

Computer Labs: Introduction to C

2º MIEIC

Pedro F. Souto (`pfs@fe.up.pt`)

September 25, 2016

Contents

C vs. C++

I/O in C

C Pointers

Bitwise and Shift Operators

C Integer Conversion

C vs. C++

- ▶ C++ is a **super**-set of C
 - ▶ C++ has classes – facilitates OO programming
 - ▶ C++ has references – safer and simpler than C pointers
- ▶ It is possible, and often desirable, to use OO programming in C
 - ▶ We'll dedicate a full class to that

Contents

C vs. C++

I/O in C

C Pointers

Bitwise and Shift Operators

C Integer Conversion

I/O in C

- ▶ C provides standard streams for I/O:

`stdin`

`stdout`

`stderr`

- ▶ But C does not have the `cin` and `cout` objects nor the `>>` or the `<<` operators

- ▶ C does not support classes

- ▶ Instead you should use the functions:

`scanf`

- ▶ Not very useful for LCOM

`printf` or `fprintf()`

declared in `<stdio.h>`

printf()

```
printf("video_txt:: vt_print_string(%s, %lu, %lu, 0x%X)\n",  
str, row, col, (unsigned)attr);
```

- ▶ The first argument is the format string, which comprises:
 - ▶ Standard characters, which will be printed verbatim
 - ▶ Conversion specifications, which start with a % character
 - ▶ Format characters, such as \n or \t, for newline and tabs.
- ▶ The syntax of the conversion specifications is somewhat complex, but at least must specify the types of the values to be printed:
 - ▶ %c for a character, %x for an unsigned integer in hexadecimal, %d for an integer in decimal, %u for an unsigned integer in decimal, %l for a long in decimal, %lu for an unsigned long in decimal, %s for a string, %p for an address
- ▶ The remaining arguments should:
 - ▶ Match in number that of conversion specifications;
 - ▶ Have types compatible to those of the corresponding conversion specification
 - ▶ The first conversion specification refers to the 2nd argument, and so on

Contents

C vs. C++

I/O in C

C Pointers

Bitwise and Shift Operators

C Integer Conversion

C Variables and Memory

- ▶ C variables abstract memory, and in particular memory addresses.
- ▶ When we declare a variable, e.g.:

```
int n; /* Signed int variable */
```

what the compiler does is to allocate a region of the process' address space large enough to contain the value of a signed integer variable, usually 4 bytes;

- ▶ Subsequently, while that declaration is in effect (this is usually called the **scope** of the declaration), uses of this variable name translate into accesses to its memory region:

```
n = 2*n; /* Double the value of n */
```

- ▶ However, in C, almost any “real world” program must explicitly use addresses
 - ▶ C++ provides references which are substitutes of C addresses that work in most cases

C Pointers

- ▶ A C pointer is a data type whose value is a memory address.
 - ▶ Program variables are stored in memory
 - ▶ Other C entities are also memory addresses
- ▶ C provides two basic operators to support pointers:
 - & to obtain the address of a variable. E.g.

```
p = &n; /* Initialize pointer p with
        the address of variable n */
```

- * to dereference the pointer, i.e. to read/write the memory positions it refers to.

```
*p = 8; /* Assign the value 8 to memory position
        whose address is
        the value of p (variable n) */
```

- ▶ To declare a pointer (variable), use the * operator:

```
int *p; /* Variable/pointer p points to integers or
        the value pointed to by p is of type int */
```

- ▶ Use of pointers in C is similar to the use of indirect addressing in assembly code, and as prone to errors.

C Pointers as Function Arguments

- ▶ In C, function arguments (or parameters) are passed by value
 - ▶ In a function call the value of the (actual) arguments are copied onto the stack, and then used as values of the function's formal arguments
- ▶ Thus the following code snippet will not work as a naïve C programmer is likely to expect:

```
int    a, b;  
[...]  
swap(a, b);
```

- ▶ To actually swap the values of variables `a` and `b`, you need a different `swap()` function:

```
int    a, b;  
[...]  
swap(&a, &b);
```

- ▶ One of the most common uses of pointers in C is as function arguments to return values from the callee to the caller function
 - ▶ Unlike C++, C does not support **reference variables**

C Pointers and Arrays

- ▶ The elements of an array are stored in consecutive memory positions
- ▶ In C, the name of an array is the address of the first element of that array:

```
int a[5];  
p = a;          /* set p to point to the first element */  
p = &(a[0]); /* same as above */
```

- ▶ C supports pointer arithmetic – meaningful only when used with arrays. E.g. to iterate through the elements of an array using a pointer:

```
for( i = 0, p = a; i < 5; i++, p++) {  
    ...  
}
```

or, without using variable *i*:

```
for( p = a; p-a < 5; p++) {  
    ...  
}
```

IMP: Pointer *p* must be declared to point to variables of the type of the elements of array *a*.

Strings and Pointers in C

- ▶ A string is an array of characters terminated by character code 0x00 (zero), also known as *end of string* character.
 - ▶ In C, a string is completely defined by the address of its first character

```
#define HELLO "Hello, World!"
```

```
...
```

```
char *p = HELLO; /* Set p to point to string HELLO */  
for( len = 0; *p != 0; p++, len++);
```

- ▶ The C standard library provides a set of string operations, that are declared in `<string.h>`

```
#include <string.h>
```

```
...
```

```
char *p = HELLO; /* Set p to point to string HELLO */  
len = strlen(p);
```

- ▶ Array names and string literals are constants not variables. The following is **WRONG**:

```
char a[20];
```

```
a = HELLO; /* This is similar to 2 = 5; */
```

```
HELLO = a; /* Same as above */
```

may use instead:

```
strncpy(a, HELLO, 20); /* Should NOT use strcpy */
```

Contents

C vs. C++

I/O in C

C Pointers

Bitwise and Shift Operators

C Integer Conversion

Bitwise Operations

- ▶ Bitwise operations

- ▶ are boolean operations, either binary or unary
- ▶ take integral operands, i.e. one of the following types `char`, `short`, `int`, `long`, whether signed or unsigned
- ▶ apply the operation on every bit of these operands

y_n $y_1 y_0$

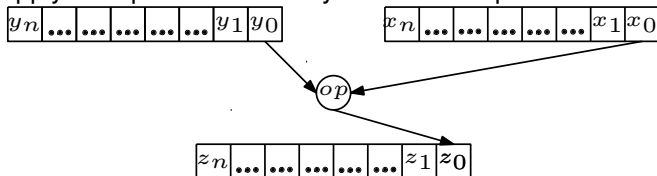
x_n $x_1 x_0$

(op)

z_n $z_1 z_0$

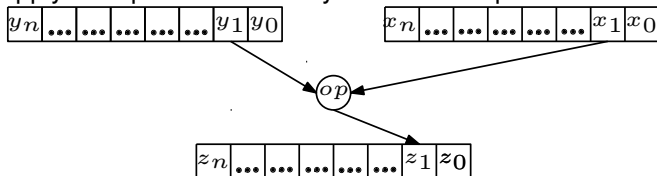
Bitwise Operations

- ▶ Bitwise operations
 - ▶ are boolean operations, either binary or unary
 - ▶ take integral operands, i.e. one of the following types `char`, `short`, `int`, `long`, whether signed or unsigned
 - ▶ apply the operation on every bit of these operands



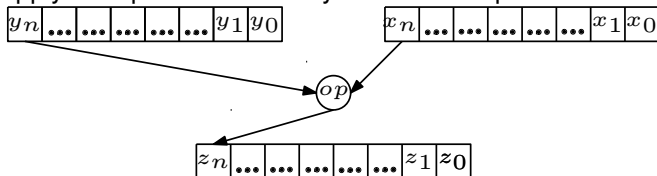
Bitwise Operations

- ▶ Bitwise operations
 - ▶ are boolean operations, either binary or unary
 - ▶ take integral operands, i.e. one of the following types `char`, `short`, `int`, `long`, whether signed or unsigned
 - ▶ apply the operation on every bit of these operands



Bitwise Operations

- ▶ Bitwise operations
 - ▶ are boolean operations, either binary or unary
 - ▶ take integral operands, i.e. one of the following types `char`, `short`, `int`, `long`, whether signed or unsigned
 - ▶ apply the operation on every bit of these operands



Bitwise Operators

- ▶ Bitwise operators:
 - & bitwise AND
 - | bitwise inclusive OR
 - ^ bitwise exclusive OR
 - ~ one's complement (unary)
- ▶ Do not confuse them with the logical operators which evaluate the truth value of an expression:
 - && logical and
 - || logical or
 - ! negation

Bitwise Operators: Application

- ▶ Use with bit masks:

```
uchar mask = 0x80;      // 10000000b
...
if ( flags & mask )     // test value of flags MS bit
    ...
flags = flags | mask;   // set flags MS bit
flags ^= mask;          // toggle flags MS bit
mask = ~mask;           // mask becomes 01111111b
flags &= mask;          // reset flags MS bit
```

- ▶ In Lab 2, you can use the | operator to set the operating mode of the i8254 timer/counter:

```
#define SQR_WAVE 0x06

[...]
```

```
control |= SQR_WAVE;
```

```
[...]
```

Shift Operators

- ▶ Similar to corresponding assembly language shift operations
 - >> right shift of left hand side (LHS) operand by the number of bits positions given by the RHS operand
 - ▶ Vacated bits on the left are filled with:
 - 0 if the LHS is unsigned (logical shift)
 - either 0 or 1 (machine/compiler dependent] if the LHS operand is signed
 - << left shift
 - ▶ Vacated bits on the right are always filled with 0's
 - ▶ LHS operand must be of an integral type
 - ▶ RHS operand must be non-negative

Shift Operators: Application

- ▶ Integer multiplication/division by a power of 2:

```
unsigned int n;
```

```
n <<= 4;    // multiply n by 16 (2^4)
```

```
n >>= 3;    // divide n by 8 (2^3)
```

- ▶ Flags definitions (to avoid mistakes)

```
#define SQR_WAVE_BIT0 1
```

```
#define SQR_WAVE_BIT1 2
```

```
#define BIT(n) (0x1 << (n))
```

```
mode |= BIT(SQR_WAVE_BIT1) | BIT(SQR_WAVE_BIT0);
```

Contents

C vs. C++

I/O in C

C Pointers

Bitwise and Shift Operators

C Integer Conversion

C Integer Conversion Rules

- ▶ C supports different integer types, which differ in their:
 - Signedness** i.e. whether they can represent negative numbers
 - Precision** i.e. the number of bits used in their representation
- ▶ The C standard specifies a set of rules for conversion from one integer type to another integer type so that:
 - ▶ The results of code execution are what the programmer expects
- ▶ One such rule is that:
 - ▶ Operands of arithmetic/logic operators whose type is smaller than `int` are promoted to `int` before performing the operation

the rationale for this is

- ▶ To prevent errors that result from overflow. E.g:

```
signed char cresult, c1, c2, c3;  
c1 = 100;  
c2 = 3;  
c3 = 4;  
cresult = c1 * c2 / c3;
```

Problems Source: CMU SEI

Let:

```
uint8_t port = 0x5a;  
uint8_t result_8 = ( ~port ) >> 4;
```


Problems Source: CMU SEI

Let:

```
uint8_t port = 0x5a;  
uint8_t result_8 = ( ~port ) >> 4;
```

Question: What is the value of `result_8`?

Problems Source: CMU SEI

Let:

```
uint8_t port = 0x5a;  
uint8_t result_8 = ( ~port ) >> 4;
```

Question: What is the value of `result_8`?

Answer: Most likely, you'll think in terms of 8-bit integers:

Expr.	8-bit
<code>port</code>	0x5a
<code>~port</code>	0xa5
<code>(~port) >> 4</code>	0x0a
<code>result_8</code>	0x0a

Problems Source: CMU SEI

Let:

```
uint8_t port = 0x5a;  
uint8_t result_8 = ( ~port ) >> 4;
```

Question: What is the value of `result_8`?

Answer: ... but because of integer promotion, need to think in terms of `sizeof(int)`:

Expr.	8-bit	32-bit
<code>port</code>	<code>0x5a</code>	<code>0x0000005a</code>
<code>~port</code>	<code>0xa5</code>	<code>0xffffffa5</code>
<code>(~port)>>4</code>	<code>0x0a</code>	<code>0xfffffff0</code>
<code>result_8</code>	<code>0x0a</code>	<code>0xfa</code>

Problems Source: CMU SEI

Let:

```
uint8_t port = 0x5a;  
uint8_t result_8 = ( ~port ) >> 4;
```

Question: What is the value of `result_8`?

Answer: ... but because of integer promotion, need to think in terms of `sizeof(int)`:

Expr.	8-bit	32-bit	Solution
<code>port</code>	0x5a	0x0000005a	0x0000005a
<code>~port</code>	0xa5	0xffffffffa5	0xffffffffa5
<code>(uint_8)</code>	N/A	N/A	0xa5
<code>(~port)>>4</code>	0x0a	0xfffffffffa	0x0a
<code>result_8</code>	0x0a	0xfa	0x0a

Solution: One way to fix this is to use a cast on the value after the complement:

```
uint8_t port = 0x5a;  
uint8_t result_8 = (uint8_t) ( ~port ) >> 4;
```

This truncates the result of the complement to its LSB, and therefore the right shift works as expected

Further Reading

- ▶ INT02-C. Understand integer conversion rules