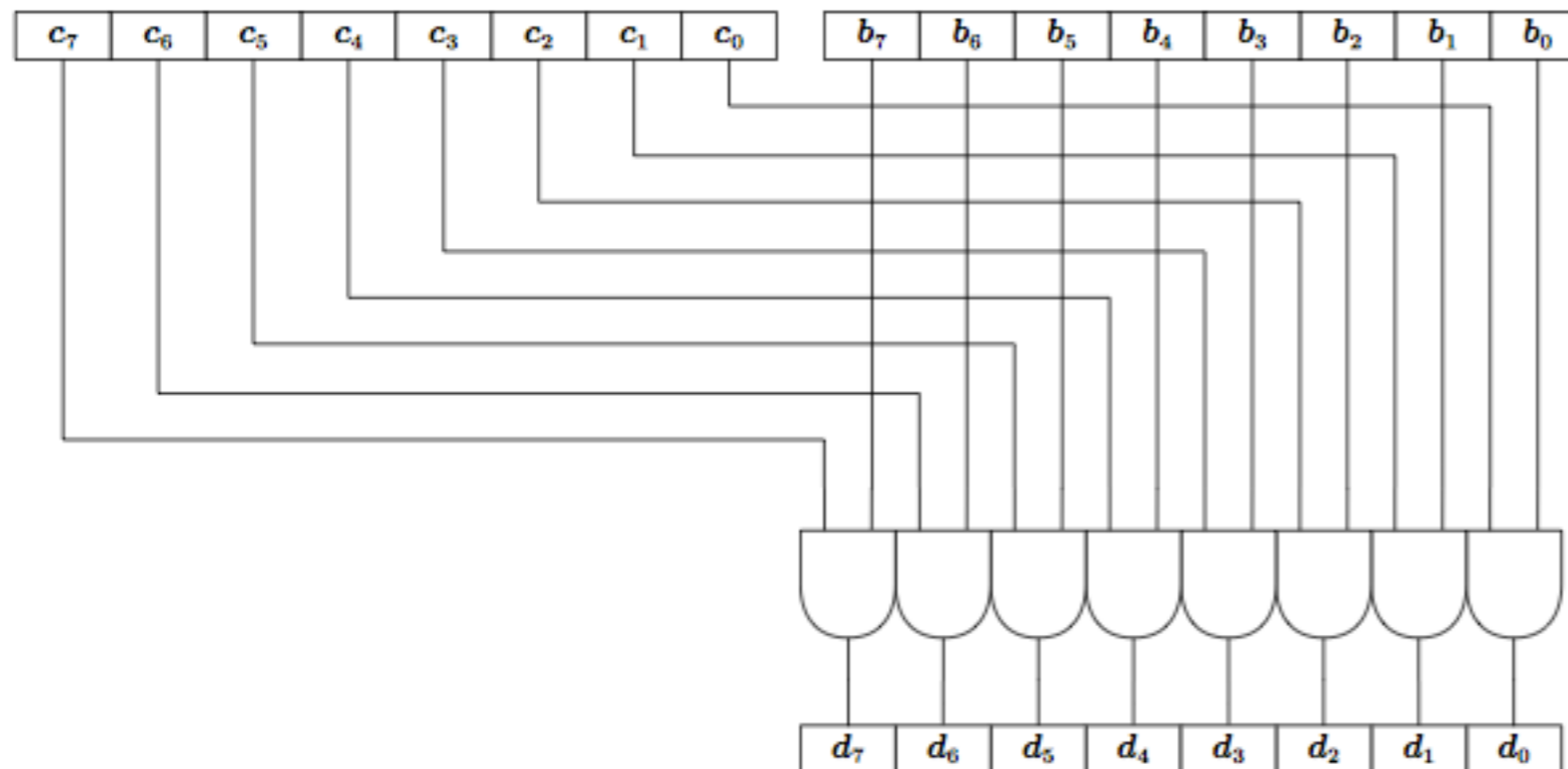


# Bitwise operators, Macros and Enumeration

Reference: Russell Chapter 2


# List of bitwise operators in C

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



- Bitwise operations occur on a per-bit level
- $d = c \& b$
- All three variables are declared as unsigned char (assumed to be 8-bit elements).

# Code example




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

$$0x0F = 0F_{16} = 15_{10} = 0000\ 1111_2$$

$$0x55 = 55_{16} = 85_{10} = 0101\ 0101_2$$

$$0101\ 0101_2 \text{ AND } 0000\ 1111_2 = 0000\ 0101_2 = 05_{16} = 0x05$$

# Code example




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

$$0x0F = 0F_{16} = 15_{10} = 0000\ 1111_2$$

$$0x55 = 55_{16} = 85_{10} = 0101\ 0101_2$$

$$0101\ 0101_2 \text{ OR } 0000\ 1111_2 = 0101\ 1111_2 = 5F_{16} = 0x5F$$

# Code example




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

$$0x0F = 0F_{16} = 15_{10} = 0000\ 1111_2$$

$$0x55 = 55_{16} = 85_{10} = 0101\ 0101_2$$

$$0101\ 0101_2 \text{ XOR } 0000\ 1111_2 = 0101\ 0101_2 = 5A_{16} = 0x5A$$

# Code example



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

$$0x55 = 55_{16} = 85_{10} = 0101\ 0101_2$$

$$0101\ 0101_2 \ll 3 = 1010\ 1000_2 = A8_{16} = 0xA8 = 168_{10}$$

Note that the previous operation overflowed (hence the truncated data)

# Code example

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

$$0x55 = 55_{16} = 85_{10} = 0101\ 0101_2$$

$$\sim 0101\ 0101_2 = 1010\ 1010_2 = AA_{16} = 0xAA$$

# Warning

---

Statement	x	y	z After	Operation
<code>z = (x &amp; y);</code>	1	2	0	Bitwise AND
<code>z = (x &amp;&amp; y);</code>	1	2	1	Logical AND
<code>z = (x   y);</code>	1	2	3	Bitwise OR
<code>z = (x    y);</code>	1	2	1	Logical OR

- There is a difference between logical and bitwise operations.
- All the statements in the table above are perfectly legal, so the compiler will not indicate an error (or even a warning).
- `x && y` will return a boolean variable...
- Since the variables **x** and **y** are both not 0, the logic AND will be 1.

# Macros & enumerated datatypes

# Macros

---

- It's good practice to define all constants (except 0 and 1, where meaning is obvious) in a header file or at the top of the current source file.
- This is done by creating a macro with the preprocessor directive statement `#define`.
- Before compilation occurs, the preprocessor replaces all occurrences of the macro label with its defined value.

```
#define SPECIAL_MAGIC_NUMBER 3

/* skipping many lines */

int primeNumber;

/* skipping many lines */

primeNumber = SPECIAL_MAGIC_NUMBER;
```

# Enumeration

---

- Another way to define constants is via an enumeration.

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

- In the example the first statement defines the constant FALSE, assigns it the default starting value of 0, and then increments the value by one for each successive element in the list; in this case, TRUE is defined as 1.
- Similarly, the constant JAN is defined with an explicit value of 1, and then increments the value by one for each successive element in its list. Thus DEC is defined as 12.

# Enumeration example

---

```
enum month {january=1, february,  
            march};
```

```
int main ()  
{  
    enum month current_month;  
    current_month=january;  
  
    printf("%d",current_month);  
    //this will return 1  
  
    printf("%d",march);  
    //this will return 3  
}
```

```
enum month {january=1, february,  
            march} current_month;
```

```
int main ()  
{  
    current_month=january;  
  
    printf("%d",current_month);  
    //this will return 1  
  
    printf("%d",march);  
    //this will return 3  
}
```

These two codes are the same... however in the right code **current\_month** is a global variable.

# Another enumeration example

---

```
int main()
{
    enum days{Jan=31, Feb=28, Mar=31,
              Apr=30, May=31, Jun=30,
              Jul=31, Aug=31, Sep=30,
              Oct=31, Nov=30, Dec=31};

    //define 'month' variable of type
    //'months'

    enum days month;

    printf("%d\n", month=Feb);
    //Assign integer value via an
    //alias... This will return 28
}
```

# Enumeration error

---

```
//This program will fail to compile because
//'Alex' is in both enum lists.
int main()
{
    enum People1 {Alex=0,    Tracy,    Kristian} Girls;
    enum People2 {William=0, Martin, Alex    Boys;

    switch (Boys)
    {
    case William:
        puts("William"); break;

    case Martin:
        puts("Martin"); break;

    case Alex:
        puts("Alex"); break;

    default:
        break;
    }
}
```

# Another enumeration error

---

```
//This program will fail to compile because the
//preprocessor will change the FALSE to 1 on the enum statement....

#define RAIN 1

main()
{
    enum Weather {RAIN=0, SNOW} todays_weather;

    printf("Rain has a value of %d", RAIN);
    printf("Snow has a value of %d", SNOW);
}
```

# C shortcuts

# Assignment operators

---

Operator	Syntax	Equivalent Operation
<code>+=</code>	<code>i += j;</code>	<code>i = (i + j);</code>
<code>-=</code>	<code>i -= j;</code>	<code>i = (i - j);</code>
<code>*=</code>	<code>i *= j;</code>	<code>i = (i * j);</code>
<code>/=</code>	<code>i /= j;</code>	<code>i = (i / j);</code>
<code>%=</code>	<code>i %= j;</code>	<code>i = (i % j);</code>
<code>&amp;=</code>	<code>i &amp;= j;</code>	<code>i = (i &amp; j);</code>
<code> =</code>	<code>i  = j;</code>	<code>i = (i   j);</code>
<code>^=</code>	<code>i ^= j;</code>	<code>i = (i ^ j);</code>
<code>&lt;&lt;=</code>	<code>i &lt;&lt;= j;</code>	<code>i = (i &lt;&lt; j);</code>
<code>&gt;&gt;=</code>	<code>i &gt;&gt;= j;</code>	<code>i = (i &gt;&gt; j);</code>

# Conditional expressions

---

```
/* This conditional expression... */  
z = (a > b) ? c : d;  
  
/* ...is the same as the following code. */  
if (a > b)  
{  
    z = c;  
}  
else  
{  
    z = d;  
}
```

The conditional expression,

**((expr) ? trueValue : falseValue)**

is a kind-of operator that evaluates any expression and returns a different result based on if the expression is true (i.e., non-zero) or false (i.e., 0).

# Conditional expressions

---

```
/* This conditional expression... */  
z = (a > b) ? c : d;  
  
/* ...is the same as the following code. */  
if (a > b)  
{  
    z = c;  
}  
else  
{  
    z = d;  
}
```

- You should not use it directly as it makes the code harder to read.
- It is useful when you are performing macro definitions:

```
#define MAX(a,b) ((a > b) ? a : b)  
  
/* z will be assigned the maximum of the two values. */  
z = MAX(x,y);
```