 10; i++)
      {
          *(p++[0][0]) = 1;
      }
}
```

In the above example, the compiler-generated temporary register variable, `p`, strides through the array `a` in the innermost loop. This register pointer variable is initialized outside the innermost loop and auto-incremented within the innermost loop as a side-effect of the pointer dereference.

Register reassociation can often enable another loop optimization. After performing the register reassociation optimization, the loop variable may be needed only to control the iteration count of the loop. If this is case, the original loop variable can be eliminated altogether by using the PA-RISC `ADDIB` and `ADDB` machine instructions to control the loop iteration count.

Level 3 Optimizations

Level 3 optimization includes level 2 optimizations, plus full optimization across all subprograms within a single file. Level 3 also inlines certain subprograms within the input file. Use `+O3` to get level 3 optimization.

Level 3 optimization produces faster run-time code than level 2 on code that frequently calls small functions within a file. Level 3 links faster than level 4.

Inlining within a Single Source File

Inlining substitutes functions calls with copies of the function's object code. Only functions that meet the optimizer's criteria are inlined. This may result in slightly larger executable files. However, this increase in size is offset by the elimination of time-consuming procedure calls and procedure returns.

Example of Inlining

The following is an example of inlining at the source code level. Before inlining, the source file looks like this:

```
/* Return the greatest common divisor of two positive integers,  */
/* int1 and int2, computed using Euclid's algorithm.  (Return 0  */
/* if either is not positive.)                                   */
int gcd(int1,int2)
  int int1;
  int int2;
{
  int inttemp;

    if ( ( int1 <= 0 ) || ( int2 <= 0 ) ) {
        return(0);
    }
    do {
        if ( int1 < int2 ) {
            inttemp = int1;
            int1    = int2;
            int2    = inttemp;
        }
```

```
        int1 = int1 - int2;
    } while (int1 > 0);
    return(int2);
}

main()
{
  int xval,yval,gcdxy;
    /* statements before call to gcd */
    gcdxy = gcd(xval,yval);
    /* statements after call to gcd */
}
```

After inlining, the source file looks like this:

```
   main()
   {
     int xval,yval,gcdxy;
       /* statements before inlined version of gcd */
       {
         int int1;
         int int2;

           int1 = xval;
           int2 = yval;
           {
             int inttemp;

               if ( ( int1 <= 0 ) || ( int2 <= 0 ) ) {
                   gcdxy = ( 0 );
                   goto AA003;
               }
               do {
                   if ( int1 < int2 ) {
                       inttemp = int1;
                       int1    = int2;
                       int2    = inttemp;
                   }
                   int1 = int1 - int2;
```

```
            } while ( int1 > 0 );
            gcdxy = ( int2 );
        }
    }
AA003 : ;
    /* statements after inlined version of gcd */
}
```

Level 4 Optimizations

Level 4 performs optimizations across all files in a program. At level 4, all optimizations of the prior levels are performed. Two additional optimizations are performed:

- Inlining across multiple source files.
- Global and static variable optimization.

Interprocedural global optimizations across all files within a program searches across function boundaries to produce better and faster code sequences. Normally, global optimizations are performed within individual functions or source code files. Interprocedural optimizations look at function interactions within a program and transform particular code sequences into faster code. Since information about every function within a program is required, this level of optimization must be performed at link time.

Inlining Across Multiple Files

Inlining at Level 4 is performed across all procedures within the program. Inlining at level 3 is done within one file.

Inlining substitutes function calls with copies of the function's object code. Only functions that meet the optimizer's criteria are inlined. This may result in slightly larger executable files. However, this increase in size is offset by the elimination of time-consuming procedure calls and procedure returns.

Global and Static Variable Optimization

Global and static variable optimizations look for ways to reduce the number of instructions required for accessing global and static variables. The compiler normally generates two machine instructions when referencing global variables. Depending on the locality of the global variables, single machine instructions may sometimes be used to access these variables. The linker rearranges the storage location of global and static data to increase the number of variables that can be referenced by single instructions.

Global Variable Optimization Coding Standards

Since this optimization rearranges the location and data alignment of global variables, *avoid* the following programming practices:

- Making assumptions about the relative storage location of variables, such as generating a pointer by adding an offset to the address of another variable.
- Relying on pointer or address comparisons between two different variables.
- Making assumptions about the alignment of variables, such as assuming that a short integer is aligned the same as an integer.

Guidelines for Using the Optimizer

The following guidelines help you effectively use the optimizer and write efficient HP C programs.

1. Use register variables where needed.
2. Hash table sizes should be in powers of 2; field sizes of variables should also be in powers of 2.
3. Where possible, use local variables to help the optimizer promote variables to registers.
4. When using `short` or `char` variables or bit-fields, it is more efficient to use unsigned variables rather than signed because a signed variable causes an extra instruction to be generated.

5. The code generated for a test for a loop termination is more efficient with a test against zero than for a test against some other value. Therefore, where possible, construct loops so the control variable increases or decreases towards zero.
6. `Do` loops and `for` loops are more efficient than `while` loops because opportunities for removing loop invariant code are greater.
7. Whenever possible, pass and return pointers to large `structs` instead of passing and returning large `structs` by value.
8. Do shift, multiplication, division, or remainder operations using constants instead of variables whenever possible.
9. Insure all local variables are initialized before they are used.
10. Use type checking tools like `lint` to help eliminate semantic errors.

Optimizer Assumptions

During optimization, the compiler gathers information about the use of variables and passes this information to the optimizer. The optimizer uses this information to ensure that every code transformation maintains the correctness of the program, at least to the extent that the original unoptimized program is correct.

When gathering this information, the HP C compiler makes the following assumption: while inside a function, the only variables that can be accessed indirectly through a pointer or by another function call are:

- Global variables, that is, all variables with file scope.
- Local variables that have had their addresses taken either explicitly by the `&` operator, or implicitly by the automatic conversion of array references to pointers.

  In general, you do not need to be concerned about this assumption. Standard conformant C programs do not violate this assumption. However, if you have code that does violate this assumption, the optimizer can change the behavior of the program in an undesired way. In particular, you should

avoid the following coding practices to ensure correct program execution for optimized code:

- Avoid referencing outside the bounds of an array.
- Avoid passing incorrect number of arguments to functions.
- Avoid accessing an array other than the one being subscripted. For example, the construct `a[b-a]` where `a` and `b` are the same type of array actually references the array `b`, because it is equivalent to `*(a+(b- a))`, which is equivalent to `*b`. Using this construct might yield unexpected optimization results.
- Avoid referencing outside the bounds of the objects a pointer is pointing to. All references of the form `*(p+i)` are assumed to remain within the bounds of the variable or variables that `p` was assigned to point to.
- Avoid using variables that are accessed by external processes. Unless a variable is declared with the `volatile` attribute, the compiler will assume that a program's data structures are accessed only by that program. Using the `volatile` attribute may significantly slow down a program.
- Avoid using local variables before they are initialized. When you request optimization level 2, 3, or 4, the compiler tries to detect and flag violations of this rule.
- Avoid relying on the memory layout scheme when manipulating pointers; incorrect optimizations may result. For example, if `p` is pointing to the first member of structure , it should not be assumed that `p+1` points to the second member of the structure. Another example: if `p` is pointing to the first in a list of declared variables, `p+1` should not be assumed to be pointing to the second variable in the list.

Optimizer Pragmas

Pragmas give you the ability to:

- Control compilation in finer detail than what is allowed by command line options.
- Give information about the program to the compiler.

Pragmas cannot cross line boundaries and the word `pragma` must be in lowercase letters. Optimizer pragmas may not appear inside a function.

Optimizer Control Pragmas

The `OPTIMIZE` and `OPT_LEVEL` pragmas control which functions are optimized, and which set of optimizations are performed. These pragmas can be placed before any function definitions and override any previous pragma. These pragmas cannot raise the optimization level above the level specified in the command line.

`OPT_LEVEL 0, 1,` and `2` provide more control over optimization than the `+O1` and `+O2` compiler options because these pragmas can be used to raise or lower optimization at a function by function level inside the source file using different levels for different functions. Whereas, the compiler options can only be used for an entire source file. (`OPT_LEVEL 3` and `4` can only be used at the beginning of the source file.)

Table 4-6 shows the possible combinations of options and pragmas and the resulting optimization levels. The level at which a function will be optimized is the lower of the two values specified by the command line optimization level and the optimization pragma in force.

Table 4-6. Optimization Level Precedence

Command-line Optimization Level	#Pragma OPT_LEVEL	Resulting OPT_LEVEL
none	OFF	0
none	1	0
none	2	0
+O1	OFF	0
+O1	1	1
+O1	2	1
+O1	3	1
+O1	4	1
+O2	OFF	0
+O2	1	1
+O2	2	2
+O2	3	2
+O2	4	2
+O3	OFF	0
+O3	1	1
+O3	2	2
+O3	3	3
+O3	4	3
+O4	OFF	0
+O4	1	1
+O4	2	2
+O4	3	3
+O4	4	4

The values of `OPTIMIZE` and `OPT_LEVEL` are summarized in Table 4-7.

Table 4-7. Optimizer Control Pragmas

Pragma	Description
`#pragma OPTIMIZE ON`	Turns optimization on.
`#pragma OPTIMIZE OFF`	Turns optimization off.
`#pragma OPT_LEVEL 1`	Optimize only within small blocks of code
`#pragma OPT_LEVEL 2`	Optimize within each procedure.
`#pragma OPT_LEVEL 3`	Optimize across all procedures within a source file.
`#pragma OPT_LEVEL 4`	Optimize across all procedures within a program.

Inlining Pragmas

When `INLINE` is specified without a *functionname*, any function can be inlined. When specified with *functionname(s)*, these functions are candidates for inlining.

The `NOINLINE` pragma disables inlining for all functions or specified *functionname(s)*.

The syntax for performing inlining is:

```
#pragma INLINE [functionname(1), ..., functionname(n)]
#pragma NOINLINE [functionname(1), ..., functionname(n)]
```

For example, to specify inlining of the two subprograms `checkstat` and `getinput`, use:

```
#pragma INLINE checkstat, getinput
```

To specify that an infrequently called routine should not be inlined when compiling at optimization level 3 or 4, use:

```
#pragma NOINLINE opendb
```

See the related `+O[no]inline` optimization option.

Alias Pragmas

The compiler gathers information about each function (such as information about function calls, variables, parameters, and return values) and passes this information to the optimizer. The `NO_SIDE_EFFECTS` and `ALLOCS_NEW_MEMORY` pragma tell the optimizer to make assumptions it can not normally make, resulting in improved compile-time and run-time speed. They change the default information the compiler collects.

If used, the `NO_SIDE_EFFECTS` and `ALLOCS_NEW_MEMORY` pragmas should appear before the first function defined in a file and are in effect for the entire file. When used appropriately, these optional pragmas provide better optimization.

NO_SIDE_EFFECTS Pragma

By default, the optimizer assumes that all functions *might* modify global variables. To some degree, this assumption limits the extent of optimizations it can perform on global variables. The `NO_SIDE_EFFECTS` directive provides a way to override this assumption. If you know for certain that some functions do *not* modify global variables, you can gain further optimization of code containing calls to these functions by specifying the function names in this directive.

`NO_SIDE_EFFECTS` has the following form:

`#pragma NO_SIDE_EFFECTS` *functionname(1), ..., functionname(n)*

All functions in *functionname* are the names of functions that do not modify the values of global variables. Global variable references can be optimized to a greater extent in the presence of calls to the listed functions. Note that you need the `NO_SIDE_EFFECTS` directive in the files *where the calls are made*, not where the function is defined. This directive takes effect from the line it first occurs on to the end of the file.

ALLOCS_NEW_MEMORY pragma

The `ALLOCS_NEW_MEMORY` pragma states that the function *functionname* returns a pointer to *new* memory that it either allocates or a routine that it calls allocates. `ALLOCS_NEW_MEMORY` has the following form:

`#pragma ALLOCS_NEW_MEMORY` *functionname(1), ..., functionname(n)*

The *new* memory must be memory that was either newly allocated or was previously freed and is now reallocated. For example, the standard routines `malloc()` and `calloc()` satisfy this requirement.

Large applications might have routines that are layered *above* `malloc()` and `calloc()`. These interface routines make the calls to `malloc()` and `calloc()`, initialize the memory, and return the pointer that `malloc()` or `calloc()` returns. For example, in the program below:

```
struct_type *get_new_record(void)
   {
   struct_type *p;

   if ((p=malloc(sizeof(*p))) == NULL) {
        printf("get_new_record():out of memory\n");
        abort();
        }
   else {
        /* initialize the struct */
        .
        .
        .
        return p;
        }
```

the routine `get_new_record` falls under this category, and can be included in the `ALLOCS_NEW_MEMORY` pragma.

FLOAT_TRAPS_ON pragma

Informs the compiler that the function(s) may enable floating-point trap handling. When the compiler is so informed, it will not perform loop invariant code motion (LICM) on floating-point operations in the function(s) named in the pragma. This pragma is required for proper code generation when floating-point traps are enabled.

```
#pragma FLOAT_TRAPS_ON { functionname,...functionname
                       { _ALL
```

For example:

```
#pragma FLOAT_TRAPS_ON xyz,abc
```

informs the compiler and optimizer that `xyz` and `abc` have floating-point traps turned on and therefore LICM optimization should not be performed.

[NO]PTRS_STRONGLY_TYPED Pragma

The PTRS_STRONGLY_TYPED pragma allows you to specify when a subset of types are type-safe. This provides a finer lever of control than `+O[no]ptrs_strongly_typed`.

```
#pragma PTRS_STRONGLY_TYPED BEGIN

#pragma PTRS_STRONGLY_TYPED END

#pragma NOPTRS_STRONGLY_TYPED BEGIN

#pragma NOPTRS_STRONGLY_TYPED END
```

Any types that are defined between the begin-end pair are taken to apply type-safe assumptions. These pragmas are not allowed to nest. For each `BEGIN` an associated `END` must be defined in the compilation unit.

The pragma will take precedence over the command line option. Although, sometimes both are required (see example 2).

Example 1

```
double *d;
#pragma PTRS_STRONGLY_TYPED BEGIN
int *i;
float *f;
#pragma PTRS_STRONGLY_TYPED END
main(){
.......................
}
```

In this example only two types, pointer-to-int and pointer-to-float will be assumed to be type-safe.

Example 2

```
cc +Optrs_strongly_typed foo.c

/*source for Ex.2 */
double *d;
  ...
#pragma NOPTRS_STRONGLY_TYPED BEGIN
int *i;
float *f;
#pragma NOPTRS_STRONGLY_TYPED END
  ...
main(){
  ...
}
```

In this example all types are assumed to be type-safe except the types bracketed by pragma `NOPTRS_STRONGLY_TYPED`. The command line option is required because the default option is `+Onoptrs_strongly_typed`.

Aliasing Options

To be conservative, the optimizer assumes that a pointer can point to any object in the entire application. Instead, if the optimizer can be educated on the application's pointer usage, then the optimizer can generate more efficient code, due to the elimination of some false assumptions. Such behavior can be communicated to the optimizer by using the following options:

- `+O[no]ptrs_strongly_typed`
- `+O[no]ptrs_to_globals[=list]`
- `+O[no]global_ptrs_unique[=list]`
- `+O[no]ptrs_ansi`

where list is a comma-separated list of global variable names.

Here are the type-inferred aliasing rules:

- Type-aliasing optimizations are based on the assumption that pointer dereferences obey their declared types.
- A C variable is considered *address-exposed* if and only if the address of that variable is assigned to another variable or passed to a function as an actual parameter. In general, address-exposed objects are collected into a separate group based on their declared type. Global variables and static variables are considered address-exposed by default. Local variables and actual parameters are considered address-exposed only if their address has been computed using the address operator &.
- Dereferences of pointers to a certain type will be assumed to only alias with the corresponding equivalent group. An equivalent group includes all the address exposed objects of the same type. The dereferences of pointers are also assumed to alias with other pointer dereferences associated with the same equivalent group.

  In the example

  ```
  int *p, *q;
  ```

  `*p` and `*q` are assumed to alias with any objects of type `int`. Also `*p` and `*q` are assumed to alias with each other.
- Signed/Unsigned type distinctions are ignored in grouping objects into an equivalent group. Likewise, `long` and `int` types are considered to map to the same equivalent group. However, the `volatile` type qualifier is considered significant in grouping objects into equivalent groups (e.g., a pointer to `int` will not be considered to alias with a `volatile int` object).
- If two type names reduce to the same type, they are considered synonymous.

  In the following example, both types `type_old` and `type_new` will reduce to the same type, `struct foo`.

  ```
  typedef struct foo_st type_old;
  typedef type_old type_new;
  ```
- Each field of a structure type is placed in a separate equivalent group which is distinct from the equivalent group of the field's base type. (The assumption here is that a pointer to `int` will not be assigned the address of a structure field whose type is `int`). The actual type name of a structure type is not considered significant in constructing equivalent groups (e.g.,

dereferences of a `struct foo` pointer and a `struct bar` pointer will be assumed to alias with each other even if `struct foo` and `struct bar` have identical field declarations).

- All fields of a union type are placed in the same equivalent group, which is distinct from the equivalent group of any of the field's base types. (Thus, all dereferences of pointers to a particular union type will be assumed to alias with each other, regardless of which union field is being accessed.)
- Address-exposed array variables are grouped into the equivalent group of the array element type.
- Explicit pointer typecasts applied to expression values will be honored in that it would alter the equivalent group associated with an ensuing use of the typecast expression value. For example, an `int` pointer that is first typecast into a float pointer and then dereferenced will be assumed to potentially access objects in the float equivalent group—and not the int equivalent group. However, type-incompatible assignments to pointer variables will not alter the aliasing assumptions on subsequent references of such pointer variables.

  In general, type incompatible assignments can potentially invalidate some of the type-safe assumptions, and such constructs may elicit compiler warning messages.

Note Variables declared to be of type `void *` need to be typecast into a pointer to a specific type before they can be dereferenced.

Parallel Execution

This section provides information on parallel execution.

Transforming Eligible Loops for Parallel Execution (+Oparallel)

The `+Oparallel` option causes the compiler to transform eligible loops for parallel execution on multiprocessor machines.

If you link separately from the compile line and you compiled with the `+Oparallel` option, you must link with the `cc` command and specify the `+Oparallel` option to link in the right startup files and runtime support.

When a program is compiled with the `+Oparallel` option, the compiler looks for opportunities for parallel execution in loops and generates parallel code to execute the loop on the number of processors set by the `MP_NUMBER_OF_THREADS` environment variable discussed below. By default, this is the number of processors on the executing machine.

For a discussion of parallelization, including how to use the `+Oparallel` option, see "Parallelizing C Programs" below. For more detail on `+Oparallel`, see the description in "Controlling Specific Optimizer Features" earlier in this chapter.

Environment Variable for Parallel Programs

The environment variable `MP_NUMBER_OF_THREADS` is available for use with parallel programs.

MP_NUMBER_OF_THREADS

The `MP_NUMBER_OF_THREADS` environment variable enables you to set the number of processors that are to execute your program in parallel. If you do not set this variable, it defaults to the number of processors on the executing machine.

On the C shell, the following command sets `MP_NUMBER_OF_THREADS` to indicate that programs compiled for parallel execution can execute on two processors:

```
setenv MP_NUMBER_OF_THREADS 2
```

If you use the Korn shell, the command is:

```
export MP_NUMBER_OF_THREADS=2
```

Parallelizing C Programs

The following sections discuss how to compile C programs for parallel execution and inhibitors to parallelization.

Compiling Code for Parallel Execution

The following command lines compile (without linking) three source files: `x.c`, `y.c`, and `z.c`. The files `x.c` and `y.c` are compiled for parallel execution. The file `z.c` is compiled for serial execution, even though its object file will be linked with `x.o` and `y.o`.

```
cc +O3 +Oparallel -c x.c y.c
cc +O3 -c z.c
```

The following command line links the three object files, producing the executable file `para_prog`:

```
cc +O3 +Oparallel -o para_prog x.o y.o z.o
```

As this command line implies, if you link and compile separately, you must use `cc`, not `ld`. The command line to link must also include the `+Oparallel` and `+O3` options in order to link in the right startup files and runtime support.

Note To ensure the best performance from a parallel program, do not run more than one parallel program on a multiprocessor machine at the same time. Running two or more parallel programs simultaneously or running one parallel program on a heavily loaded system, will slow performance. You should run a parallel-executing program at a higher priority than any other user program; see *rtprio*(1) for information about setting real-time priorities.

HP-UX 10.20 users: At runtime, compiler-inserted code performs a check to determine if the call is to a system routine or to a user-defined routine with the same name as a system routine. If the call is to a system routine, the code inhibits parallel execution. If your program makes explicit use of threads, do not attempt to parallelize it.

Profiling Parallelized Programs

Profiling a program that has been compiled for parallel execution is performed in much the same way as it is for non-parallel programs:

1. Compile the program with the `-G` option.
2. Run the program to produce profiling data.
3. Run `gprof` against the program.
4. View the output from `gprof`.

The differences are:

- Running the program in Step 2 produces a `gmon.out` file for the master process and `gmon.out.1`, `gmon.out.2`, and so on for each of the slave processes. Thus, if your program is to execute on two processors, Step 2 will produce two files, `gmon.out` and `gmon.out.1`.
- The flat profile that you view in Step 4 indicates loops that were parallelized with the following notation:

  routine_name`##pr_line_0123`

  where *routine_name* is the name of the routine containing the loop, `pr` (parallel region) indicates that the loop was parallelized, and `0123` is the line number of the beginning of the loop or loops that are parallelized.

Conditions Inhibiting Loop Parallelization

The following sections describe different conditions that can inhibit parallelization.

Additionally, `+Onoloop_transform` and `+Onoinline` may be helpful options if you experience any problem while using `+Oparallel`.

Calling Routines with Side Effects. The compiler will not parallelize any loop containing a call to a routine that has side effects. A routine has side effects if it does any of the following:

- Modifies its arguments
- Modifies an `extern` or `static` variable
- Redefines variables that are local to the calling routine

- Performs I/O
- Calls another subroutine or function that does any of the above

Indeterminate Iteration Counts. If the compiler determines that a runtime determination of a loop's iteration count cannot be made before the loop starts to execute, the compiler will not parallelize the loop. The reason for this precaution is that the runtime code must know the iteration count in order to know how many iterations to distribute to the different processors for execution.

The following conditions can prevent a runtime count:

- The loop is an infinite loop.
- A conditional break statement or `goto` out of the loop appears in the loop.
- The loop modifies either the loop-control or loop-limit variable.
- The loop is a `while` construct and the condition being tested is defined within the loop.

Data Dependence. When a loop is parallelized, the iterations are executed independently on different processors, and the order of execution will differ from the serial order that occurs on a single processor. This effect of parallelization is not a problem. The iterations could be executed in any order with no effect on the results. Consider the following loop:

```
for (i=0; i<5; i++)
    a[i] = a[i] * b[i]
```

In this example, the array `a` would always end up with the same data regardless of whether the order of execution were 0-1-2-3-4, 4-3-2-1-0, 3-1-4-0-2, or any other order. The independence of each iteration from the others makes the loop eligible candidate for parallelization.

Such is not the case in the following:

```
for (i=1; i<5; i++)
    a[i] = a[i-1] * b[i]
```

In this loop, the order of execution does matter. The data used in iteration `i` is *dependent* upon the data that was produced in the previous iteration [`i-1`]. `a` would end up with very different data if the order of execution were any other

than 1-2-3-4. The data dependence in this loop thus makes it ineligible for parallelization.

Not all data dependences must inhibit parallelization. The following paragraphs discuss some of the exceptions.

Nested Loops and Matrices. Some nested loops that operate on matrices may have a data dependence in the inner loop only, allowing the outer loop to be parallelized. Consider the following:

```
for (i=0; i<10; i++)
    for (j=1; j<100; j++)
 a[i][j] = a[i][j-1] + 1;
```

The data dependence in this nested loop occurs in the inner [j] loop: Each row access of `a[j][i]` depends upon the preceding row [`j-1`] having been assigned in the previous iteration. If the iterations of the `j` loop were to execute in any other order than the one in which they would execute on a single processor, the matrix would be assigned different values. The inner loop, therefore, must not be parallelized.

But no such data dependence appears in the outer loop: Each column access is independent of every other column access. Consequently, the compiler can safely distribute entire columns of the matrix to execute on different processors; the data assignments will be the same regardless of the order in which the columns are executed, so long as each executes in serial order.

Assumed Dependences. When analyzing a loop, the compiler will err on the safe side and assume that what looks like a data dependence really is one and so not parallelize the loop. Consider the following:

```
for (i=100; i<200; i++)
    a[i] = a[i-k];
```

The compiler will assume that a data dependence exists in this loop because it appears that data that has been defined in a previous iteration is being used in a later iteration. However, if the value of `k` is 100, the dependence is assumed rather than real because `a[i-k]` is defined outside the loop.

5

Programming for Portability

The syntax of C is well defined as a result of the efforts of the ANSI X3J11 Technical Committee. The standard C function libraries are rich with features that isolate programs from operating system specific function calls. These factors make C programs highly portable between various combinations of hardware platforms and operating systems.

The C programming language was first described in *The C Programming Language*, by Brian Kernighan and Dennis Ritchie. This original language definition has proven powerful enough to provide the functionality that programmers need. The HP C compiler supports this language definition, including some Berkeley Software Distribution (BSD) extensions.

S.C. Johnson developed the Portable C Compiler (pcc) that became available on a wide range of systems, including the VAX and the HP 9000 Series 300/400 computers. The syntax and semantics of HP C are closely compatible with those of pcc.

In December, 1989, the American National Standards Institute (ANSI) approved a standard for the C programming language. The ANSI standard clarified a number of areas that were ambiguous and tended to vary among C compilers. It gave full specifications for the required library and codified a number of extensions that have been added to C over the years. (ANSI mode first became available with release 7.0 on the Series 800, release 7.40 on Series 300/400, and release 8.05 on Series 700. Compatibility mode supports the C syntax and semantics of previous releases.)

The ANSI standard specifies which aspects of C are required to work the same on conforming implementations, and which can work differently. Since many ANSI-conforming compilers are available on a wide variety of platforms, it is easy to develop portable programs. HP C, when invoked in ANSI mode and used with the preprocessor (`cpp`), headers, libraries, and linker, conforms fully with the standard.

Portable C programs are clear, reliable, and easily maintainable and can be easily transported from one machine to another. With few modifications, C programs written with portability in mind can be recompiled and run on different computers. For specific information on system dependencies, refer to the *HP C/HP-UX Reference Manual*.

The ANSI C standard document *American National Standard for Information Systems - Programming Language C, ANSI/ISO 9899-1990* contains complete details on the language including an appendix with a comprehensive list of portability issues. This document is available from ANSI at 11 West 42nd Street, New York City, New York, 10036, telephone (212) 642-4900.

This chapter discusses some guidelines for making your C programs more portable. Emphasis is placed on HP C specific portability issues, especially as they relate to porting from pre-ANSI mode HP C (Kernighan and Ritchie plus BSD extensions) to ANSI mode HP C.

Guidelines for Portability

This section lists some things you can do to make your HP C programs more portable.

- Use the ANSI C compiler option whenever possible when writing new programs. HP C conforms to the standard when it is invoked with the `-Aa` option. The `-w` and `+e` options should not be used with the `-Aa` option, as these options will suppress warning messages and allow non-conforming extensions.
- When you recompile existing programs, try compiling in ANSI mode. ANSI C mandates more thorough error checking, so portability problems are more likely to be flagged by the compiler in this mode. (Bugs are also more likely to be caught.) Many existing programs will compile and execute correctly in ANSI mode with few or no changes.
- Pay attention to all warnings produced by the compiler. Most warnings represent potentially problematic program constructs. You should consider warnings to be portability risks.

- For an additional level of warnings, compile with the `+w1` option. Pay particular attention to the warnings that mention "ANSI migration" issues. These identify most program constructs that are legal but are likely to work differently between pre-ANSI and ANSI compilers.
- Consult the detailed listing of diagnostic messages in the HP C Reference Manual for more information on how to correct each problem. For most messages, a reference to the relevant section of the ANSI standard is also given.
- On HP-UX, use `lint`, the C program syntax checker, to detect potential portability problems in your program. The `lint` utility also produces warnings about poor style, lack of efficiency, and inconsistency within the program.
- Use the `#define, #if,` and `#ifdef` preprocessing directives and `typedef` declarations to isolate any necessary machine or operating system dependencies.
- Declare all variables with the correct types. For example, functions and parameters default to `int`. On many implementations, pointers and integers are the same size, and the defaults work correctly. However, for maximum portability, the correct types should be used.
- Use only the standard C library functions.
- Code bit manipulation algorithms carefully to gain independence from machine-specific representations of numeric values. For example, use `x & ~3` instead of `x & 0xFFFFFFFC` to mask the low-order 2 bits to zero.
- Avoid absolute addressing.

Examples

The following example illustrates some ways to program for portability. In this example, the include files `IEEE.h` and `floatX.h` isolate machine-dependent portions of the code. These include files use the `#define` and `typedef` mechanisms to define macro constants and type definitions in the main body of the program.

The main program `fmult.c` uses the `#ifdef` preprocessor command to include `floatX.h` by default. If the option `-D IEEE_FLOAT` is passed to the

compiler, and subsequently the preprocessor, the program will use the IEEE representation for the structure `float_rep` rather than a machine-dependent representation.

Partial contents of the file `IEEE.h`:

```
#define FLT_MAX  3.4028235E38
#define PLUS_INFINITY  0X7F800000
#define MINUS_INFINITY 0XFF800000
typedef struct {
    unsigned sign : 1;
    unsigned exp : 8;
    unsigned mant : 23;
} FLOAT_REP;
#define EXP_BIAS 127
⋮
```

Partial contents of the file `floatX.h`:

```
#define FLT_MAX  1.70141E38
#define PLUS_INFINITY  0X7FFFFFFE
#define MINUS_INFINITY 0XFFFFFFFE
typedef struct {
    unsigned sign : 1;
    unsigned mant : 23;
    unsigned exp : 7;
    unsigned exp_sign : 1;
} FLOAT_REP;
#define EXP_BIAS 0
⋮
```

Partial contents of the file `fmult.c`:

```
#ifdef IEEE_FLOAT
#include "IEEE.h"
#else
#include "floatX.h"
#endif
union  {
    float f;
    FLOAT_REP f_rep;
    FLOAT_INT f_int;
} float_num;
float f_mult(float val1, float val2)
{
    if (val1 > 1.0F && val2 >1.0F) {
        if (val1 > FLT_MAX/val2 ||
            val2 > FLT_MAX/val1) {
            float_num.f_int = PLUS_INFINITY;

            return float_num.f;
        }
    ⋮
```

Practices to Avoid

To make a program portable, you need to minimize machine dependencies. The following are programming practices you should *avoid* to ensure portability:

- Using dollar signs (`$`) in identifiers.
- Using underscores (`_`) as the first character in an identifier.
- Using sized enumerations.
- Reliance on implicit expression evaluation order.
- Making assumptions regarding storage allocation and layout.
- Dependence on the number of significant characters in an identifier. Identifiers should differ as early as possible in the name. ANSI C requires that the first 31 characters of an internal name are significant. Only the first 6 characters of an external name are required to be significant by ANSI C.
- Dereferencing null pointers.
- Dependence on pointer representation.
- Dependence on being able to dereference a pointer to an object that is not correctly aligned.
- Dependence on the ability to store a pointer in a variable of type `int`.
- Dependence on case distinctions in external names.
- Dependence on `char` being signed or unsigned.
- Dependence on bitwise operations in signed integers.
- Dependence on bit-fields of any type except `int`, `unsigned int`, or `signed int`.
- Dependence on the sign of the remainder in integer division.
- Dependence on right shifts of negative signed values.
- Dependence on more than six declarators modifying a basic type.
- Dependence on values of automatic variables after a `longjmp` call when the values were changed between the `setjmp` and `longjmp` calls.

- Dependence on being able to call `setjmp` within an arbitrarily complex expression.
- Dependence on file system characteristics.
- Dependence on string literals being modifiable.
- Dependence on `extern` declarations within a block being visible outside of the block.

General Portability Considerations

This section summarizes some of the general considerations to take into account when writing portable HP C programs. Some of the features listed here *may* be different on other implementations of C. Differences between Series 300/400 versus workstations and servers implementations are also noted in this section.

Data Type Sizes and Alignments

Table 2-1 in Chapter 2 shows the sizes and alignments of the C data types on the different architectures.

Differences in data alignment can cause problems when porting code or data between systems that have different alignment schemes. For example, if you write a C program on Series 300/400 that writes records to a file, then read the file using the same program on HP 9000 workstations and servers, it may not work properly because the data may fall on different byte boundaries within the file due to alignment differences. To help alleviate this problem, HP C provides the `HP_ALIGN` pragma, which forces a particular alignment scheme, regardless of the architecture on which it is used. The `HP_ALIGN` pragma is described in Chapter 2.

Accessing Unaligned Data

The HP 9000 workstations and servers like all PA-RISC processors require data to be accessed from locations that are aligned on multiples of the data size. The C compiler provides an option to access data from misaligned addresses using code sequences that load and store data in smaller pieces, but this option will increase code size and reduce performance. A bus error handling routine is also available to handle misaligned accesses but can reduce performance severely if used heavily.

Here are your specific alternatives for avoiding bus errors:

1. Change your code to eliminate misaligned data, if possible. This is the only way to get maximum performance, but it may be difficult or impossible to do. The more of this you can do, the less you'll need the next two alternatives.
2. Use the `+u`*bytes* compiler option available at 9.0 to allow 2-byte alignment. However, the `+u`*bytes* option, as noted above, creates big, slow code compared to the default code generation which is able to load a double precision number with one 8-byte load operation. Refer to the *HP C/HP-UX Reference Manual* for more information.
3. Finally, you can use `allow_unaligned_data_access()` to avoid alignment errors. `allow_unaligned_data_access()` sets up a signal handler for the SIGBUS signal. When the SIGBUS signal occurs, the signal handler extracts the unaligned data from memory byte by byte.

   To implement, just add a call to `allow_unaligned_data_access()` within your main program *before* the first access to unaligned data occurs. Then link with `-lhppa`. Any alignment bus errors that occur are trapped and emulated by a routine in the `libhppa.a` library in a manner that will be transparent to you. The performance degradation will be significant, but if it only occurs in a few places in your program it shouldn't be a big concern.

Whether you use alternative 2 or 3 above depends on your specific code.

The `+u`*bytes* option costs significantly less per access than the handler, but it costs you on every access, whether your data is aligned or not, and it can make your code quite a bit bigger. You should use it selectively if you can isolate the routines in your program that may be exposed to misaligned pointers.

There is a performance degradation associated with alternative 3 because each unaligned access has to trap to a library routine. You can use the `unaligned_access_count` variable to check the number of unaligned accesses in your program. If the number is fairly large, you should probably use 2. If you only occasionally use a misaligned pointer, it is probably better just use the `allow_unaligned_data_access` handler. There is a stiff penalty per bus error, but it doesn't cause your program to fail and it won't cost you anything when you operate on aligned data.

The following is a an example of its use within a C program:

```
extern int unaligned_access_count;
                  /* This variable keeps a count
                     of unaligned accesses. */

char arr[]="abcdefgh";
char *cp, *cp2;
int i=99, j=88, k;
int *ip;          /* This line would normally result in a
                     bus error on workstations or servers */
main()
{
   allow_unaligned_data_access();
   cp = (char *)&i;
   cp2 = &arr[1];
   for (k=0; k<4; k++)
       cp2[k] = * (cp+k);
   ip = (int *)&arr[1];
   j = *ip;
   printf("%d\n", j);
   printf("unaligned_access_count is : %d\n", unaligned_access_count);
}
```

To compile and link this program, enter

```
cc filename.c -lhppa
```

This enables you to link the program with `allow_unaligned_data_access()` and the `int unaligned_access_count` that reside in `/usr/lib/libhppa.a`.

Note that there is a performance degradation associated with using this library since each unaligned access has to trap to a library routine. You can use the `unaligned_access_count` variable to check the number of unaligned accesses in your program. If the number is fairly large, you should probably use the compiler option.

Checking for Alignment Problems with lint

If invoked with the `-s` option, the `lint` command generates warnings for C constructs that may cause portability and alignment problems between Series 300/400 and Series 9000 workstations and servers, and vice versa. Specifically, `lint` checks for these cases:

- Internal padding of structures. `lint` checks for instances where a structure member may be aligned on a boundary that is inappropriate according to the most-restrictive alignment rules. For example, given the code

  ```
  struct s1 { char c; long l; };
  ```

  `lint` issues the warning:

  ```
  warning:  alignment of struct 's1' may not be portable
  ```

- Alignment of structures and simple types. For example, in the following code, the nested `struct` would align on a 2-byte boundary on Series 300/400 and an 8-byte boundary on HP 9000 workstations and servers:

  ```
  struct s3 { int i; struct { double d; } s; };
  ```

  In this case, `lint` issues this warning about alignment:

  ```
  warning:  alignment of struct 's3' may not be portable
  ```

- End padding of structures. Structures are padded to the alignment of the most-restrictive member. For example, the following code would pad to a 2-byte boundary on Series 300/400 and a 4-byte boundary for HP 9000 workstations and servers:

```
struct s2 { int i; short s; };
```

  In this case, `lint` issues the warning:

```
warning:  trailing padding of struct/union 's2' may not be portable
```

Note that these are only *potential* alignment problems. They would cause problems only when a program writes raw files which are read by another system. This is why the capability is accesible only through a command line option; it can be switched on and off.

`lint` does not check the layout of bit-fields.

Ensuring Alignment without Pragmas

Another solution to alignment differences between systems would be to define structures in such a way that they are forced into the same layout on different systems. To do this, use **padding** bytes—that is, dummy variables that are inserted solely for the purpose of forcing `struct` layout to be uniform across implementations. For example, suppose you need a structure with the following definition:

```
struct S {
    char   c1;
    int    i;
    char   c2;
    double d;
};
```

An alternate definition of this structure that uses filler bytes to ensure the same layout on Series 300/400 and workstations and servers would look like this:

```
struct S {
    char   c1;                  /* byte 0 */
    char   pad1,pad2,pad3;      /* bytes 1 through 3 */
    int    i;                   /* bytes 4 through 7 */
    char   c2;                  /* byte 8 */
    char   pad9,pad10,pad11,    /* bytes 9 */
           pad12,pad13,pad14,   /*  through */
           pad15;               /*    15 */
    double d;                   /* bytes 16 through 23 */
};
```

Casting Pointer Types

Before understanding how casting pointer types can cause portability problems, you must understand how HP 9000 workstations and servers align data types. In general, a data type is aligned on a byte boundary equivalent to its size. For example, the `char` data type can fall on any byte boundary, the `int` data type must fall on a 4-byte boundary, and the `double` data type must fall on an 8-byte boundary. A valid *location* for a data type would then satisfy the following equation:

location mod `sizeof(`*data_type*`)` $== 0$

Consider the following program:

```
#include <string.h>
#include <stdio.h>
main()
{
  struct chStruct {
    char ch1;                   /* aligned on
                                   an even boundary */
    char chArray[9];            /* aligned on
                                   an odd byte boundary */
  } foo;
```

```
    int *bar;                       /* must be aligned
                                       on a word boundary */

    strcpy(foo.chArray, "1234");    /* place a value
                                       in the ch array */
    bar = (int *) foo.chArray;      /* type cast */
    printf("*bar = %d\n",*bar);     /* display the value */
}
```

Casting a smaller type (such as `char`) to a larger type (such as `int`) will not cause a problem. However, casting a `char*` to an `int*` and then dereferencing the `int*` may cause an alignment fault. Thus, the above program crashes on the call to `printf()` when `bar` is dereferenced.

Such programming practices are inherently non-portable because there is no standard for how different architectures reference memory. You should try to avoid such programming practices.

As another example, if a program passes a casted pointer to a function that expects a parameter with stricter alignment, an alignment fault may occur. For example, the following program causes an alignment fault on the HP 9000 workstations and servers:

```
void main (int argc, char *argv[])
{
    char   pad;
    char   name[8];

intfunc((int *)&name[1]);
}

int intfunc (int *iptr)
{
    printf("intfunc got passed %d\n", *iptr);
}
```

Type Incompatibilities and typedef

The C `typedef` keyword provides an easy way to write a program to be used on systems with different data type sizes. Simply define your own type equivalent to a provided type that has the size you wish to use.

For example, suppose system A implements `int` as 16 bits and `long` as 32 bits. System B implements `int` as 32 bits and `long` as 64 bits. You want to use 32 bit integers. Simply declare all your integers as type `INT32`, and insert the appropriate `typedef` on system A:

```
typedef long INT32;
```

The code on system B would be:

```
typedef int INT32;
```

Conditional Compilation

Using the `#ifdef` C preprocessor directive and the predefined symbols `__hp9000s300`, `__hp9000s700`, and `__hp9000s800`, you can group blocks of system-dependent code for conditional compilation, as shown below:

```
#ifdef  __hp9000s300
    :
  Series 300/400-specific code goes here...
    :
#endif

#ifdef  __hp9000s700
    :
  Series 700-specific code goes here...
    :
#endif

#ifdef  __hp9000s800
    :
  Series 700/800-specific code goes here...
    :
#endif
```

If this code is compiled on a Series 300/400 system, the first block is compiled; if compiled on a Series 700 system, the second block is compiled; if compiled on *either* the Series 700 or Series 800, the third block is compiled. You can use this feature to ensure that a program will compile properly on either Series 300/400 or workstations or servers.

If you want your code to compile *only* on the Series 800 but not on the 700, surround your code as follows:

```
#if (defined(__hp9000s800) && !defined(__hp9000s700))
    &vellipsis;
  Series 800-specific code goes here...
    &vellipsis;
#endif
```

Isolating System-Dependent Code with #include Files

`#include` files are useful for isolating the system-dependent code like the type definitions in the previous section. For instance, if your type definitions were in a file `mytypes.h`, to account for all the data size differences when porting from system A to system B, you would only have to change the contents of file `mytypes.h`. A useful set of type definitions is in `/usr/include/model.h`.

Note If you use the symbolic debugger, `xdb`, `include` files used within `union`, `struct`, or `array` initialization will generate correct code. However, such use is discouraged because `xdb` may show incorrect debugging information about line numbers and source file numbers.

Parameter Lists

On the Series 300/400, parameter lists grow towards higher addresses. On the HP 9000 workstations and servers, parameter lists are usually stacked towards decreasing addresses (though the stack itself grows towards higher addresses). The compiler may choose to pass some arguments through registers for efficiency; such parameters will have no stack location at all.

ANSI C function prototypes provide a way of having the compiler check parameter lists for consistency between a function declaration and a function

call within a compilation unit. `lint` provides an option (`-Aa`) that flags cases where a function call is made in the absence of a prototype.

The ANSI C `<stdarg.h>` header file provides a portable method of writing functions that accept a variable number of arguments. You should note that `<stdarg.h>` supersedes the use of the `varargs` macros. `varargs` is retained for compatibility with the pre-ANSI compilers and earlier releases of HP C/HP-UX. See *varargs*(5) and *vprintf*(3S) for details and examples of the use of `varargs`.

The char Data Type

The `char` data type defaults to signed. If a `char` is assigned to an `int`, sign extension takes place. A `char` may be declared `unsigned` to override this default. The line:

```
unsigned char   ch;
```

declares one byte of unsigned storage named `ch`. On some non-HP-UX systems, `char` variables are unsigned by default.

Register Storage Class

The `register` storage class is supported on Series 300/400 and workstation and servers, and if properly used, can reduce execution time. Using this type should not hinder portability. However, its usefulness on systems will vary, since some ignore it. Refer to the *HP-UX Assembler and Supporting Tools* for Series 300/400 for a more complete description of the use of the `register` storage class on Series 300/400.

Also, the `register` storage class declarations are ignored when optimizing at level 2 or greater on all Series.

Identifiers

To guarantee portable code to non-HP-UX systems, the ANSI C standard requires identifier names without external linkage to be significant to 31 case-sensitive characters. Names with external linkage (identifiers that are defined in another source file) will be significant to six case-insensitive

characters. Typical C programming practice is to name variables with all lower-case letters, and `#define` constants with all upper case.

Predefined Symbols

The symbol `__hp9000s300` is predefined on Series 300/400; the symbols `__hp9000s800` and `__hppa` are predefined on Series 700/800; and `__hp9000s700` is predefined on Series 700 only. The symbols `__hpux` and `__unix` are predefined on all HP-UX implementations.

This is only an issue if you port code to or from systems that also have predefined these symbols.

Shift Operators

On left shifts, vacated positions are filled with 0. On right shifts of signed operands, vacated positions are filled with the sign bit (arithmetic shift). Right shifts of unsigned operands fill vacated bit positions with 0 (logical shift). Integer constants are treated as signed unless cast to unsigned. Circular shifts are not supported in any version of C. Shifts greater than 32 bits give an undefined result.

The sizeof Operator

The `sizeof` operator yields an `unsigned int` result, as specified in section 3.3.3.4 of the ANSI C standard (X3.159-1989). Therefore, expressions involving this operator are inherently unsigned. Do not expect any expression involving the `sizeof` operator to have a negative value (as may occur on some other systems). In particular, logical comparisons of such an expression against zero may not produce the object code you expect as the following example illustrates.

```
main()
{
       int i;
       i = 2;
 if ((i-sizeof(i)) < 0)          /* sizeof(i) is 4,
                                    but unsigned! */
  printf("test less than 0\n");
 else
  printf("an unsigned expression cannot be less than 0\n");
}
```

When run, this program will print

```
an unsigned expression cannot be less than 0
```

because the expression `(i-sizeof(i))` is unsigned since one of its operands is unsigned (`sizeof(i)`). By definition, an unsigned number cannot be less than 0 so the compiler will generate an unconditional branch to the `else` clause rather than a test and branch.

Bit-Fields

The ANSI C definition does not prescribe bit-field implementation; therefore each vendor can implement bit-fields somewhat differently. This section describes how bit-fields are implemented in HP C.

Bit-fields are assigned from most-significant to least-significant bit on all HP-UX and Domain systems.

On all HP-UX implementations, bit-fields can be `signed` or `unsigned`, depending on how they are declared.

On the Series 300/400, a bit-field declared without the `signed` or `unsigned` keywords will be `signed` in ANSI mode and `unsigned` in compatibility mode by default.

On the workstations and servers, plain `int`, `char`, or `short` bit-fields declared without the `signed` or `unsigned` keywords will be `signed` in both compatibility mode and ANSI mode by default.

On the HP 9000 workstations and servers, and for the most part on the Series 300/400, bit-fields are aligned so that they cannot cross a boundary of the declared type. Consequently, some padding within the structure may be required. As an example,

```
struct foo
{
        unsigned int   a:3, b:3, c:3, d:3;
        unsigned int   remainder:20;
};
```

For the above `struct`, `sizeof(struct foo)` would return 4 (bytes) because none of the bit-fields straddle a 4 byte boundary. On the other hand, the following `struct` declaration will have a larger size:

```
struct foo2
{
        unsigned char   a:3, b:3, c:3, d:3;
        unsigned int    remainder:20;
};
```

In this `struct` declaration, the assignment of data space for `c` must be aligned so it doesn't violate a byte boundary, which is the normal alignment of `unsigned char`. Consequently, two undeclared bits of padding are added by the compiler so that `c` is aligned on a byte boundary. `sizeof(struct foo2)` returns 6 (bytes) on Series 300/400, and 8 on workstations and servers. Note, however, that on Domain systems or when using `#pragma HP_ALIGN NATURAL`, which uses Domain bit-field mapping, 4 is returned because the `char` bit-fields are considered to be `int`s.)

Bit-fields on HP-UX systems cannot exceed the size of the declared type in length. The largest possible bit-field is 32 bits. All scalar types are permissible to declare bit-fields, including `enum`.

`Enum` bit-fields are accepted on all HP-UX systems. On Series 300/400 in compatibility mode they are implemented internally as unsigned integers. On workstations and servers, however, they are implemented internally as signed integers so care should be taken to allow enough bits to store the sign as well as the magnitude of the enumerated type. Otherwise your results may be unexpected. In ANSI mode, the type of `enum` bit-fields is `signed int` on *all* HP-UX systems.

Floating-Point Exceptions

HP C on workstations and servers, in accordance with the IEEE standard, does not trap on floating point exceptions such as division by zero. By contrast, when using HP C on Series 300/400, floating-point exceptions will result in the run-time error message `Floating exception (core dumped)`. One way to handle this error on workstations and servers is by setting up a signal handler using the `signal` system call, and trapping the signal `SIGFPE`. For details, see *signal*(2), *signal*(5), and "Advanced HP-UX Programming" in *HP-UX Linker and Libraries Online User Guide*.

For full treatment of floating-point exceptions and how to handle them, see *HP-UX Floating-Point Guide*.

Integer Overflow

In HP C, as in nearly every other implementation of C, integer overflow does not generate an error. The overflowed number is "rolled over" into whatever bit pattern the operation happens to produce.

Overflow During Conversion from Floating Point to Integral Type

HP-UX systems will report a `floating exception - core dumped` at run time if a floating point number is converted to an integral type and the value is outside the range of that integral type. As with the error described previously under "Floating Point Exceptions," a program to trap the floating-point exception signal (`SIGFPE`) can be used. See *signal*(2) and *signal*(5) for details.

Structure Assignment

The HP-UX C compilers support structure assignment, structure-valued functions, and structure parameters. The structs in a `struct` assignment `s1=s2` must be declared to be the same `struct` type as in:

```
struct s  s1,s2;
```

Structure assignment is in the ANSI standard. Prior to the ANSI standard, it was a BSD extension that some other vendors may not have implemented.

Structure-Valued Functions

Structure-valued functions support storing the result in a structure:

```
s = fs();
```

All HP-UX implementations allow direct field dereferences of a structure-valued function. For example:

```
x = fs().a;
```

Structure-valued functions are ANSI standard. Prior to the ANSI standard, they were a BSD extension that some vendors may not have implemented.

Dereferencing Null Pointers

Dereferencing a null pointer has never been defined in any C standard. Kernighan and Ritchie's *The C Programming Language* and the ANSI C standard both warn against such programming practice. Nevertheless, some versions of C permit dereferencing null pointers.

Dereferencing a null pointer returns a zero value on all HP-UX systems. The workstations and servers C compiler provides the `-z` compile line option, which causes the signal `SIGSEGV` to be generated if the program attempts to read location zero. Using this option, a program can "trap" such reads.

Since some programs written on other implementations of UNIX rely on being able to dereference null pointers, you may have to change code to check for a null pointer. For example, change:

```
if (*ch_ptr != '\0')
```

to:

```
if ((ch_ptr != NULL) && *ch_ptr != '\0')
```

Writes of location zero may be detected as errors even if reads are not. If the hardware cannot assure that location zero acts as if it was initialized to zero or is locked at zero, the hardware acts as if the `-z` flag is always set.

Expression Evaluation

The order of evaluation for some expressions will differ between HP-UX implementations. This does not mean that operator precedence is different. For instance, in the expression:

```
x1 = f(x) + g(x) * 5;
```

`f` may be evaluated before or after `g`, but `g(x)` will always be multiplied by `5` before it is added to `f(x)`. Since there is no C standard for order of evaluation of expressions, you should avoid relying on the order of evaluation when using functions with side effects or using function calls as actual parameters. You should use temporary variables if your program relies upon a certain order of evaluation.

Variable Initialization

On some C implementations, `auto` (non-`static`) variables are implicitly initialized to 0. This is *not* the case on HP-UX and it is most likely not the case on other implementations of UNIX. *Don't depend on the system initializing your local variables*; it is not good programming practice in general and it makes for nonportable code.

Conversions between unsigned char or unsigned short and int

All HP-UX C implementations, when used in compatibility mode, are **unsigned preserving**. That is, in conversions of `unsigned char` or `unsigned short` to `int`, the conversion process first converts the number to an `unsigned int`. This contrasts to some C implementations that are **value preserving** (that is, `unsigned char` terms are first converted to `char` and then to `int` before they are used in an expression).

Consider the following program:

```
main()
{
 int i = -1;
 unsigned char uc = 2;
 unsigned int ui = 2;

 if (uc > i)
  printf("Value preserving\n");
 else
  printf("Unsigned preserving\n");
 if (ui < i)
  printf("Unsigned comparisons performed\n");
}
```

On HP-UX systems in compatibility mode, the program will print:

```
Unsigned preserving
Unsigned comparisons performed
```

In contrast, ANSI C specifies value preserving; so in ANSI mode, all HP-UX C compilers are value preserving. The same program, when compiled in ANSI mode, will print:

```
Value preserving
Unsigned comparisons performed
```

Temporary Files ($TMPDIR)

All HP-UX C compilers produce a number of intermediate temporary files for their private use during the compilation process. These files are normally invisible to you since they are created and removed automatically. If, however, your system is tightly constrained for file space these files, which are usually generated on `/tmp` or `/usr/tmp`, may exceed space requirements. By assigning another directory to the `TMPDIR` environment variable you can redirect these temporary files. See the `cc` manual page for details.

Input/Output

Since the C language definition provides no I/O capability, it depends on library routines supplied by the host system. Data files produced by using the HP-UX calls *write*(2) or *fwrite*(3) should not be expected to be portable between different system implementations. Byte ordering and structure packing rules will make the bits in the file system-dependent, even though identical routines are used. When in doubt, move data files using ASCII representations (as from *printf*(3)), or write translation utilities that deal with the byte ordering and alignment differences.

Checking for Standards Compliance

In order to check for standards compliance to a particular standard, you can use the `lint` program with one of the following `-D` options:

- `-D_XOPEN_SOURCE`
- `-D_POSIX_SOURCE`

For example, the command

```
lint -D_POSIX_SOURCE file.c
```

checks the source file `file.c` for compliance with the POSIX standard.

If you have the HP Advise product, you can also check for C standard compliance using the `apex` command.

Porting to ANSI Mode HP C

This section describes porting non-ANSI mode HP C programs to ANSI C. Specifically, it discusses:

- Compile line options.
- ANSI C name spaces.
- Differences that can lead to porting problems.

ANSI Mode Compile Option (-Aa)

To compile in ANSI C mode, use the `-Aa` compile time option.

By default, beginning at the HP-UX 10.30 operating system release, HP C compilers use `-Ae`.

The `-w` and `+e` options should not be used at compile time for true ANSI compliance. These options suppress warning messages and allow HP C extensions that are not ANSI conforming.

HP C Extensions to ANSI C (+e)

There are a number of HP C extensions enabled by the `+e` option in ANSI mode:

- Long pointers.
- Dollar sign character `$` in an identifier.
- Compiler supplied defaults for missing arguments to intrinsic calls (For example `FOPEN(`*"filename",fopt,,rsize*`)`, where `,,` indicates that the missing *aopt* parameter is automatically supplied with default values.)
- Sized enumerated types: `char enum`, `short enum`, `int enum`, and `long enum`.
- Long long integer type. Note, the `long long` data type is only available in HP 9000 workstations and servers, including workstations and servers.

These are the only HP C extensions that require using the `+e` option.

When coding for portability, you should compile your programs without the `+e` command line option, and rewrite code that causes the compiler to generate messages related to HP C extensions.

const and volatile Qualifiers

HP C supports the ANSI C `const` and `volatile` keywords used in variable declarations. These keywords qualify the way in which the compiler treats the declared variable.

The `const` qualifier declares variables whose values do not change during program execution. The HP C compiler generates error messages if there is an attempt to assign a value to a `const` variable. The following declares a constant variable `pi` of type `float` with an initial value of `3.14`:

```
const float pi = 3.14;
```

A `const` variable can be used like any other variable. For example:

```
area = pi * (radius * radius);
```

But attempting to assign a value to a `const` variable causes a compile error:

```
pi = 3.1416;  /* This causes an error. */
```

Only obvious attempts to modify `const` variables are detected. Assignments made using pointer references to `const` variables may not be detected by the compiler.

However, pointers may be declared using the `const` qualifier. For example:

```
char *const prompt = "Press return to continue> ";
```

An attempt to reassign the `const` pointer `prompt` causes a compiler error. For example:

```
prompt = "Exiting program."; /* Causes a compile time error. */
```

The `volatile` qualifier provides a way to tell the compiler that the value of a variable may change in ways not known to the compiler. The `volatile` qualifier is useful when declaring variables that may be altered by signal handlers, device drivers, the operating system, or routines that use shared memory. It may also prevent certain optimizations from occurring.

The optimizer makes assumptions about how variables are used within a program. It assumes that the contents of memory will not be changed by entities other than the current program. The `volatile` qualifier forces the compiler to be more conservative in its assumptions regarding the variable.

The `volatile` qualifier can also be used for regular variables and pointers. For example:

```
volatile int intlist[100];
volatile char *revision_level;
```

For further information on the HP C optimizer and its assumptions, see Chapter 4, "Optimizing HP C Programs." For further information on the `const` and `volatile` qualifiers see the *HP C/UX Reference Manual*.

ANSI Mode Function Prototypes

Function prototypes are function declarations that contain parameter type lists. Prototype-style function declarations are available only in ANSI mode. You are encouraged to use the prototype-style of function declarations.

Adding function prototypes to existing C programs yields three advantages:

- Better type checking between declarations and calls because the number and types of the parameters are part of the function's parameter list. For example:

```
struct s
  {
   int i;
  }
int old_way(x)
  struct s x;
  {
  /* Function body using the old method for
     declaring function parameter types
   */
  }
  int new_way(struct s x)
    {
    /* Function body using the new method for
       declaring function parameter types
     */
    }
    /* The functions "old_way" and "new_way" are
       both called later on in the program.
    */
    old_way(1);  /* This call compiles without complaint. */
    new_way(1);  /* This call gives an error. */
```

  In this example, the function `new_way` gives an error because the value being passed to it is of type `int` instead of type `struct x`.

- More efficient parameter passing in some cases. Parameters of type `float` are not converted to `double`. For example:

```
void old_way(f)
  float f;
  {
   /* Function body using the old method for
      declaring function parameter types
    */
  }
void new_way(float f)
  {
   /* Function body using the new method for
      declaring function parameter types
   */
  }
/* The functions "old_way" and "new_way" are
   both called later on in the program.
*/
float g;

old_way(g);
new_way(g);
```

In the above example, when the function `old_way` is called, the value of `g` is converted to a `double` before being passed. In ANSI mode, the `old_way` function then converts the value back to `float`. When the function `new_way` is called, the float value of `g` is passed without conversion.

- Automatic conversion of function arguments, as if by assignment. For example, integer parameters may be automatically converted to floating point.

```
/* Function declaration using the new method
   for declaring function parameter types
*/

extern double sqrt(double);

/* The function "sqrt" is called later
   on in the program.
*/
```

```
sqrt(1);
```

In this example, any value passed to `sqrt` is automatically converted to `double`.

Compiling an existing program in ANSI mode yields some of these advantages because of the existence of prototypes in the standard header files. To take full advantage of prototypes in existing programs, change old-style declarations (without prototype) to new style declarations. On HP-UX, the tool `protogen` (see *protogen(1)* in the on-line man pages) helps add prototypes to existing programs. For each source file, `protogen` can produce a header file of prototypes and a modified source file that includes prototype declarations.

Mixing Old-Style Function Definitions with ANSI Function Declarations

A common pitfall when mixing prototypes with old-style function definitions is to overlook the ANSI rule that for parameter types to be compatible, the parameter type in the prototype must match the parameter type resulting from default promotions applied to the parameter in the old-style function definition.

For example:

```
void func1(char c);
void func1(c)
char c;
{ }
```

gets the following message when compiled in ANSI mode:

```
Inconsistent parameter list declaration for "func1"
```

The parameter type for `c` in the prototype is `char`. The parameter type for `c` in the definition `func1` is also `char`, but it expects an `int` because it is an old-style function definition and in the absence of a prototype, `char` is promoted to `int`.

Changing the prototype to:

```
void func1(int c);
```

fixes the error.

The ANSI C standard does not require a compiler to do any parameter type checking if prototypes are not used. Value parameters whose sizes are larger

than 64 bits (8 bytes) will be passed via a short pointer to the high-order byte of the parameter value. The receiving function then makes a copy of the parameter pointed to by this short pointer in its own local memory.

Function Prototype Considerations

There are three things to consider when using function prototypes:

- Type differences between actual and formal parameters.
- Declarations of a structure in a prototype parameter.
- Mixing of `const` and `volatile` qualifiers and function prototypes.

Type Differences between Actual and Formal Parameters

When a prototype to a function is added, be careful that all calls to that function occur with the prototype visible (in the same context). The following example illustrates problems that can arise when this is not the case:

```
func1(){
  float f;
  func2(f);
}

int func2(float arg1){
    /* body of func2 */
}
```

In the example above, when the call to `func2` occurs, the compiler behaves as if `func2` had been declared with an old-style declaration `int func2()`. For an old-style call, the default argument promotion rules cause the parameter `f` to be converted to `double`. When the declaration of `func2` is seen, there is a conflict. The prototype indicates that the parameter `arg1` should not be converted to `double`, but the call in the absence of the prototype indicates that `arg1` should be widened. When this conflict occurs within a single file, the compiler issues an error:

```
Inconsistent parameter list declaration for "func2".
```

This error can be fixed by either making the prototype visible before the call, or by changing the formal parameter declaration of `arg1` to `double`. If the declaration and call of `func2` were in separate files, then the compiler would not detect the mismatch and the program would silently behave incorrectly.

On HP-UX, the *lint(1)* command can be used to find such parameter inconsistencies across files.

Declaration of a Structure in a Prototype Parameter

Another potential prototype problem occurs when structures are declared within a prototype parameter list. The following example illustrates a problem that may arise:

```
func3(struct stname *arg);
struct stname { int i; };

void func4(void) {
   struct stname s;
   func3(&s);
}
```

In this example, the call and declaration of `func3` are not compatible because they refer to different structures, both named `stname`. The `stname` referred by the declaration was created within prototype scope. This means it goes out of scope at the end of the declaration of `func3`. The declaration of `stname` on the line following `func3` is a new instance of `struct stname`. When conflicting structures are detected, the compiler issues an error:

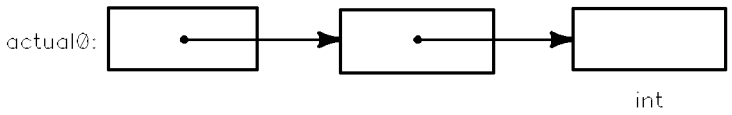
```
types in call and definition of 'func3' have incompatible
struct/union pointer types for parameter 'arg'
```

This error can be fixed by switching the first two lines and thus declaring `struct stname` prior to referencing it in the declaration of `func3`.

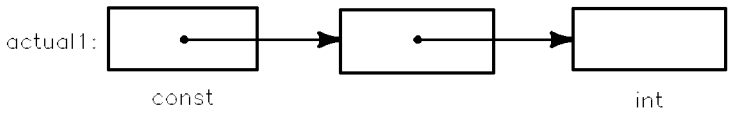
Mixing of const and volatile Qualifiers and Function Prototypes

Mixing the `const` and `volatile` qualifiers and prototypes can be tricky. Note that this section uses the `const` qualifier for all of its examples; however, you could just as easily substitute the `volatile` qualifier for `const`. The rules for prototype parameter passing are the same as the rules for assignments. To illustrate this point, consider the following declarations:

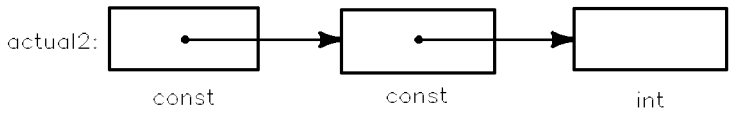
```
/* pointer to pointer to int */
int **actual0;
```



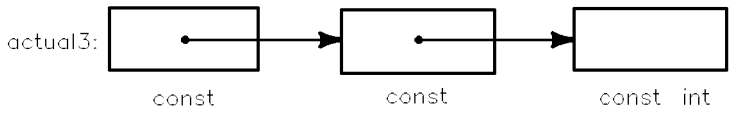

```
/* const pointer to pointer to int */
int **const actual1;
```




```
/* const pointer to const pointer to int */
int *const *const actual2;
```




```
/* const pointer to const pointer to const int */
const int *const *const actual3;
```



These declarations show how successive levels of a type may be qualified. The declaration for `actual0` has no qualifiers. The declaration of `actual1` has only the top level qualified. The declarations of `actual2` and `actual3` have two and three levels qualified. When these actual parameters are substituted into calls to the following functions:

```
void f0(int **formal0);
void f1(int **const formal1);
void f2(int *const *const formal2);
void f3(const int *const *const formal3);
```

The compatibility rules for pointer qualifiers are different for all three levels. At the first level, the qualifiers on pointers are ignored. At the second level, the qualifiers of the formal parameter must be a superset of those in the actual parameter. At levels three or greater the parameters must match exactly. Substituting `actual0` through `actual3` into `f0` through `f3` results in the following compatibility matrix:

	f0	f1	f2	f3
`actual0`	C	C	C	N
`actual1`	C	C	C	N
`actual2`	S	S	C	N
`actual3`	NS	NS	N	C

C = compatible

S = not compatible, qualifier level two of formal is not a superset of actual parameter

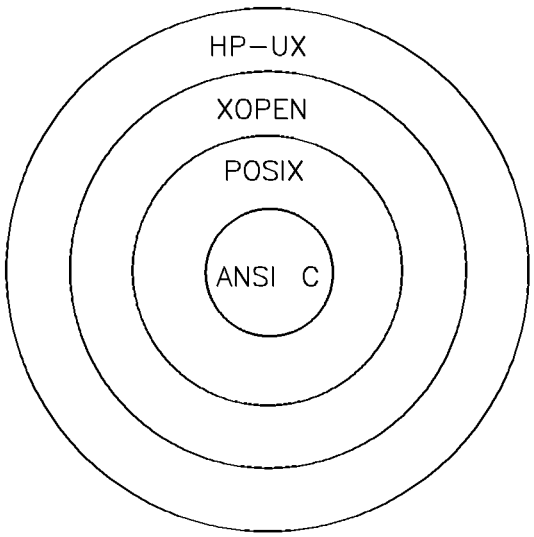
N = not compatible, qualifier level three doesn't match

Using Name Spaces in HP C and ANSI C

The ANSI standard specifies exactly which names (for example, variable names, function names, type definition names) are reserved. The intention is to make it easier to port programs from one implementation to another without unexpected collisions in names. For example, since the ANSI C standard does not reserve the identifier `open`, an ANSI C program may define and use a function named `open` without colliding with the *open(2)* system call in different operating systems.

HP Header File and Library Implementation of Name Spaces

The HP header files and libraries have been designed to support several different name spaces. On HP-UX systems, four name spaces are available:



The HP library implementation has been designed with the assumption that many existing programs will use more routines than those allowed by the ANSI C standard.

If a program calls, but does not define, a routine that is not in the ANSI C name space (for example, `open`), then the library will resolve that reference. This allows a clean name space and backward compatibility.

The HP header file implementation uses preprocessor conditional compilation directives to select the name space. In non-ANSI mode, the default is the HP-UX name space. Compatibility mode means that virtually all programs that compiled and executed under previous releases of HP C on HP-UX continue to work as expected. The following table provides information on how to select a name space from a command line or from within a program using the defined libraries.

Table 5-1. Selecting a Name Space in ANSI Mode

When using the name space ...	Use command line option ...	or #define in source program	Platform
HP-UX	`-D_HPUX_SOURCE`	`#define _HPUX_SOURCE`	HP-UX Only
XOPEN	`-D_XOPEN_SOURCE`	`#define _XOPEN_SOURCE`	HP-UX Only
POSIX	`-D_POSIX_SOURCE`	`#define _POSIX_SOURCE`	HP-UX
ANSI C	default	default	HP-UX

In ANSI mode, the default is ANSI C name space. The macro names `_POSIX_SOURCE`, `_XOPEN_SOURCE`, and `_HPUX_SOURCE` may be used to select other name spaces. The name space may need to be relaxed to make existing programs compile in ANSI mode. This can be accomplished by defining the `_HPUX_SOURCE` macro definition.

For example, in HP-UX:

```
#include <sys/types.h>
#include <sys/socket.h>
```

results in the following compile-time error in ANSI mode because `socket.h` uses the symbol `u_short` and `u_short` is only defined in the HP-UX name space section of `types.h`:

```
"/usr/include/sys/socket.h", line 79: syntax error:
   u_short sa_family;
```

This error can be fixed by adding `-D_HPUX_SOURCE` to the command line of the compile.

Silent Changes for ANSI C

Non-ANSI mode HP C is different from ANSI mode HP C in ways that generally go unnoticed. On HP-UX, many of these silent differences can be found by running the *lint(1)* program. The following list provides some of these silent changes:

- Trigraphs are new in ANSI C. A trigraph is a three character sequence that is replaced by a corresponding single character. For example, `??=` is replaced by `#`. For more information on trigraphs, refer to "Preprocessing Directives" in the *HP C/HP-UX Reference Manual*.
- Promotion rules for `unsigned char` and `unsigned short` have changed. Non-ANSI mode rules specify when an `unsigned char` or `unsigned short` is used with an integer the result is `unsigned`. ANSI mode rules specify the result is `signed`. The following program example illustrates a case where these rules differ:

```
main(){
  unsigned short us = 1;
  int i = -2;
  printf("%s\n",(i+us)>0 ? "non-ANSI mode" : "ANSI mode");
}
```

Note that differences in promotion rules can occur under the following conditions:[1]

- An expression involving an `unsigned char` or `unsigned short` produces an integer-wide result in which the sign bit is set: that is, either a unary operation on such a type, or a binary operation in which the other operand is `int` or a "narrower" type.
- The result of the preceding expression is used in a context in which its condition of being signed is significant: it is the left operand of the right-shift operator or either operand of /,%,<,<=,>, or >=.

- Floating-point expressions with `float` operands may be computed as `float` precision in ANSI mode. In non-ANSI mode they will always be computed in `double` precision.
- Initialization rules are different in some cases when braces are omitted in an initialization.
- Unsuffixed integer constants may have different types. In non-ANSI mode, unsuffixed constants have type `int`. In the ANSI mode, unsuffixed constants less than or equal to 2147483647 have type `int`. Constants larger than 2147483647 have type `unsigned`. For example:

  ```
  -2147483648
  ```

  has type `unsigned` in the ANSI mode and `int` in non-ANSI mode. The above constant is `unsigned` in the ANSI mode because `2147483648` is `unsigned`, and the `-` is a unary operator.
- Empty tag declarations in a *block scope* create a new `struct` instance in ANSI mode. The term **block scope** refers to identifiers declared inside a block or list of parameter declarations in a function definition that have meaning from their point of declaration to the end of the block. In the ANSI mode, it is possible to create recursive structures within an inner block. For example:

[1] *Rationale for Proposed American National Standard for Information Systems - Programming Language C* (311 First Street, N.W., Suite 500, Washington, DC 20001-2178; X3 Secretariat: Computer and Business Equipment Manufacturers Association), pages 34-35

```
struct x { int i; };
{ /* inner scope */
  struct x;
  struct y { struct x *xptr; };
  struct x { struct y *yptr; };
}
```

In ANSI mode, the inner `struct x` declaration creates a new version of the structure type which may then be referred to by `struct y`. In non-ANSI mode, the `struct x;` declaration refers to the outer structure.

- On Series workstations and servers, variable shifts (`<<` or `>>`) where the right operand has a value greater than 31 or less than 0 will no longer always have a result of 0. For example,

```
unsigned int i,j = 0xffffffff, k = 32;
i = j >> k;  /* i gets the value 0 in compatibility mode,  */
             /* 0xffffffff(-1) in ANSI mode.  */
```

Porting between HP C and Domain/C

All HP-UX and Domain computers have ANSI C compilers. Strictly standard-compliant programs are highly portable between all these architectures.

The following Domain/C extensions are *not* supported on HP-UX in compatibility mode and in most cases, are *not* supported in ANSI mode either:

- Reference variables.
- The following preprocessor directives: `#attribute`, `#options`, `#section`, `#module`, `#debug`, `#eject`, `#list`, `#nolist`, and `#systype`.
- `std_$call`.
- `__attribute` modifier and `__options` specifier.
- `systype` predefined macro.
- `_BFMT__COFF` predefined macro.
- `_ISP__M68K` predefined macro.
- `_ISP__A88K` predefined macro.
- `_ISP__PA_RISC` predefined macro.
- Partial specification of `struct` and `union` members.

Function prototypes, `struct` and `union` initialization, and the predefined names `__DATE__` and `__TIME__`, all of which are ANSI C features, are supported on HP-UX in ANSI mode.

Compile line options are different between HP-UX C and Domain/C. Check the respective *cc*(1) page for complete descriptions.

There are other differences between HP-UX C and Domain/C:

- Alignment: All Domain workstations have hardware or software assists to handle misaligned data. Programs that rely on these features will not run on the Series 800.
- Floating-point exceptions: All Domain workstations, by default, enable invalid operation, divide by zero, and overflow exception traps. Programs that rely on fault detection, for instance, to enter a fault handler or to terminate execution on encountering a fault, will ordinarily generate useless

output on HP-UX. However, the PA1.1 math library for the workstations and servers provides a function `fpsetdefaults`(3M), which enables these traps and therefore allows such programs to run as expected. For more information, see the *HP-UX Floating-Point Guide*.

- `struct` layout and alignment, especially bit-field, is different.
- `float` data type: Domain/C optimizes a statement all of whose atoms are `float` or floating-point constants, to be evaluated in `float` rather than `double`.
- `register` declarations: Domain/C completely ignores `register` declarations, except to ensure that language constraints are not violated.
- Include file search rules are different.
- Programs that rely on undefined behaviors, for instance, the order of expression evaluation and the application of unsequenced side-effects, will probably execute differently.

Porting between HP C and VMS C

The C language itself is easy to port from VMS to HP-UX for two main reasons:

- There is a high degree of compatibility between HP C and other common industry implementations of C as well as within the HP-UX family.
- The C language itself does not consider file manipulation or input/output to be part of the core language. These issues are handled via libraries. Thus, C avoids some of the thorniest issues of portability.

In most cases, HP C (in compatibility mode) is a superset of VMS C. Therefore, porting from VMS to HP-UX is easier than porting in the other direction. The next several subsections describe features of C that can cause problems in porting.

Core Language Features

- Basic data types in VMS have the same general sizes as their counterparts on HP-UX. In particular, all integral and floating-point types have the same number of bits. `struct`s and `union`s do not necessarily have the same size because of different alignment rules.
- Basic data types are aligned on arbitrary byte boundaries in VMS C. HP-UX counterparts generally have more restrictive alignments.
- Type `char` is signed by default on both VMS and HP-UX.
- The `unsigned` adjective is recognized by both systems and is usable on `char`, `short`, `int`, and `long`. It can also be used alone to refer to `unsigned int`.
- Both VMS and HP-UX support `void` and `enum` data types although the allowable uses of `enum` vary between the two systems. HP-UX is generally less restrictive.
- The VMS C storage class specifiers `globaldef`, `globalref`, and `globalvalue` have no direct counterparts on HP-UX or other implementations of UNIX. On HP-UX, variables are either local or global, based strictly on scope or `static` class specifiers.
- The VMS C class modifiers `readonly` and `noshare` have no direct counterparts on HP-UX.

- `structs` are packed differently on the two systems. All elements are byte aligned in VMS whereas they are aligned more restrictively on the different HP-UX architectures based upon their type. Organization of fields within the `struct` differs as well.
- Bit-fields within `structs` are more general on HP-UX than on VMS. VMS requires that they be of type `int` or `unsigned` whereas they may be any integral type on HP-UX.
- Assignment of one `struct` to another is supported on both systems. However, VMS permits assignment of `structs` provided the types of both sides have the same size. HP-UX is more restrictive because it requires that the two sides be of the same type.
- VMS C stores floating-point data in memory using a proprietary scheme. Floats are stored in `F_floating` format. Doubles are stored either in `D_floating` format or `G_floating` format. `D_floating` format is the default. HP-UX uses IEEE standard formats which are not compatible with VMS types but which are compatible with most other industry implementations of UNIX.
- VMS C converts floats to doubles by padding the mantissa with 0s. HP-UX uses IEEE formats for floating-point data and therefore must do a conversion by means of floating-point hardware or by use of library functions. When doubles are converted to floats in VMS C, the mantissa is rounded toward zero, then truncated. HP-UX uses either floating point hardware or library calls for these conversions.

  The VMS `D_floating` format can hide programming errors. In particular, you might not immediately notice that mismatches exist between formal and actual function arguments if one is declared `float` and the counterpart is declared `double` because the only difference in the internal representation is the length of the mantissa.

- Due to the different internal representations of floating-point data, the range and precision of floating-point numbers differs on the two systems according to the following tables:

Table 5-2. VMS C Floating-Point Types

Format	Approximate Range of |x|	Approximate Precision
`F_floating`	0.29E-38 to 1.7E38	7 decimal digits
`D_floating`	0.29E-38 to 1.7E38	16 decimal digits
`G_floating`	0.56E-308 to 0.99E308	15 decimal digits

Table 5-3. HP-UX C Floating-Point Types

Format	Approximate Range of |x|	Approximate Precision
`float`	1.17E-38 to 3.40E38	7 decimal digits
`double`	2.2E-308 to 1.8E308	16 decimal digits
`long double`	3.36E-4932 to 1.19E4932	31 decimal digits

- VMS C identifiers are significant to the 31st character. HP-UX C identifiers are significant to 255 characters.
- `register` declarations are handled differently in VMS. The `register` reserved word is regarded by the compiler to be a strong hint to assign a dedicated register for the variable. On Series 300/400, the `register` declaration causes an integral or pointer type to be assigned a dedicated register to the limits of the system, unless optimization at level `+O2` or greater is requested, in which case the compiler ignores `register` declarations. HP 9000 workstations and servers treat `register` declarations as hints to the compiler.
- If a variable is declared to be `register` in VMS and the `&` address operator is used in conjunction with that variable, no error is reported. Instead, the VMS compiler converts the class of that variable to `auto`. HP-UX compilers will report an error.

- Type conversions on both systems follow the usual progression found on implementations of UNIX.
- Character constants (not to be confused with string constants) are different on VMS. Each character constant can contain up to four ASCII characters. If it contains fewer, as is the normal case, it is padded on the left by NULLs. However, only the low order byte is printed when the `%c` descriptor is used with `printf`. Multicharacter character constants are treated as an overflow condition on Series 300/400 if the numerical value exceeds 127 (the overflow is silent). In compatibility mode, HP 9000 workstations and servers detect all multicharacter character constants as error conditions and reports them at compile time.
- String constants can have a maximum length of 65535 characters in VMS. They are essentially unlimited on HP-UX.
- VMS provides an alternative means of identifying a function as being the main program by the use of the adjective `main program` that is placed on the function definition. This extension is not supported on HP-UX. Both systems support the special meaning of `main()`, however.
- VMS implicity initializes pointers to 0. HP-UX makes no implicit initialization of pointers unless they are `static`, so dereferencing an uninitialized pointer is an undefined operation on HP-UX.

- VMS permits combining type specifiers with `typedef` names. So, for example:

  ```
  typedef long t;
  unsigned t x;
  ```

  is permitted on VMS. This is permitted only in compatibility mode on Series 300/400; it is not allowed in ANSI C mode on any HP-UX system. To accomplish this on HP 9000 worstations and servers, change the `typedef` to include the type specifier:

  ```
  typedef unsigned long t;
  t x;
  ```

  Or use a `#define`:

  ```
  #define t long
  unsigned t x;
  ```

Preprocessor Features

- VMS supports an unlimited nesting of `#include`s. HP-UX in compatibility mode guarantees 35 levels of nesting. HP-UX in ANSI mode guarantees 57 levels of nesting.
- The algorithms for searching for `#include`s differs on the two systems. VMS has two variables, `VAXC$INCLUDE` and `C$INCLUDE` which control the order of searching. HP-UX follows the usual order of searching found on most implementations of UNIX.
- `#dictionary` and `#module` are recognized in VMS but not on HP-UX.
- The following symbols are predefined in VMS but not on HP-UX: `vms`, `vax`, `vaxc`, `vax11c`, `vms_version`, `CC$gfloat`, `VMS`, `VAX`, `VAXC`, `VAX11C`, and `VMS_VERSION`.
- The following symbols are predefined on all HP-UX systems but not in VMS:

  `__hp9000s300` on Series 300/400
  `__hp9000s700` on Series 700
  `__hp9000s800` on Series 700/800
  `__hppa` on Series 700/800
  `__hpux` and `__unix` on all systems

- HP-UX preprocessors do not include white space in the replacement text of a macro. The VMS preprocessor does include the trailing white space. If your HP C program depends on the inclusion of the white space, you can place white space around the macro invocation.

Compiler Environment

- In VMS, files with a suffix of `.C` are assumed to be C source files, `.OBJ` suffixes imply object files, and `.EXE` suffixes imply executable files. HP-UX uses the normal conventions on UNIX that `.c` implies a C source file, `.o` implies an object file, and `a.out` is the default executable file (but there is no other convention for executable files).
- `varargs` is supported on VMS and all HP-UX implementations. See *vprintf*(3S) and *varargs*(5) for a description and examples.
- `curses` is supported on VMS and all HP-UX implementations. See *curses*(3X) for a description.
- VMS supports `VAXC$ERRNO` and `errno` as two system variables to return error conditions. HP-UX supports `errno` although there may be differences in the error codes or conditions.
- VMS supplies `getchar` and `putchar` as functions only, not as macros. HP-UX supplies them as macros and also supplies the functions `fgetc` and `fputc` which are the function versions.
- Major differences exist between the file systems of the two operating systems. One of these is that the VMS directory `SYS$LIBRARY` contains many standard definition files for macros. The HP-UX directory `/usr/include` has a rough correspondence but the contents differ greatly.
- A VMS user must explicitly link the RTL libraries `SYS$LIBRARY:VAXCURSE.OLB`, `SYS$LIBRARY:VAXCRTLG.OLB` or `SYS$LIBRARY:VAXCRTL.OLB` to perform C input/output operations. The HP-UX input/output utilities are included in `/lib/libc`, which is linked automatically by `cc` without being specified by the user.
- Certain standard functions may have different interfaces on the two systems. For example, `strcpy()` copies one string to another but the resulting destination may not be NULL terminated on VMS whereas it always will be on HP-UX.

- The commonly used HP-UX names `end`, `edata` and `etext` are not available on VMS.

Calling Other Languages

It is possible to call a routine written in another language from a C program, but you should have a good reason for doing so. Using more than one language in a program that you plan to port to another system will complicate the process. In any case, make sure that the program is thoroughly tested in any new environment.

If you do call another language from C, you will have the other language's anomalies to consider plus possible differences in parameter passing. Since all HP-UX system routines are C programs, calling programs written in other languages should be an uncommon event. If you choose to do so, remember that C passes all parameters by value except arrays and structures. The ramifications of this depend on the language of the called function.

Table 5-4. C Interfacing Compatibility

C	HP-UX Pascal	FORTRAN
`char`	none	`byte`
`unsigned char`	`char`	`character` (could reside on an odd boundary and cause a memory fault)
`char *` (string)	none	none
`unsigned char *` (string)	PAC`+chr(0)` (PAC = `packed array[1..`*n*`] of char`)	Array of `char+char(0)`
`short` (int)	-32768..32767 (`shortint` on Series 700/800)	`integer*2`
`unsigned short` (int)	`BIT16` on Series 700/800; none on Series 300/400 (`0..65535` will generate a 16-bit value only if in a packed structure)	none
`int`	`integer`	`integer` (`*`4)
`long` (int)	`integer`	`integer` (`*`4)
`unsigned` (int)	none	none
`float`	`real`	`real` (`*`4)
`double`	`longreal`	`real*8`
`long double`[1]	none	`real*16`
`type*` (pointer)	`^var`, pass by reference, or use `anyvar`	none
`&var` (address)	`addr(var)` (requires `$SYSPROG$`)	none
`*var` (deref)	`var^`	none
`struct`	`record` (cannot always be done; C and Pascal use different packing algorithms)	`structure`
`union`	`record case of ...`	`union`

1 `long double` is available only in ANSI mode.

Calling FORTRAN

You can compile FORTRAN functions separately by putting the functions you want into a file and compiling it with the `-c` option to produce a `.o` file. Then, include the name of this `.o` file on the `cc` command line that compiles your C program. The C program can refer to the FORTRAN functions by the names they are declared by in the FORTRAN source.

Remember that in FORTRAN, parameters are usually passed by reference (except `CHARACTER` parameters on Series 700/800, which are passed by descriptor), so actual parameters in a call from C must be pointers or variable names preceded by the address-of operator (`&`).

The following program uses a FORTRAN block data subprogram to initialize a common area and a FORTRAN function to access that area:

```
      double precision function get_element(i,j)
      double precision array
      common /a/array(1000,10)
      get_element = array(i,j)
      end

      block data one
      double precision array
      common /a/array(1000,10)
C Note how easily large array initialization is done.
      data array /1000*1.0,1000*2.0,1000*3.0,1000*4.0,1000*5.0,
     *  1000*6.0,1000*7.0,1000*8.0,1000*9.0,1000*10.0/
      end
```

The FORTRAN function and block data subprogram contained in file `xx.f` are compiled using `f77 -c xx.f`.

The C main program is contained in file `x.c`:

```
main()
{
int i;

extern double get_element(int *, int *);

  for (i=1; i <= 10; i++)
      printf("element = %f\n", get_element(&i,&i));
}
```

The C main program is compiled using `cc -Aa x.c xx.o`.

Another area for potential problems is passing arrays to FORTRAN subprograms. An important difference between FORTRAN and C is that FORTRAN stores arrays in column-major order whereas C stores them in row-major order (like Pascal).

For example, the following shows sample C code:

```
int i,j;
int array[10][20];

for (i=0; i<10; i++) {
    for (j=0; j<20; j++)   /* Here the 2nd dimension
                              varies most rapidly */
    array [i][j]=0;
}
```

Here is similar code for FORTRAN:

```
integer array (10,20)

do J=1,20
  do I=1,10       !Here the first dimension varies most rapidly
    array(I,J)=0
  end do
end do
```

Therefore, when passing arrays from FORTRAN to C, a C procedure should vary the first array index the fastest. This is shown in the following example in which a FORTRAN program calls a C procedure:

```
integer array (10,20)

do j=1,20
  do i=1,10
    array(i,j)=0
  end do
end do
call cproc (array)
    .
    .
    .
cproc (array)
int array [][];

for (j=1; j<20; j++) {
    for (i=1; i<20; i++)   /* Note that this is the reverse from
                              how you would normally access the
                              array in C as shown above */
    array [i][j]= ...
}
        .
        .
        .
```

There are other considerations as well when passing arrays to FORTRAN subprograms.

It should be noted that a FORTRAN `main` should *not* be linked with `cc`.

Calling Pascal

Pascal gives you the choice of passing parameters *by value* or *by reference* (`var` parameters). C passes all parameters (other than arrays and structures) by value, but allows passing pointers to simulate pass by reference. If the Pascal function does not use `var` parameters, then you may pass values just as you would to a C function. Actual parameters in the call from the C program corresponding to formal `var` parameters in the definition of the Pascal function should be pointers.

Arrays correlate fairly well between C and Pascal because elements of a multidimensional array are stored in row-major order in both languages. That is, elements are stored by rows; the rightmost subscript varies fastest as elements are accessed in storage order.

Note that C has no special type for boolean or logical expressions. Instead, any integer can be used with a zero value representing false, and non-zero representing true. Also, C performs all integer math in full precision (32-bit); the result is then truncated to the appropriate destination size.

To call Pascal procedures from C on the HP 9000 workstations and servers, a program may first have to call the Pascal procedure `U_INIT_TRAPS`. See the *HP Pascal Programmer's Guide* for details about the `TRY/RECOVER` mechanism.

As true of FORTRAN `main`s, a Pascal `main` should *not* be linked with `cc`.

The following source is the Pascal module:

```
 module a;
export
 function cfunc : integer;
 function dfunc : integer;

implement
 function cfunc : integer;
  var x : integer;

  begin
   x := MAXINT;
   cfunc := x;
  end;

 function dfunc : integer;
  var x : integer;

  begin
   x := MININT;
   dfunc := x;
  end;
  end.
```

The command line for producing the Pascal relocatable object is

```
$ pc -c pfunc.p
```

The command line for compiling the C main program and linking the Pascal module is

```
$ cc x.c pfunc.o -lcl
```

The following output results:

```
2147483647
-2147483648
```

6

Migrating C Programs to HP-UX

This chapter discusses issues to consider when migrating C language programs from VAX systems, HP 9000 Series 300/400, and HP 9000 Series 500 computers to HP 9000 workstations and servers. The first section lists some steps you need to take to migrate an application program to an HP 9000 workstation or server. Subsequent sections in this chapter highlight major differences between various C compilers and suggest how to modify source files to ease migration.

Because C is a highly portable language, if you follow the recommendations given in the chapter "Programming for Portability," your program should migrate easily. However, if you use system-dependent programming practices, a program that executes successfully on one computer may not execute properly when transferred to a HP 9000 workstation or server. For example, if you use system-specific I/O routines outside of the standard C library, you will have difficulty with portability.

Migrating an Application

Following are the general steps to migrate a C program from an HP-UX or UNIX system.

1. Test your program on the current system so you have a copy of the results.
2. Use the `tar` command (see the *HP-UX Reference* manual) with the `cv` options to transfer the source files you want to migrate to tape.
3. Use the `tar` command with the `r` option to transfer any associated data files to tape.

4. Install the source files and any related data files on the HP 9000 workstation or server using the **tar** command with the `x` option.
5. Check your makefiles for any implementation-specific options. Change programs depending on implementation-specific command options. On HP-UX systems, these options are generally preceded by `-W` or `+`, and may include options to be passed to `ld` or `cpp`. You can optionally include the `-g` option to permit symbolic debugging.
6. Review the lists of "Guidelines for Portability" and "Practices to Avoid" in the previous chapter and check over the source code for system-dependent programming. (If the source files are extensive, you may want to skip this step and catch errors when you run `lint` or compile.)
7. Search for instances of `#include` files and make sure that the files or routines included appear in the correct directory or library on the HP 9000 workstation or server.
8. Run `lint`, a C program checker that verifies source code and prints warning messages about problems with the source code style, efficiency, portability, and consistency.
9. Compile the program on the HP 9000 workstation or server using the `cc` command. (Refer to the *HP C/HP-UX Reference Manual* for details about the `cc` command and options, and explanations of error, warning, and panic messages.) Change the source code to resolve any messages you receive.
10. Recompile the program until you receive no messages.
11. Link the program. The linker reports any symbols that cannot be found.
12. Run the program on the HP 9000 workstation or server. Compare the results with those received on the original computer.

Byte Order

The VAX computer has a different byte order from HP 9000 computers. Binary data files created on a VAX computer may need to be swapped before they can be interpreted on an HP 9000 workstation or server. Use the descriptions of storage and alignment on both systems to write a programming tool to reorder the data. The C library function `swab` (see the *HP-UX Reference Manual*) can be used to swap bytes, if that is sufficient for the particular application. Otherwise, you need to write a customized tool. ASCII code and data files should migrate to the HP 9000 workstation or server without change.

Data Alignment

The HP 9000 workstations and servers are more strict than other machines with respect to data alignment. Misaligned data addresses cause bus errors when attempting to dereference them. Use the `+w1` option when compiling to report occurrences of "Casting from loose to strict alignment." Fix occurrences that result from using the address of a more loosely aligned item (such as `char`) to access a more strictly aligned item (such as `int`).

Unsupported Keywords

Some implementations of C permit use of the keywords `asm`, `fortran`, and `entry`. These are not supported on the HP 9000 workstations and servers. You must rewrite any code that uses these keywords.

Predefined Macro Names

In non-ANSI mode, there are several HP C specific macro names defined. These names may conflict with identifiers used in the source code.

The HP 9000 workstation and server preprocessors predefine the macro names `PWB`, `hpux`, and `unix`. The HP 9000 workstations and servers predefine the macro name `hp9000s800`; the HP 9000 Series 500 predefines `hp9000s500`; and the HP 9000 Series 300/400 predefine the macro name `hp9000s300`. The VAX predefines the macro name `vax`. If any of these macro names is used as an identifier in the source code, use the `#undef` preprocessor directive to "undefine" the macro or rename the identifier(s).

In ANSI mode, none of the above macro names are defined and you should not have difficulty with these HP C specific macro names.

White Space

HP 9000 Series 300/400, 500, and workstation and server preprocessors do not include trailing white space in the replacement text of a macro. The VAX preprocessor includes the trailing white space. If your program depends on the inclusion of the white space, you can place white space around the macro invocation.

Hexadecimal Escape Sequence

The HP 9000 workstations and servers compiler allows character constants containing hexadecimal escape sequences. For example, `'A'` can be expressed with the hexadecimal escape sequence `'\x41'`. The HP 9000 Series 200, 300, and 500 do not allow hexadecimal escape sequences.

Check your source files for any occurrences for `\x`, and verify that a hexadecimal escape sequence is intended.

Invalid Structure References

The HP 9000 workstations and server compiler does not allow structure members to be referenced through a pointer to a different type of object. The VAX pcc and HP 9000 Series 200 and 500 compilers allow this. Change any invalid structure references to cast the pointer to the appropriate type before referencing the member. For example, given the following:

```
struct x {
  int y;
}z;
char *c;
c -> y=5;
```

`c -> y=5;` is invalid. Instead, use the following code:

```
c = (char *) &z;
((struct x *) c)->y = 5;
```

Leading Underscore

External names on the HP 9000 workstations and servers do not contain a leading underscore. You need to change any programs that rely on external names containing leading underscores. Note that all languages on the HP 9000 workstations and servers follow the same convention. Therefore, only assembly language code and names that were aliased in other languages are affected by this. Because there is no leading underscore, external names contain one additional significant character. Identifiers that differ only in the 255th character will denote different items on the HP 9000 workstations and servers.

Library Functions

The set of library routines available on HP-UX systems may differ from those available on BSD 4.2 systems. If you encounter an unresolved function after linking, refer to the *HP-UX Reference Manual* to see if there is an HP-UX function that does what you want it to do. If not, you will have to write one of your own.

Floating-Point Format

The VAX floating-point representation is different from that on HP 9000 computers. You will have to change any programs dependent on the characteristics of VAX floating point. In particular, this difference could expose errors in the code that happen to work acceptably on the VAX. These errors include mismatched function return types (`float` on one side, `double` on the other), and passing the address of a `double` instead of a `float` to `scanf`. The VAX representation of a `float` differs in the number of bits in the exponent, as well as the mantissa. Therefore, mismatched types can cause a vastly different answer on HP 9000 computers.

Bit-Fields

The HP 9000 workstations and servers C compiler treats bit-fields without the `unsigned` type modifier as signed. The VAX, HP 9000 Series 300/400, and 500 compilers treat them as unsigned.

Data Storage and Alignment

The alignment requirements of some data types are different on the HP 9000 workstations and servers. Check any externally imposed data structure layouts for differences. These may include byte and bit-field order, if you are migrating from a VAX, or different internal padding for structure member alignment. On the HP 9000 workstations and servers, `doubles` must be aligned on a 64-bit boundary, whereas other machines require alignment on a 32-bit boundary. Refer to Chapter 2 for complete storage and alignment information.

Typedefs

The HP C compiler does not allow combining type specifiers with `typedef` names.

For example:

```
typedef long t;
unsigned t var;
```

Compilers derived from pcc accept this code, but HP C does not. Change the `typedef` to include the type specifier:

```
typedef unsigned long t;
t var;
```

or use a define:

```
#define t long
unsigned t var;
```

7

Using C Programming Tools

This chapter contains a list and a description of the C tools. It also provides "how to use" information on `lint` and discusses HP specific features of `lex` and `yacc`. For more information on each of the HP C tools see the `man` pages or *HP-UX Reference Vol. 1: Section 1*. Another general source of information on `lex` and `yacc` is *lex and yacc* by John R. Levine, Tony Mason, and Doug Brown.

Description of C Programming Tools

Below is a brief description of each of the C tools.

- `cb` is a C program beautifier.
- `cflow` is a C flow graph generator.
- `cpp` is the C language preprocessor.
- `ctags` is a C programming tool that creates a tag file for *ex(1)* or *vi(1)* from the specified C, Pascal, and FORTRAN sources.
- `cxref` is a C program cross-reference generator.
- `lex` is a program generator for lexical analysis of text.
- `lint` is a C program checker.
- `yacc` is a programming tool for describing the input to a computer program.

HP Specific Features of lex and yacc

The following native language support features have been added to the HP C `lex` and `yacc` tools:

- `LC_CTYPE` and `LC_MESSAGES` environment variable support in `lex` - Determines the size of the characters and language in which messages are displayed while you use `lex`.
- `-m` command line option for `lex` - Specifies that multibyte characters may be used anywhere single byte characters are allowed. You can intermix both 8-bit and 16-bit multibyte characters in regular expressions if you enable the `-m` command line option.
- `-w` command line option for `lex` - Includes all features in `-m` and returns data in the form of the `wchar_t` data type.
- `%l <locale>` directive for `lex` - Specifies the locale at the beginning of the definitions section. Any valid locale recognized by the `setlocale` function can be used. This directive is similar to using the `LC_CTYPE` environment variable. To receive `wchar_t` support with `%l`, use the `-w` command line option.
- `LC_CTYPE` environment variable support in `yacc` - Determines the native language set used by `yacc` and enables multibyte character sets. Multibyte characters can appear in token names, on terminal symbols, strings, comments, or anywhere ASCII characters can appear, except as separators or special characters.
- If you see the diagnostic message `yacc stack overflow`, then add the macro

  ```
  #define __RUNTIME_YYMAXDEPTH
  ```

  at the beginning of the user subroutine section in the `.y` file.

Using lint

The main purpose of `lint` is to supply the programmer with warning messages about problems with the source code's style, efficiency, portability, and consistency. The `lint` command can be used before compiling a program to check for syntax errors and after compiling a program to test for subtle errors such as type differences.

Error messages and `lint` warnings are sent to standard error (`stderr`). Once the code errors are corrected, the C source file(s) should be run through the C compiler to produce the necessary object code.

The `lint` command has the form:

`lint [`*options*`]`*files ... library-descriptors ...*

where *options* are options flags to control `lint` checking and messages, *files* are the files to be checked that end with `.c` or `.ln`, and *library descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the `lint` command are:

`-a`	Suppresses messages about assignments of long values to variables that are not long.
`-b`	Suppresses messages about break statements that cannot be reached.
`-c`	Only checks for intrafile bugs; leaves external information in files suffixed with `.ln`.
`-h`	Does not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).
`-n`	Does not check for compatibility with either the standard or the portable `lint` library.
`-o` *name*	Creates a `lint` library from input files named `llib-l`*name*`.ln`.
`-p`	Attempts to check portability to other dialects of C language.

`-s`	Checks for cases where the alignment of structures, unions, and pointers may not be portable.
`-u`	Suppresses messages about function and external variables used and not defined or defined and not used.
`-v`	Suppresses messages about unused arguments and functions.
`-x`	Does not report variables referred to by external declarations but never used.
`-Aa`	Invokes `lint` in ANSI mode.
`-Ac`	Invokes `lint` in compatibility mode. The default is compatibility mode.

The names of files that contain C language programs should end with the suffix `.c`, which is mandatory for `lint` and the C compiler.

The `lint` command accepts certain arguments, such as:

```
-lm
```

The `lint` library files are processed almost exactly like ordinary source files. The only difference is that functions that are defined on a library file but are not used on a source file do not result in messages. The `lint` command does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, `lint` checks the programs it is given against a standard library file which contains descriptions of the programs which are normally loaded when a C language program is run. When the `-p` option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The `-n` option can be used to suppress all library checking.

`lint` also recognizes the `-LINTLIBRARY` the HP C `-Wp` option. The `lint -LINTLIBRARY` option is equivalent to using `lint` comment `/*LINTLIBRARY*/` in source files. `lint` also recognizes the `-Wp` option and passes named arguments to the preprocessor.

Directives

The alternative to using options to suppress `lint`'s comments about problem areas is to use directives. Directives appear in the source code in the form of code comments. The `lint` command recognizes five directives.

`/*NOTREACHED*/`	Stops an unreachable code comment about the next line of code.
`/*NOSTRICT*/`	Stops `lint` from strictly type checking the next expression.
`/*ARGSUSED*/`	Stops a comment about any unused parameters for the following function.
`/*VARARGS`*n*`*/`	Stops `lint` from reporting variable numbers of parameters in calls to a function. The function's definition follows this comment. The first *n* parameters must be present in each call to the function; `lint` comments if they aren't. If `/*VARARGS*/` appears without the *n*, none of the parameters must be present. This comment must precede the actual code for a function. It *should not* precede `extern` declarations.
`/*LINTLIBRARY*/`	Tells `lint` that the source file is used to create a `lint` library file and to suppress comments about the unused functions. `lint` objects if other files redefine routines that are found there. This directive must be placed at the beginning of a source file.

Problem Detection

Remember that a compiler reports errors only when it encounters program source code that cannot be converted into object code. The main purpose of `lint` is to find problem areas in C source code that it considers to be inefficient, nonportable, bad style, or a possible bug, but which the C compiler accepts as error-free because it can be converted into object code.

Comments about problems that are local to a function are produced as each problem is detected. They have the following form:

(*line #*) `warning`: *message text*

Information about external functions and variables is collected and analyzed after `lint` has processed the source files. At that time, if a problem has been detected, it outputs a warning message with the form

message text

followed by a list of external names causing the message and the file where the problem occurred.

Code causing `lint` to issue a warning message should be analyzed to determine the source of the problem. Sometimes the programmer has a valid reason for writing the problem code. Usually, though, this is not the case. The `lint` command can be very helpful in uncovering subtle programming errors.

The `lint` command checks the source code for certain conditions, about which it issues warning messages. These can be grouped into the following categories:

- variable or function is declared but not used
- variable is used before it is set
- portion of code is unreachable
- function values are used incorrectly
- type matching does not adhere strictly to C rules
- code has portability problems
- code construction is strange

The code that you write may have constructions in it that `lint` objects to but that are necessary to its application. Warning messages about problem areas that you know about and do not plan to correct can be suppressed. There are two methods for suppressing warning messages from `lint`. The use of `lint` options is one. The `lint` command can be called with any combination of its defined option set. Each option causes `lint` to ignore a different problem area. The other method is to insert `lint` directives into the source code. For information about `lint` directives, see the section "Directives" in this chapter.

Unused Variables and Functions

The `lint` command objects if source code declares a variable that is never used or defines a function that is never called. Unused variables and functions are considered bad style because their declarations clutter the code.

Unused static identifiers cause the following message:

`(1)static identifier` *'name'* `defined but never used`

Unused automatic variables cause the following message:

`(1) warning:` *'name'* `unused in function` *'name'*

A function or external variable that is unused causes the message

`name defined but never used`

followed by the function or variable name, the line number and file in which it was defined. The `lint` command also looks at the special case where one of the parameters of a function is not used. The warning message is:

`warning:` *(line number) 'arg_name'* `in` *func_name'*

If functions or external variables are declared but never used or defined, `lint` responds with

`name declared but never used or defined`

followed by a list of variable and functions names and the names of files where they were declared.

Suppressing Unused Functions and Variables Reports

Sometimes it is necessary to have unused function parameters to support consistent interfaces between functions. The `-v` option can be used with `lint` to suppress warnings about unused parameters.

If `lint` is run on a file that is linked with other files at compile time, many external variables and functions can be defined but not used, as well as used but not defined. If there is no guarantee that the definition of an external object is always seen before the object code is used, it is declared `extern`. The `-u` option can be used to stop complaints about all external objects, whether or not they are declared `extern`. If you want to inhibit complaints about only the `extern` declared functions and variables, use the `-x` option.

Set/Used Information

A problem exists in a program if a variable's value is used before it is assigned. Although `lint` attempts to detect occurrences of this, it takes into account only the physical location of the code. If code using a local variable is located before the variable is given a value, the message is:

`warning:` *'name'* `may be used before set`

The `lint` command also objects if automatic variables are set in a function but not used. The message given is:

`warning:` *'name'* `set but not used in function` *'func_name'*

Note that `lint` *does not* have an option for suppressing the display of warnings for variables that are used but not set or set but not used.

Unreachable Code

The `lint` command checks for three types of unreachable code. Any statement following a `goto`, `break`, `continue`, or `return` statement must either be labeled or reside in an outer block for `lint` to consider it reachable. If neither is the case, `lint` responds with:

`warning:` *(line number)* `statement not reached`

The same message is given if `lint` finds an infinite loop. It only checks for the infinite loop cases of `while(1)` and `for(;;)`. The third item that `lint` looks for is a loop that cannot be entered from the top. If one is found, then the message sent is:

```
warning: loop not entered from top
```

The `lint` command's detection of unreachable code is by no means exhaustive. Warning messages can be issued about valid code, and conversely `lint` may overlook code that cannot be reached.

Programs that are generated by `yacc` or `lex` can have many unreachable `break` statements. Normally, each one causes a complaint from `lint`. The `-b` option can be used to force `lint` to ignore unreachable `break` statements.

Function Value

The C compiler allows a function containing both the statement

```
return();
```

and the statement

```
return(expression);
```

to pass through without complaint. The `lint` command, however, detects this inconsistency and responds with the message:

```
warning: function 'name' has 'return(expression)' and 'return'
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f(a)
{
    if (a) return (3);
    g();
}
```

Notice that is `a` tests false, `f` will call `g` and then return with no defined value. This will trigger a message for `lint`. If `g` (like `exit`) never returns, the message will still be produced when in fact nothing is wrong. In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, `lint` detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition. When the value is sometimes used, it may represent bad style (e.g., not testing for error conditions).

The `lint` command will not issue a diagnostic message if that function call is cast as `void`. For example,

```
(void) printf("%d\n",i);
```

tells `lint` to not warn about the ignored return value.

The dual problem—using a function value when the function does not return one—is also detected. This is a serious problem.

The `lint` command *does not* have an option for suppressing the display of warning for inconsistent `return` functions and functions that return no value.

Portability

The `-p` option of `lint` aids the programmer is writing portable code in four areas:

- character comparisons
- pointer alignments (this is default on PA-RISC computers)
- length of external variables
- type casting

Character representation varies on different machines. Characters may be implemented as signed values. As a result, certain comparisons with characters give different results on different machines. The expression

```
c<0
```

where `c` is defined as type `char`, is always false if characters are unsigned values. If, however, characters are signed values, the expression could be either true or false. Where character comparisons could result in different values depending on the machine used, `lint` outputs the message:

```
warning: nonportable character comparison
```

Legal pointer assignments are determined by the alignment restrictions of the particular machine used. For example, one machine may allow double-precision values to begin on any modulo-4 boundary, but another may restrict them to modulo-8 boundaries. If alignment requirements are different, code containing an assignment of a double pointer to an integer pointer could cause problems. The `lint` command attempts to detect where the effect of pointer assignments is machine dependent. The warning that it outputs is:

```
warning: possible pointer alignment problem
```

The amount of information about external symbols that is loaded depends on: the machine being used, the number of significant characters, and whether or not uppercase/lowercase distinction is kept. The `lint -p` command truncates all external symbols to six characters and allows only one case distinction. (It changes uppercase characters to lowercase.) This provides a worst-case analysis so that the uniqueness of an external symbol is not machine-dependent.

The effectiveness of type casting in C programs can depend on the machine that is used. For this reason, `lint` ignores type casting code. All assignments that use it are subject to `lint`'s type checking.

Alignment Portability

The `-s` option of the `lint` command checks for the following portability considerations:

- pointer alignments (same as `-p` option)
- a structure's member alignments
- trailing padding of structures and unions

The checks made for pointer alignments are exactly the same as for the `-p` option. The warning for these cases is:

```
warning: possible pointer alignment problem
```

The alignment of structure members is different between architectures. For example, MC680x0 computers pad structures internally so that all fields of type `int` begin on an even boundary. In contrast, PA-RISC computers pad structures so that all fields of type `int` begin on a four-byte boundary. The following structure will be aligned differently on the two architectures:

```
struct s
   { char c;
     long l; /* The offset equals 2 on MC680x0 computers */
   };        /* and 4 on PA-RISC computers.              */
```

In many cases the different alignment of structures does not affect the behavior of a program. However, problems can happen when raw structures are written to a file on one architecture and read back in on another. The `lint` command checks for cases where a structure member is aligned on a boundary that is not a multiple of its size (for example, `int` on `int` boundary, `short` on `short` boundary, and `double` on `double` boundary). The warning that it outputs is:

```
warning: alignment of struct 'name' may not be portable
```

The `lint` command also checks for cases where the internal padding added at the end of a structure may differ between architectures. The amount of trailing padding can change the size of a structure. The warning that `lint` outputs is:

```
warning: trailing padding of struct/union 's' may not be portable
```

Strange Constructions

A *strange construction* is code that `lint` considers to be bad style or a possible bug.

The `lint` command looks for code that has no effect. For example,

```
*p++;
```

where the * has no effect. The statement is equivalent to "p++;". In cases like this, the message

```
warning: null effect
```

is sent.

The treatment of unsigned numbers as signed numbers in comparison causes `lint` to report the following:

```
warning: degenerate unsigned comparison
```

The following code would produce such a message:

```
unsigned x;
    .
    .
    .
if (x >=0) ...
```

The `lint` command also objects if constants are treated as variables. If the boolean expression in a conditional has a set value due to constants, such as

```
if(1 !=0) ...
```

`lint`'s response is:

```
warning: constant in conditional context
```

To avoid operator precedence confusion, `lint` encourages using parentheses in expressions by sending the message:

```
warning: precedence confusion possible: parenthesize!
```

The `lint` command judges it bad style to redefine an outer block variable in an inner block. Variables with different meanings should normally have different names. If variables are redefined, the message sent is:

```
warning: name redefinition hides earlier one
```

The `-h` option suppresses `lint` diagnostics of strange constructions.

Standards Compliance

The `lint` libraries are arranged for standards checking. For example,

```
lint -D_POSIX_SOURCE file.c
```

checks for routines referenced in `file.c` but not specified in the POSIX standard.

The `lint` command also accepts ANSI standard C `-Aa` as well as compatible C `-Ac`. In ANSI mode, `lint` invokes the ANSI preprocessor (`/lib/cpp.ansi`) instead of the compatibility preprocessor (`/lib/cpp`). ANSI mode `lint` should be used on source that is compiled with the ANSI standard C compiler.

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U