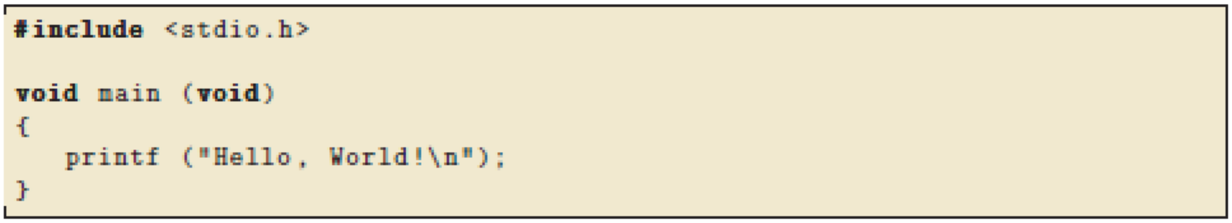
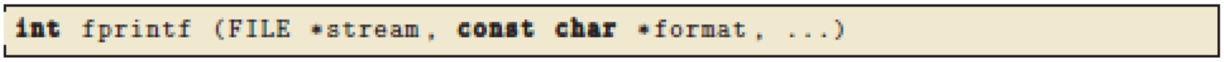
EE4144 – C Reference Code:

Formatted Output/Strings



```
#include <stdio.h>

void main (void)
{
   printf ("Hello, World!\n");
}
```



```
int fprintf (FILE *stream, const char *format, ...)
```

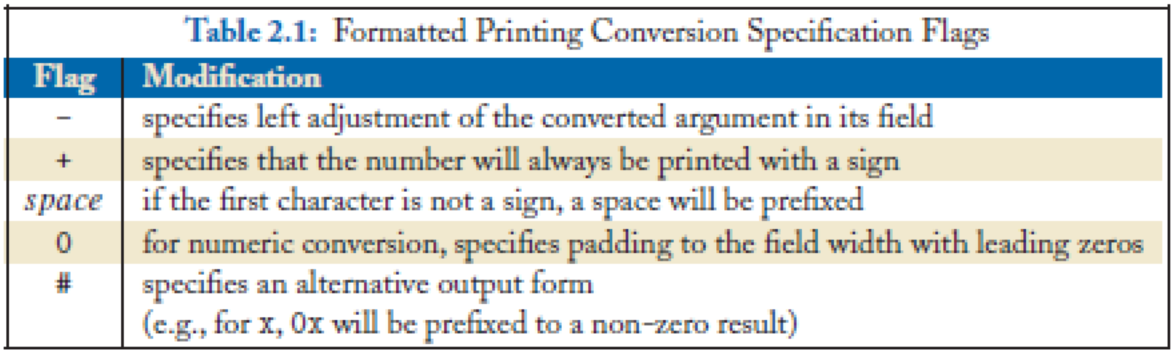


Table 2.1: Formatted Printing Conversion Specification Flags

Flag	Modification
-	specifies left adjustment of the converted argument in its field
+	specifies that the number will always be printed with a sign
space	if the first character is not a sign, a space will be prefixed
0	for numeric conversion, specifies padding to the field width with leading zeros
#	specifies an alternative output form (e.g., for `x`, `0x` will be prefixed to a non-zero result)

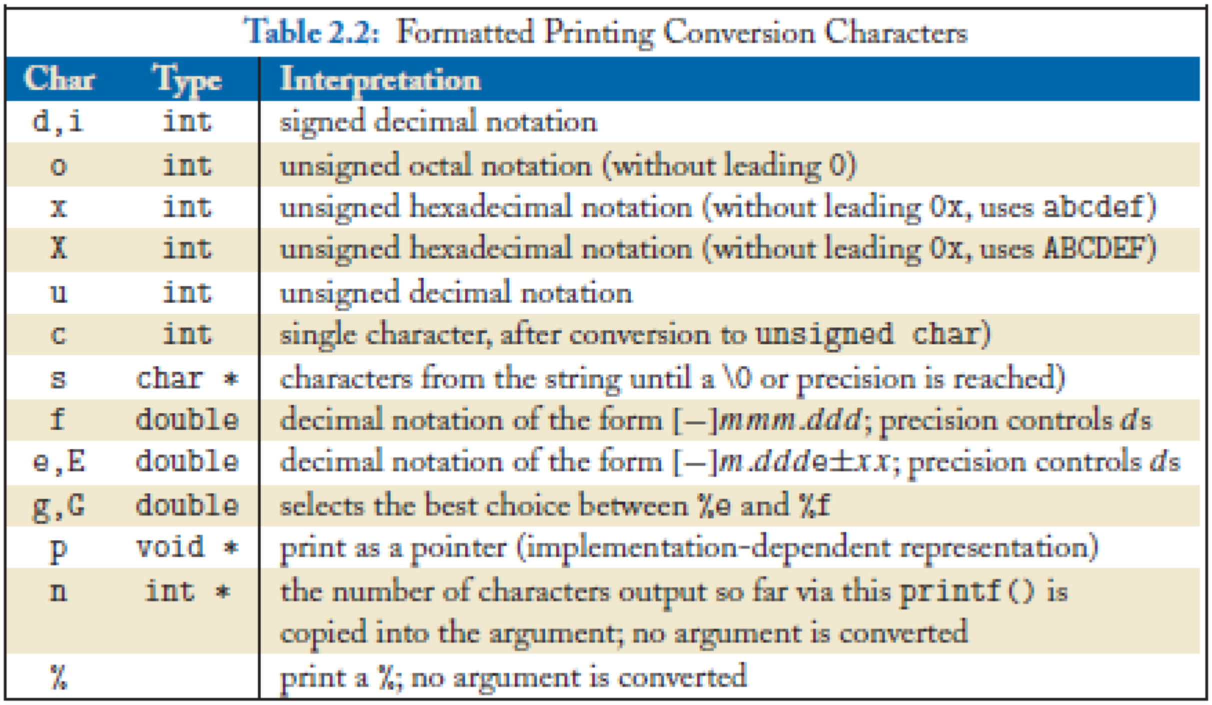


Table 2.2: Formatted Printing Conversion Characters

Char	Type	Interpretation
`d,i`	`int`	signed decimal notation
`o`	`int`	unsigned octal notation (without leading 0)
`x`	`int`	unsigned hexadecimal notation (without leading `0x`, uses `abcdef`)
`X`	`int`	unsigned hexadecimal notation (without leading `0x`, uses `ABCDEF`)
`u`	`int`	unsigned decimal notation
`c`	`int`	single character, after conversion to `unsigned char`)
`s`	`char *`	characters from the string until a \0 or precision is reached)
`f`	`double`	decimal notation of the form [−]*mmm.ddd*; precision controls *d*s
`e,E`	`double`	decimal notation of the form [−]*m.ddd*e±*xx*; precision controls *d*s
`g,G`	`double`	selects the best choice between `%e` and `%f`
`p`	`void *`	print as a pointer (implementation-dependent representation)
`n`	`int *`	the number of characters output so far via this `printf()` is copied into the argument; no argument is converted
`%`		print a `%`; no argument is converted

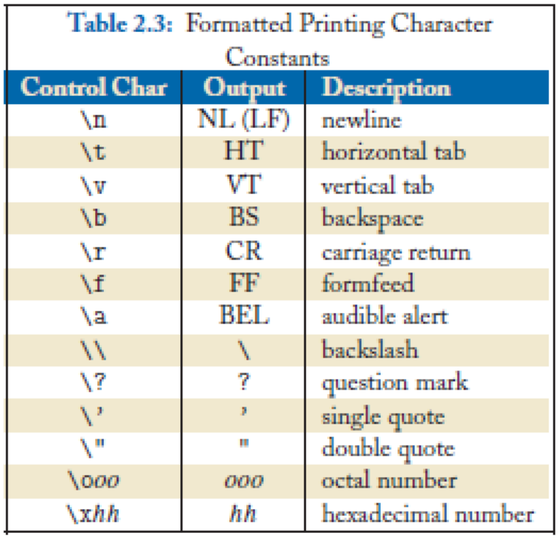
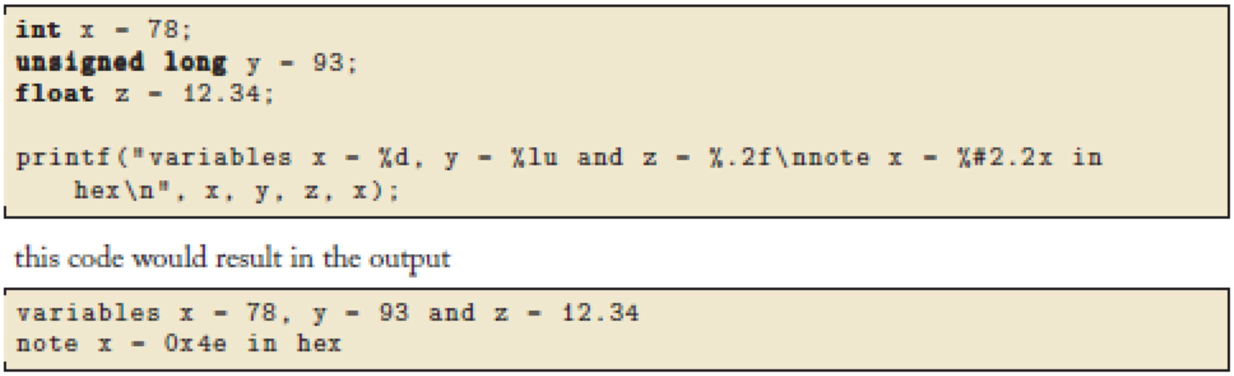


Table 2.3: Formatted Printing Character Constants

Control Char	Output	Description
\n	NL (LF)	newline
\t	HT	horizontal tab
\v	VT	vertical tab
\b	BS	backspace
\r	CR	carriage return
\f	FF	formfeed
\a	BEL	audible alert
\\	\	backslash
\?	?	question mark
\'	'	single quote
\"	"	double quote
\ooo	ooo	octal number
\xhh	hh	hexadecimal number

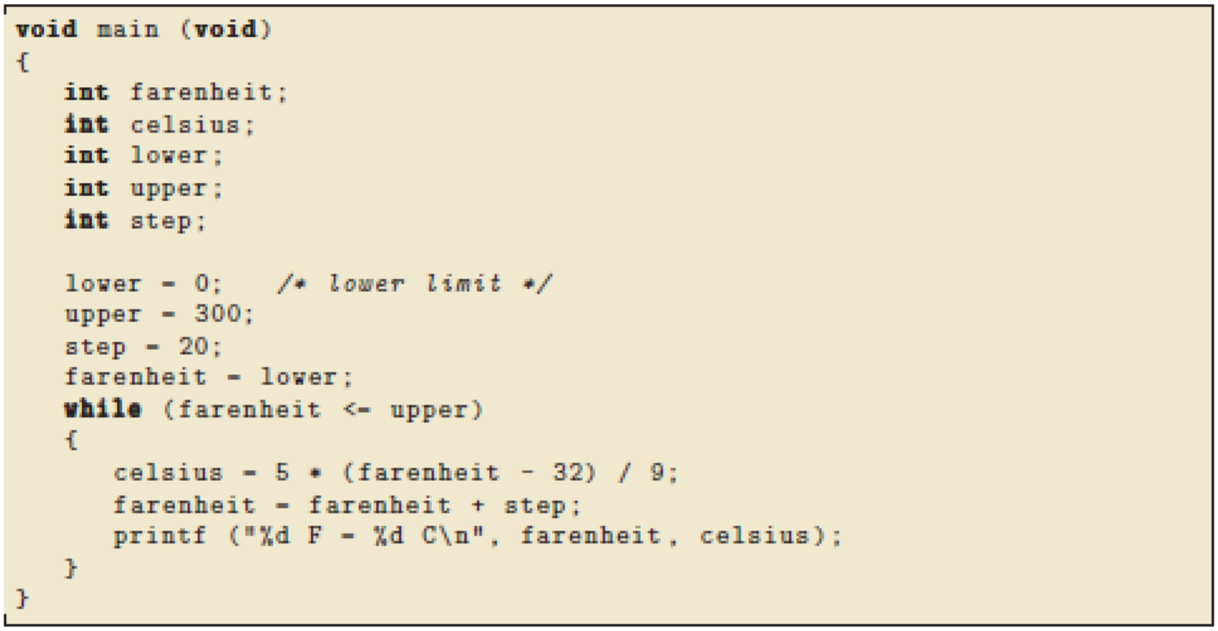


```
int x = 78;
unsigned long y = 93;
float z = 12.34;

printf("variables x = %d, y = %lu and z = %.2f\nnote x = %#2.2x in
    hex\n", x, y, z, x);
```

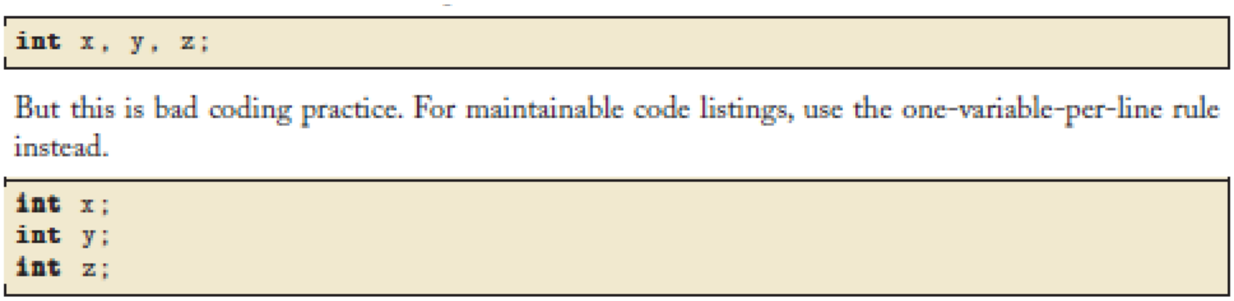
this code would result in the output

```
variables x = 78, y = 93 and z = 12.34
note x = 0x4e in hex
```

Variables:



```
void main (void)
{
   int farenheit;
   int celsius;
   int lower;
   int upper;
   int step;

   lower = 0;    /* lower limit */
   upper = 300;
   step = 20;
   farenheit = lower;
   while (farenheit <= upper)
   {
      celsius = 5 * (farenheit - 32) / 9;
      farenheit = farenheit + step;
      printf ("%d F = %d C\n", farenheit, celsius);
   }
}
```



```
int x, y, z;
```

But this is bad coding practice. For maintainable code listings, use the one-variable-per-line rule instead.

```
int x;
int y;
int z;
```

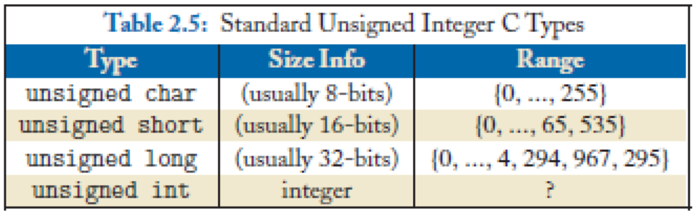


Table 2.5: Standard Unsigned Integer C Types

Type	Size Info	Range
unsigned char	(usually 8-bits)	{0, ..., 255}
unsigned short	(usually 16-bits)	{0, ..., 65, 535}
unsigned long	(usually 32-bits)	{0, ..., 4, 294, 967, 295}
unsigned int	integer	?

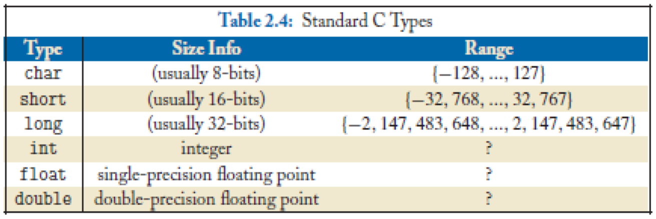
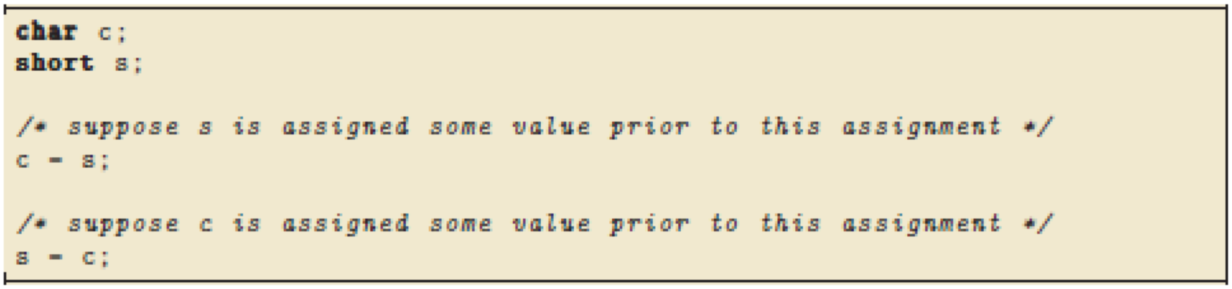


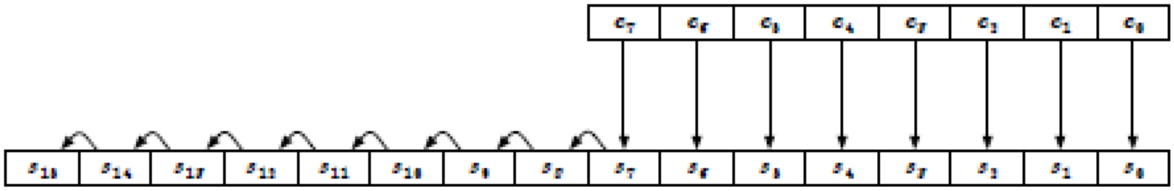
Table 2.4: Standard C Types

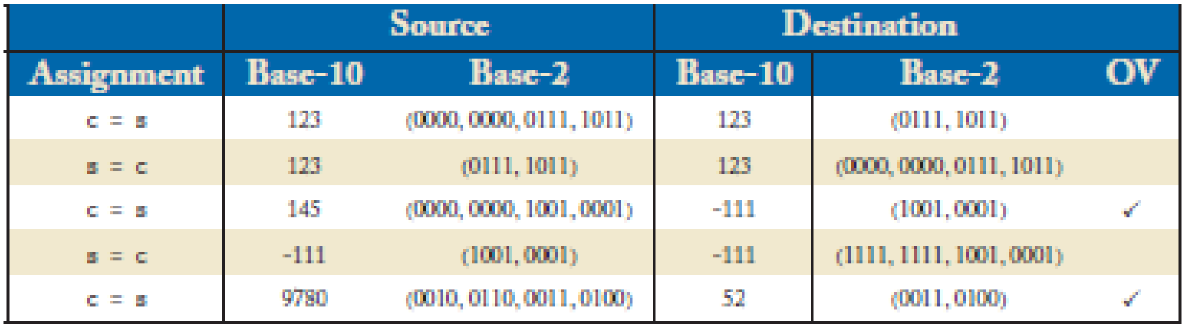
Type	Size Info	Range
char	(usually 8-bits)	{−128, ..., 127}
short	(usually 16-bits)	{−32, 768, ..., 32, 767}
long	(usually 32-bits)	{−2, 147, 483, 648, ..., 2, 147, 483, 647}
int	integer	?
float	single-precision floating point	?
double	double-precision floating point	?

Conversions:

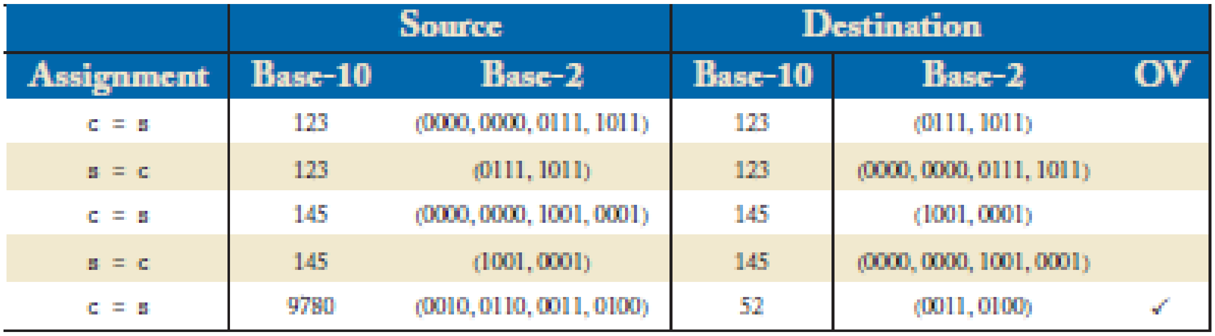


```
char c;
short s;

/* suppose s is assigned some value prior to this assignment */
c = s;

/* suppose c is assigned some value prior to this assignment */
s = c;
```

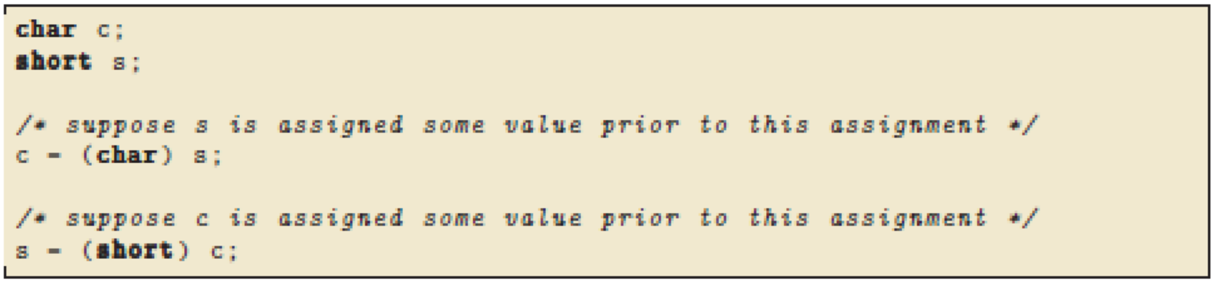




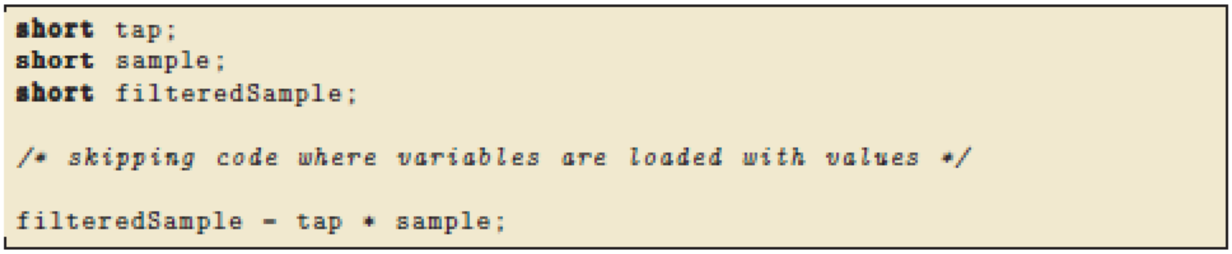
	Source		Destination		
Assignment	Base-10	Base-2	Base-10	Base-2	OV
c = s	123	(0000, 0000, 0111, 1011)	123	(0111, 1011)	
s = c	123	(0111, 1011)	123	(0000, 0000, 0111, 1011)	
c = s	145	(0000, 0000, 1001, 0001)	-111	(1001, 0001)	✓
s = c	-111	(1001, 0001)	-111	(1111, 1111, 1001, 0001)	
c = s	9780	(0010, 0110, 0011, 0100)	52	(0011, 0100)	✓



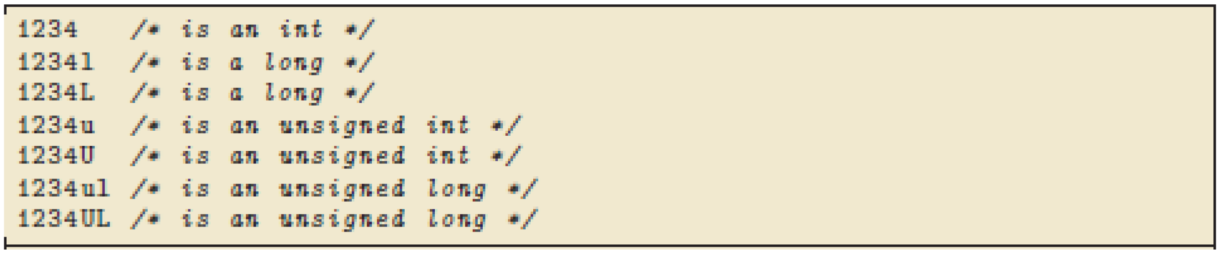
	Source		Destination		
Assignment	Base-10	Base-2	Base-10	Base-2	OV
c = s	123	(0000, 0000, 0111, 1011)	123	(0111, 1011)	
s = c	123	(0111, 1011)	123	(0000, 0000, 0111, 1011)	
c = s	145	(0000, 0000, 1001, 0001)	145	(1001, 0001)	
s = c	145	(1001, 0001)	145	(0000, 0000, 1001, 0001)	
c = s	9780	(0010, 0110, 0011, 0100)	52	(0011, 0100)	✓



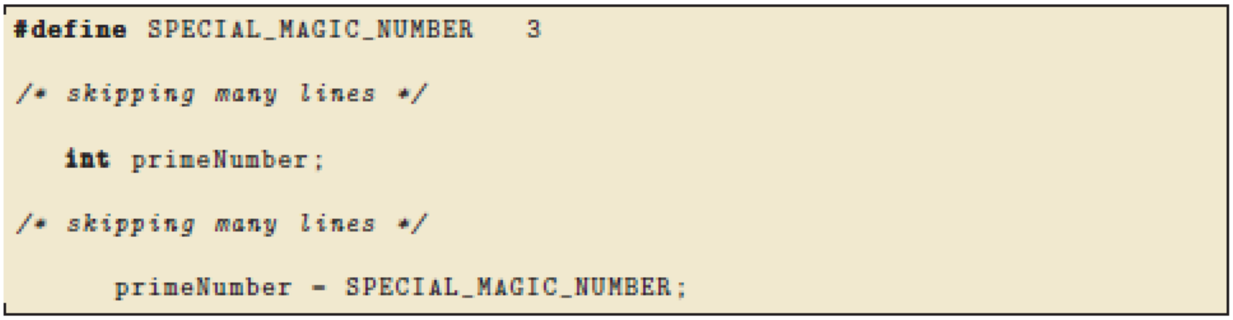
```
char c;
short s;

/* suppose s is assigned some value prior to this assignment */
c = (char) s;

/* suppose c is assigned some value prior to this assignment */
s = (short) c;
```



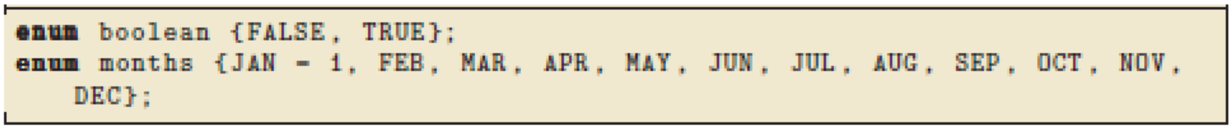
```
short tap;
short sample;
short filteredSample;

/* skipping code where variables are loaded with values */

filteredSample = tap * sample;
```



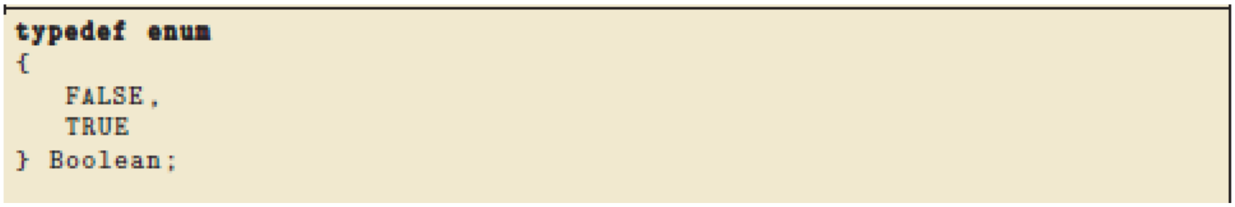
```
1234   /* is an int */
1234l  /* is a long */
1234L  /* is a long */
1234u  /* is an unsigned int */
1234U  /* is an unsigned int */
1234ul /* is an unsigned long */
1234UL /* is an unsigned long */
```



```
#define SPECIAL_MAGIC_NUMBER   3

/* skipping many lines */

   int primeNumber;

/* skipping many lines */

      primeNumber = SPECIAL_MAGIC_NUMBER;
```



```
enum boolean {FALSE, TRUE};
enum months {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV,
    DEC};
```



```
typedef enum
{
   FALSE,
   TRUE
} Boolean;
```

Arithmetic:

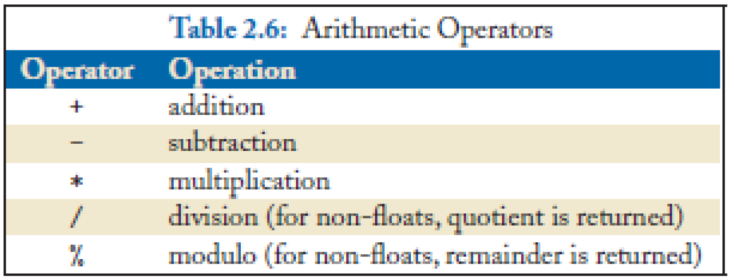
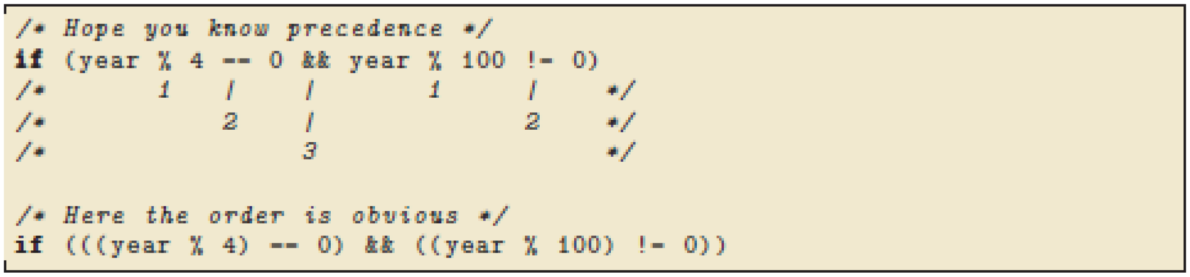


Table 2.6: Arithmetic Operators

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division (for non-floats, quotient is returned)
%	modulo (for non-floats, remainder is returned)



```
/* Hope you know precedence */
if (year % 4 == 0 && year % 100 != 0)
/*       1   |    |       1     |     */
/*           2    |             2     */
/*                3                   */

/* Here the order is obvious */
if (((year % 4) == 0) && ((year % 100) != 0))
```

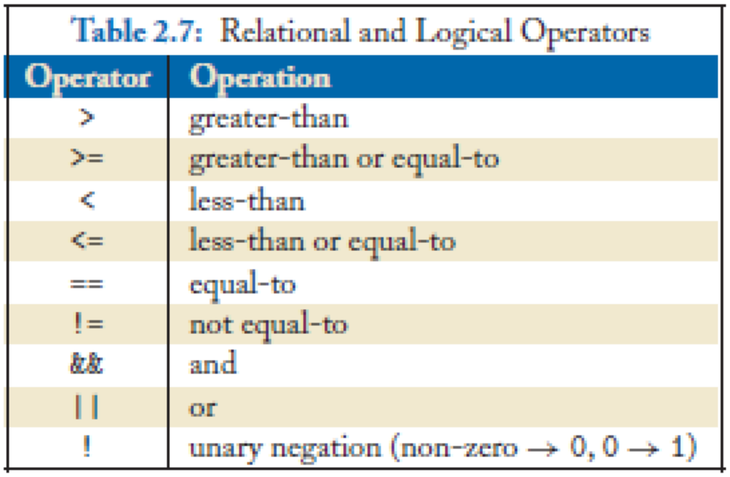
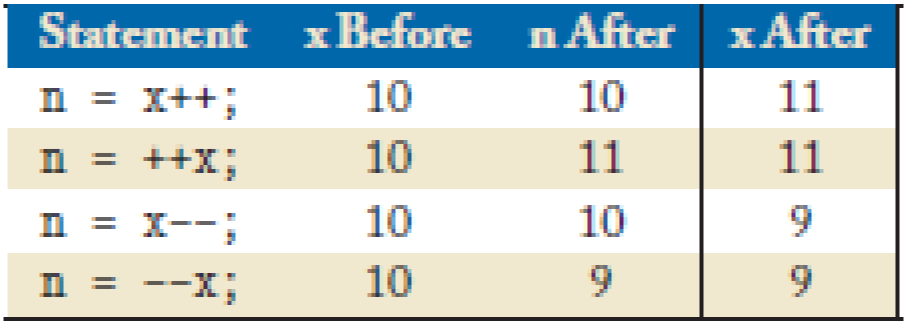


Table 2.7: Relational and Logical Operators

Operator	Operation
>	greater-than
>=	greater-than or equal-to
<	less-than
<=	less-than or equal-to
==	equal-to
!=	not equal-to
&&	and
||	or
!	unary negation (non-zero → 0, 0 → 1)



Statement	x Before	n After	x After
n = x++;	10	10	11
n = ++x;	10	11	11
n = x--;	10	10	9
n = --x;	10	9	9

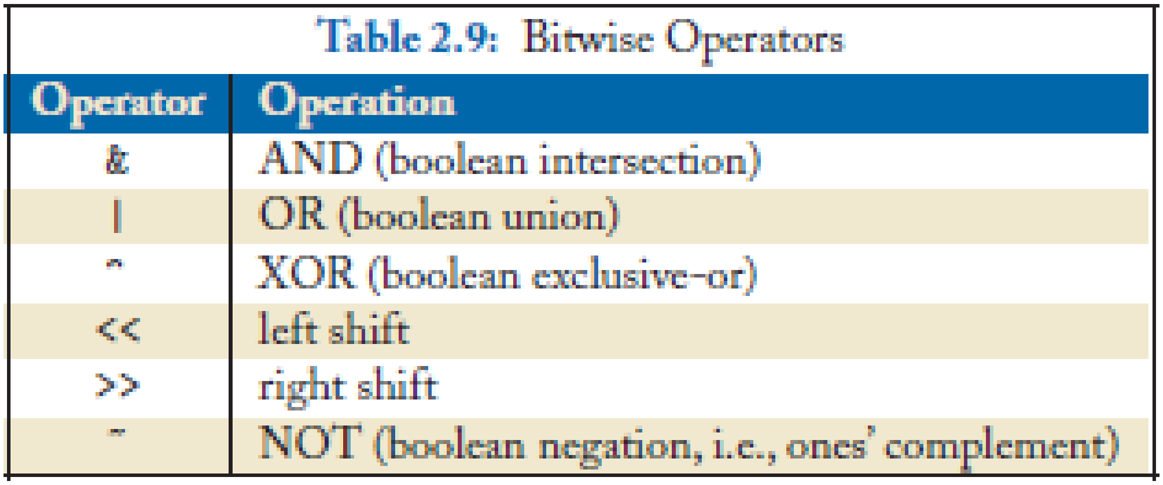
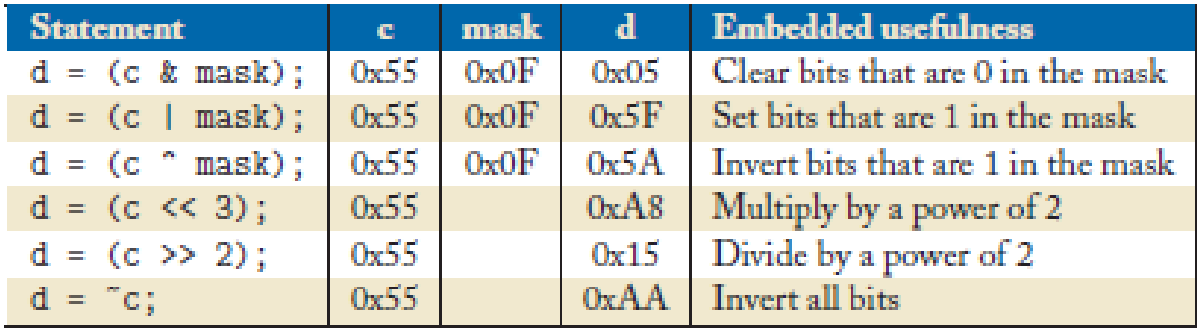
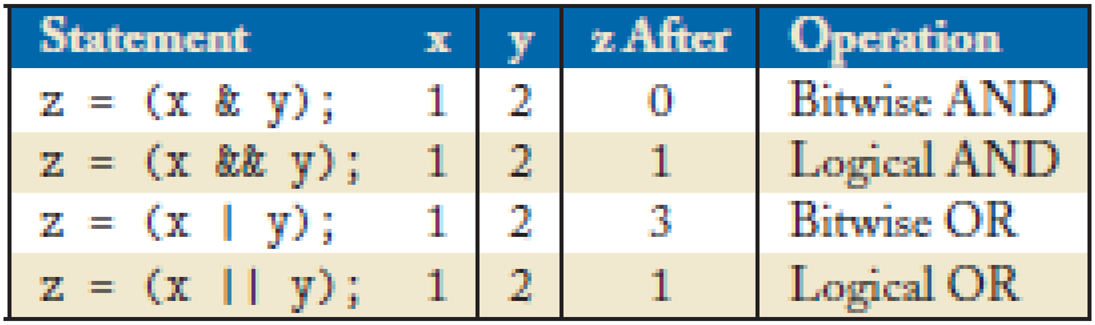


Table 2.9: Bitwise Operators

Operator	Operation
&	AND (boolean intersection)
|	OR (boolean union)
^	XOR (boolean exclusive-or)
<<	left shift
>>	right shift
~	NOT (boolean negation, i.e., ones' complement)



Statement	c	mask	d	Embedded usefulness
`d = (c & mask);`	0x55	0x0F	0x05	Clear bits that are 0 in the mask
`d = (c | mask);`	0x55	0x0F	0x5F	Set bits that are 1 in the mask
`d = (c ^ mask);`	0x55	0x0F	0x5A	Invert bits that are 1 in the mask
`d = (c << 3);`	0x55		0xA8	Multiply by a power of 2
`d = (c >> 2);`	0x55		0x15	Divide by a power of 2
`d = ~c;`	0x55		0xAA	Invert all bits



Statement	x	y	z After	Operation
`z = (x & y);`	1	2	0	Bitwise AND
`z = (x && y);`	1	2	1	Logical AND
`z = (x | y);`	1	2	3	Bitwise OR
`z = (x || y);`	1	2	1	Logical OR

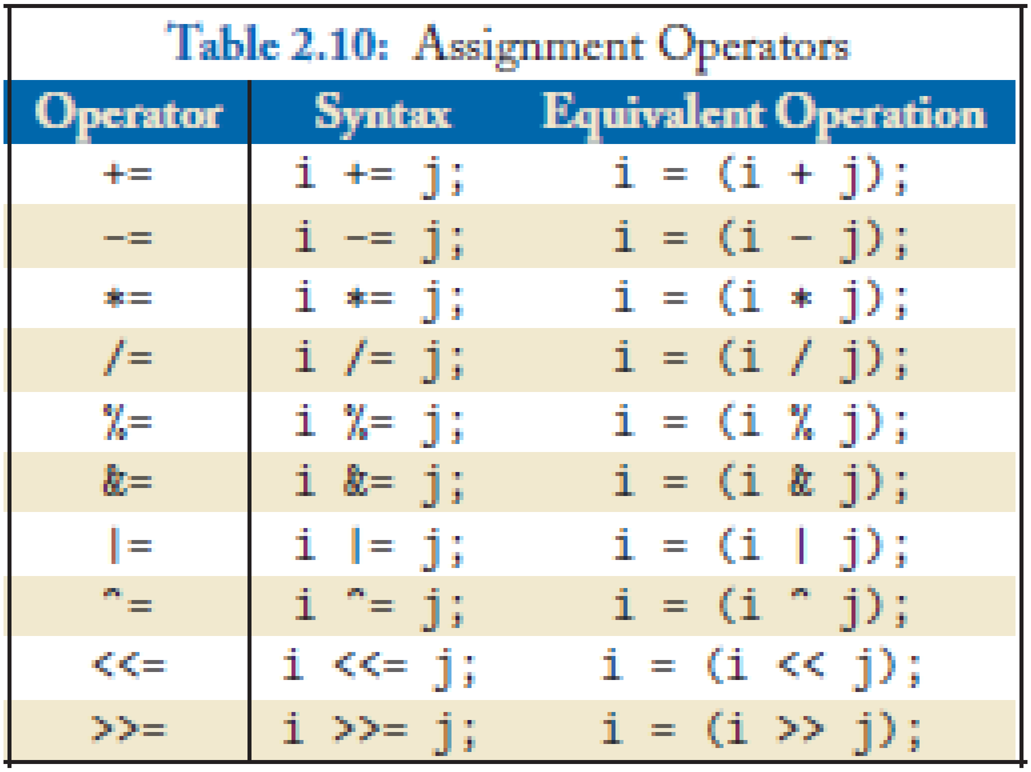
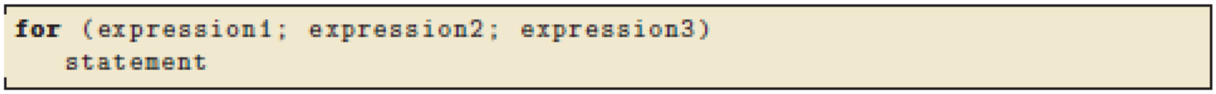


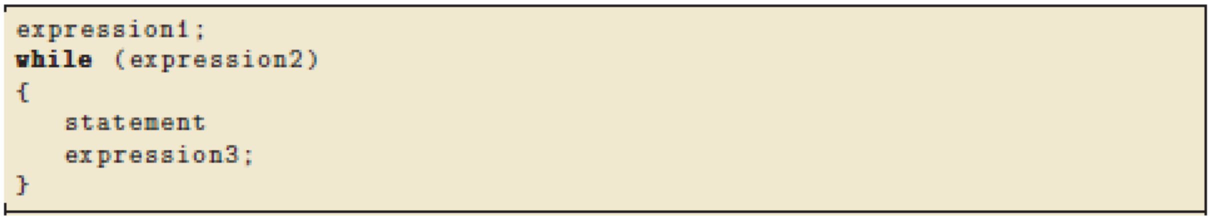
Table 2.10: Assignment Operators

Operator	Syntax	Equivalent Operation
`+=`	`i += j;`	`i = (i + j);`
`-=`	`i -= j;`	`i = (i - j);`
`*=`	`i *= j;`	`i = (i * j);`
`/=`	`i /= j;`	`i = (i / j);`
`%=`	`i %= j;`	`i = (i % j);`
`&=`	`i &= j;`	`i = (i & j);`
`|=`	`i |= j;`	`i = (i | j);`
`^=`	`i ^= j;`	`i = (i ^ j);`
`<<=`	`i <<= j;`	`i = (i << j);`
`>>=`	`i >>= j;`	`i = (i >> j);`

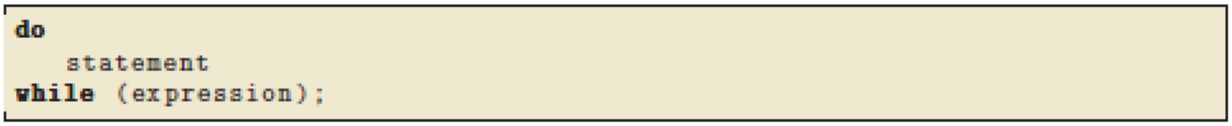
Loops



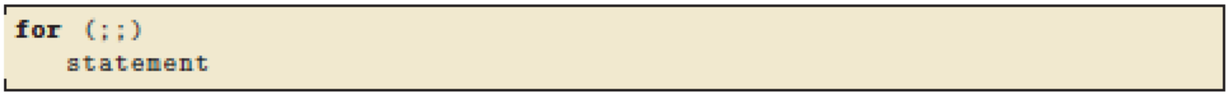
```
for (expression1; expression2; expression3)
   statement
```



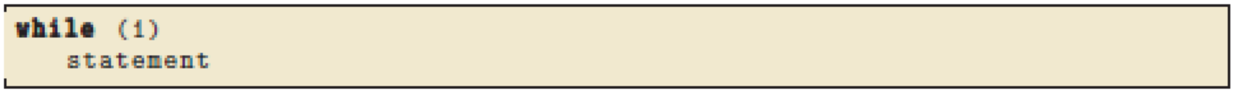
```
expression1;
while (expression2)
{
   statement
   expression3;
}
```



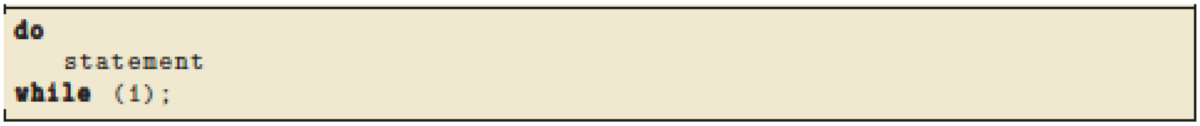
```
do
   statement
while (expression);
```



```c
for (;;)
   statement
```

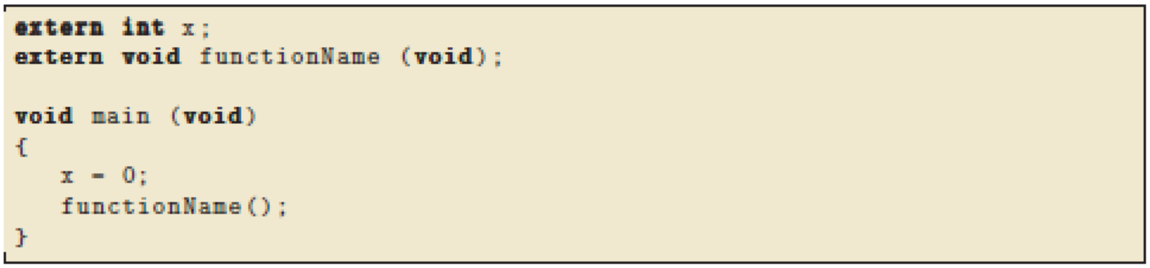


```c
while (1)
   statement
```

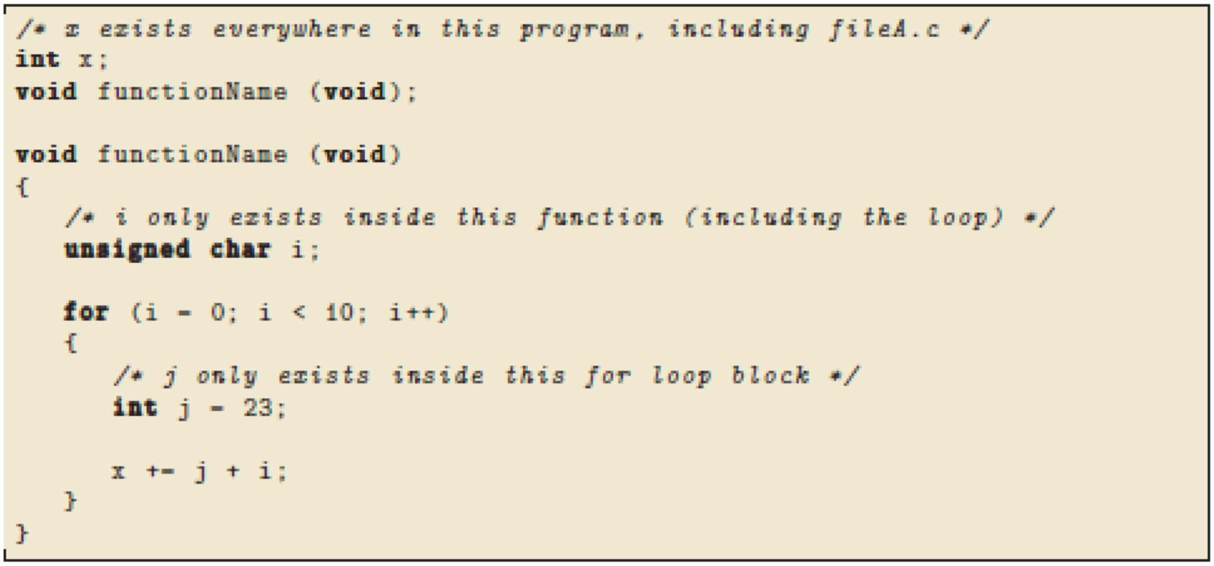


```c
do
   statement
while (1);
```

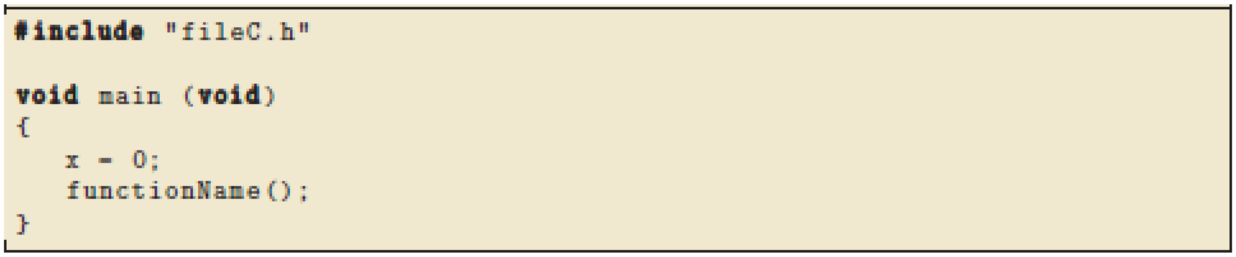
Scope:



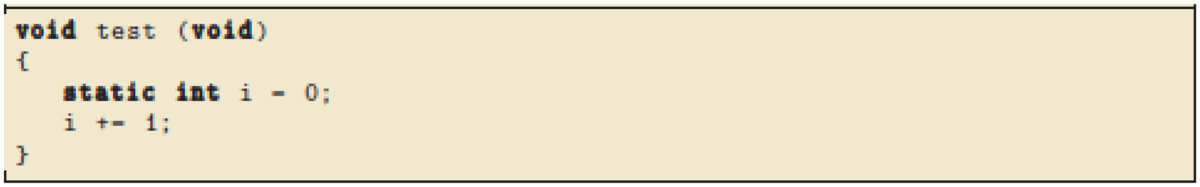
```c
extern int x;
extern void functionName (void);

void main (void)
{
   x = 0;
   functionName();
}
```



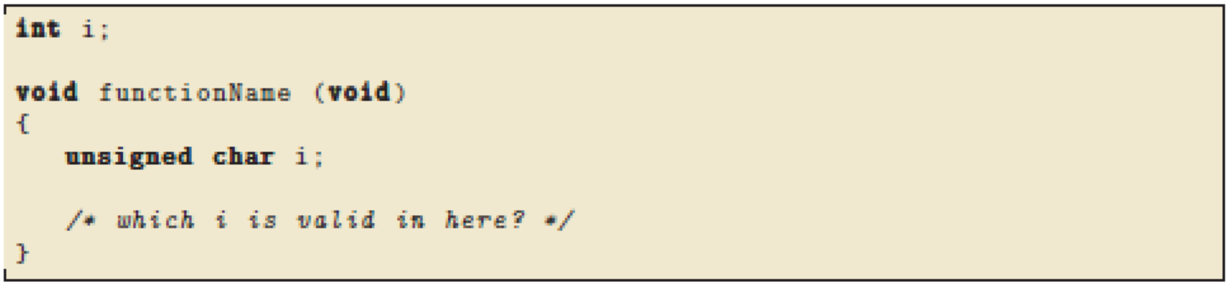
```c
/* x exists everywhere in this program, including fileA.c */
int x;
void functionName (void);

void functionName (void)
{
   /* i only exists inside this function (including the loop) */
   unsigned char i;

   for (i = 0; i < 10; i++)
   {
      /* j only exists inside this for loop block */
      int j = 23;

      x += j + i;
   }
}
```



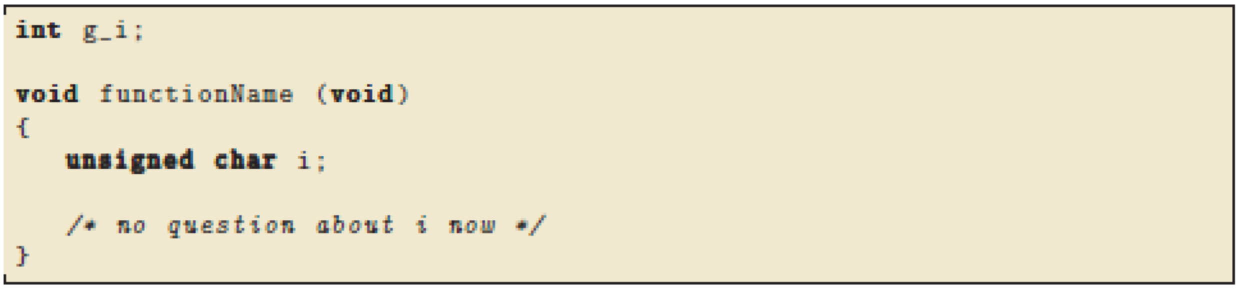
```
#include "fileC.h"

void main (void)
{
   x = 0;
   functionName();
}
```



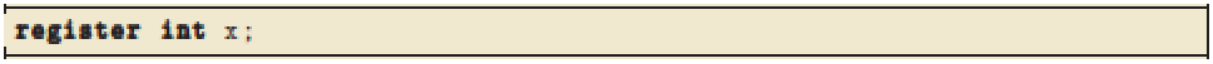
```
void test (void)
{
   static int i = 0;
   i += 1;
}
```



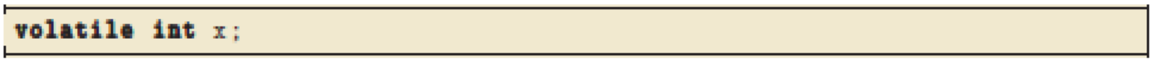
```
int i;

void functionName (void)
{
   unsigned char i;

   /* which i is valid in here? */
}
```



```
int g_i;

void functionName (void)
{
   unsigned char i;

   /* no question about i now */
}
```



```
register int x;
```



```
volatile int x;
```

Pointers:

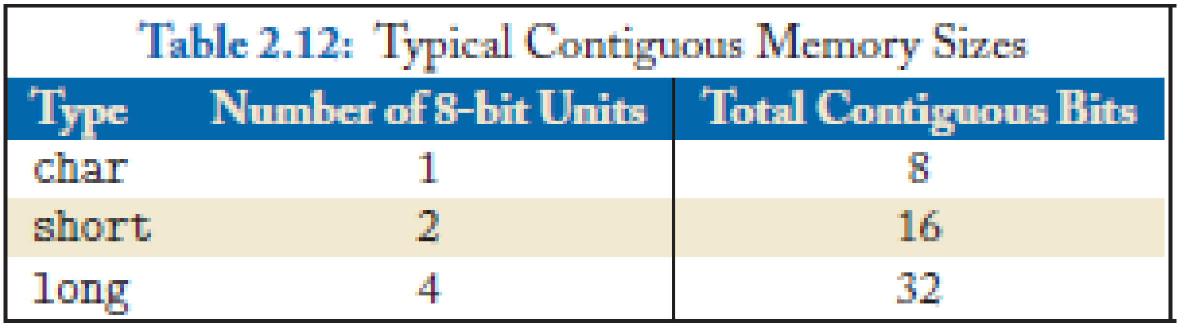
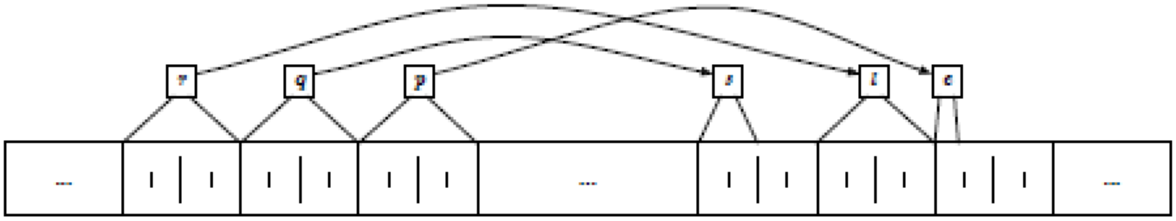
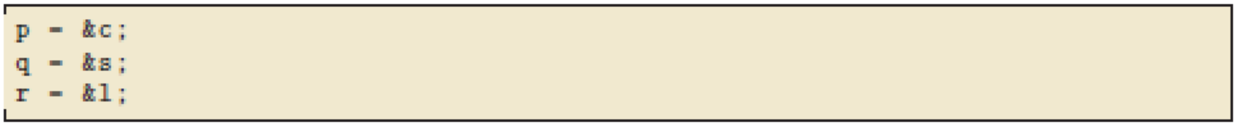


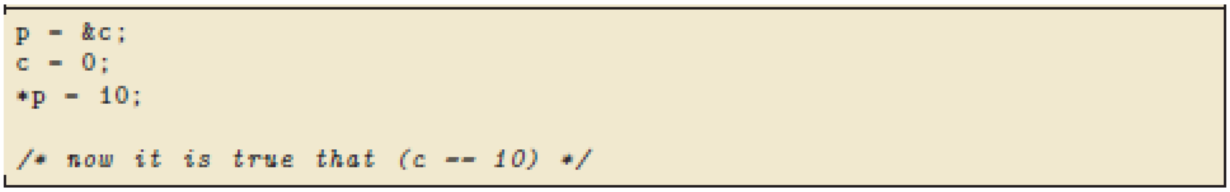
Table 2.12: Typical Contiguous Memory Sizes

Type	Number of 8-bit Units	Total Contiguous Bits
char	1	8
short	2	16
long	4	32

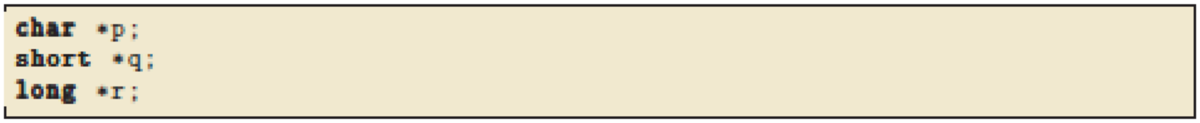




```
p = &c;
q = &s;
r = &l;
```



```
p = &c;
c = 0;
*p = 10;

/* now it is true that (c == 10) */
```



```
char *p;
short *q;
long *r;
```

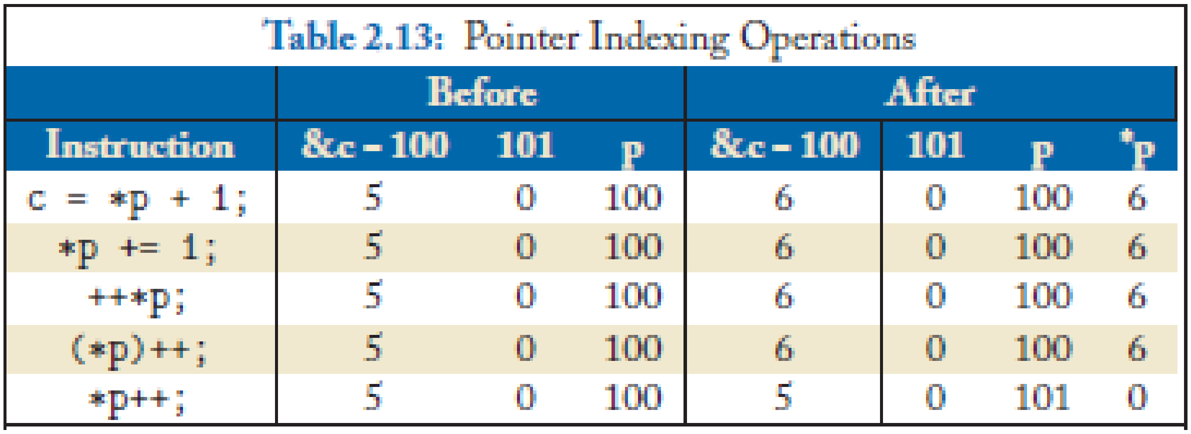
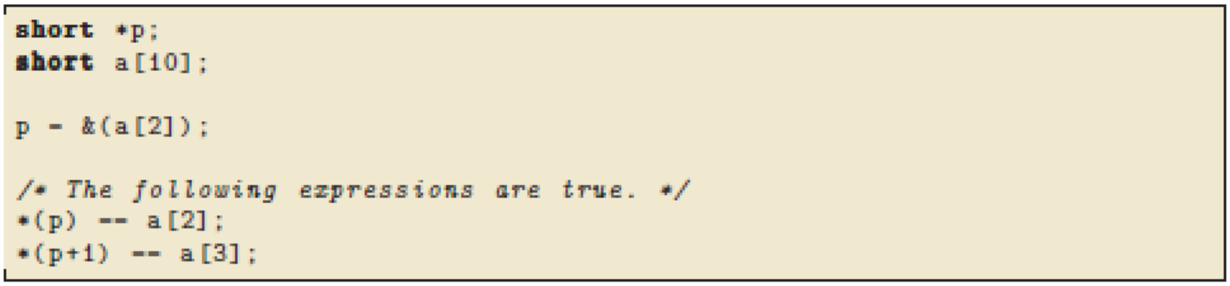


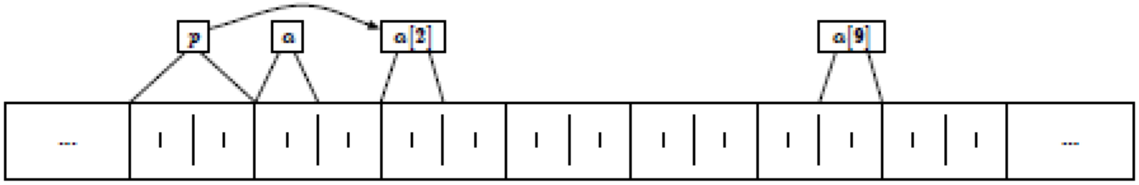
Table 2.13: Pointer Indexing Operations

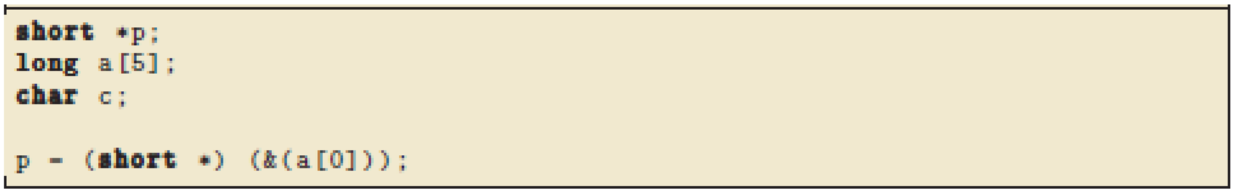
	Before			After			
Instruction	&c – 100	101	p	&c – 100	101	p	*p
c = *p + 1;	5	0	100	6	0	100	6
*p += 1;	5	0	100	6	0	100	6
++*p;	5	0	100	6	0	100	6
(*p)++;	5	0	100	6	0	100	6
*p++;	5	0	100	5	0	101	0

Pointer Issues:

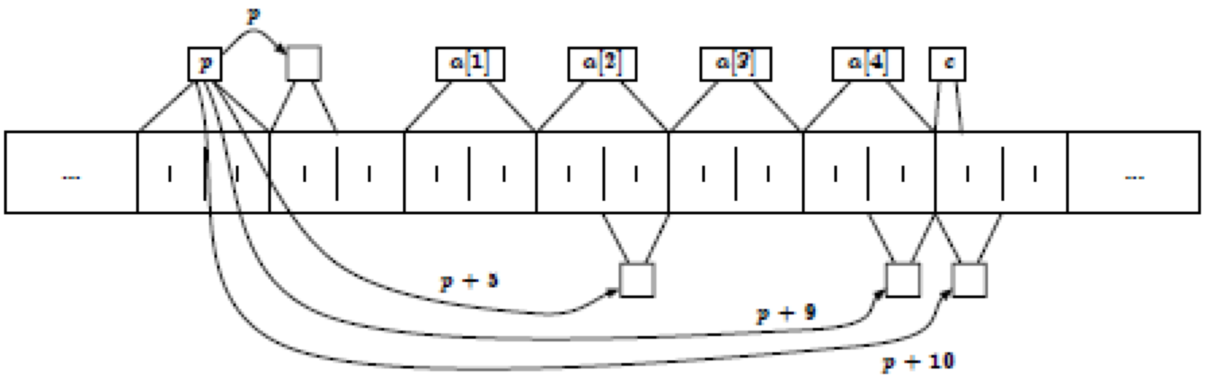


```
short *p;
short a[10];

p = &(a[2]);

/* The following expressions are true. */
*(p) == a[2];
*(p+1) == a[3];
```



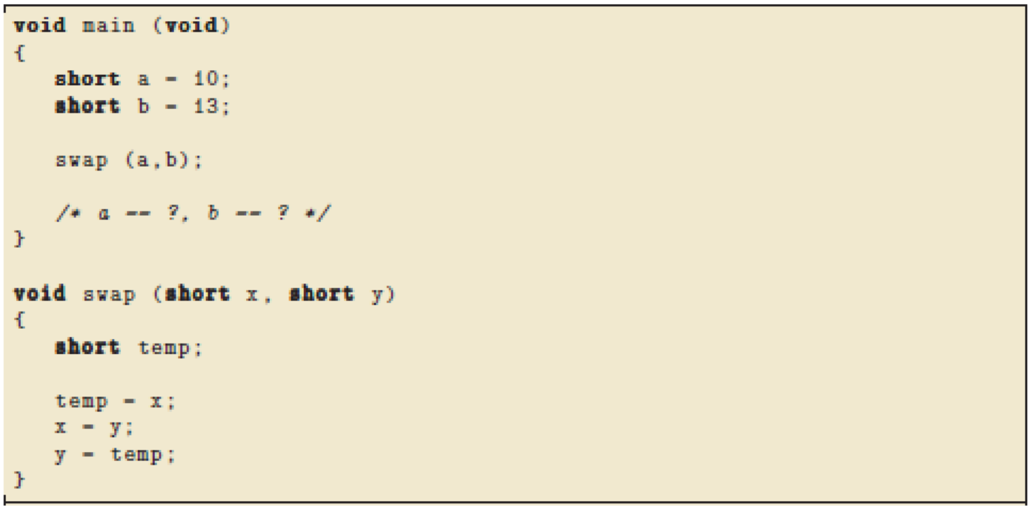


```
short *p;
long a[5];
char c;

p = (short *) (&(a[0]));
```

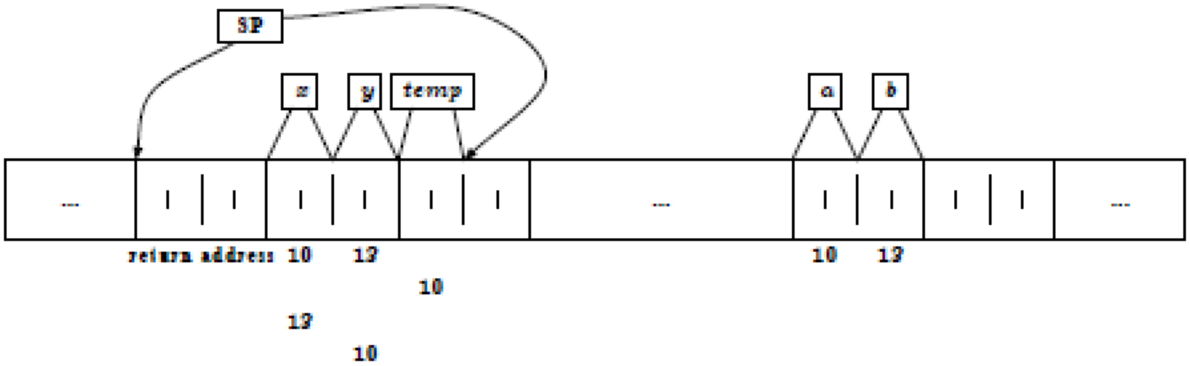


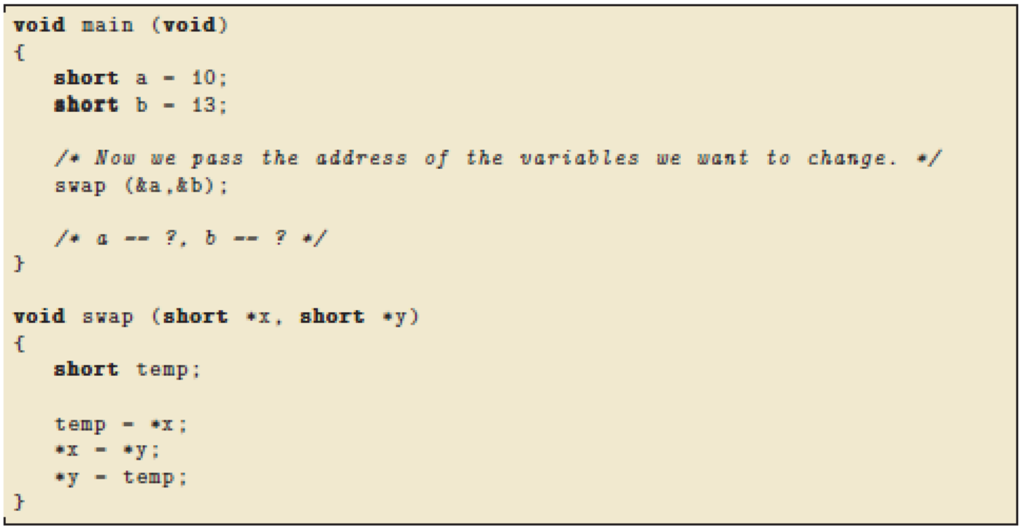
```
/* these are equivalent ways of accessing the fifth element away from p
    */
*(p+5) == p[5];
```

Passing by Value/Reference:



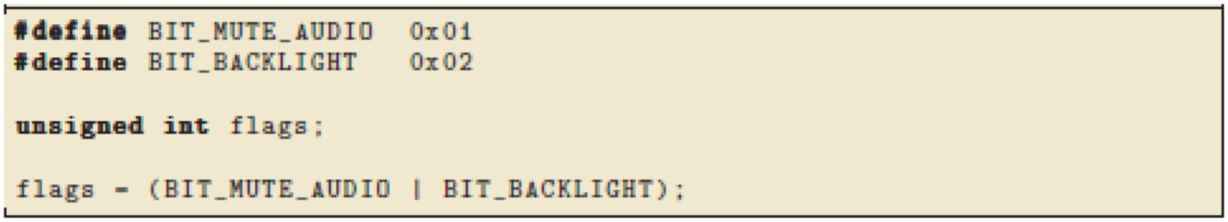
```
void main (void)
{
   short a = 10;
   short b = 13;

   swap (a,b);

   /* a == ?, b == ? */
}

void swap (short x, short y)
{
   short temp;

   temp = x;
   x = y;
   y = temp;
}
```





```
void main (void)
{
   short a = 10;
   short b = 13;

   /* Now we pass the address of the variables we want to change. */
   swap (&a,&b);

   /* a == ?, b == ? */
}

void swap (short *x, short *y)
{
   short temp;

   temp = *x;
   *x = *y;
   *y = temp;
}
```

Bit Fields:



```
#define BIT_MUTE_AUDIO   0x01
#define BIT_BACKLIGHT    0x02

unsigned int flags;

flags = (BIT_MUTE_AUDIO | BIT_BACKLIGHT);
```