Computer Labs: Introduction to C

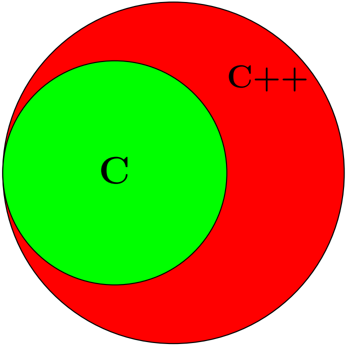
2º L.EIC

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

February 20, 2025

Contents

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

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()

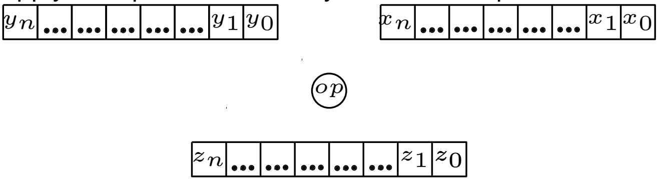
```
printf("video_txt:: vt_print_string(%s, %lu, %lu, 0x%X)\n",
str, row, col, (unsigned)attr);
```

- 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. As a minimum, a conversion specification indicates the type of the value 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

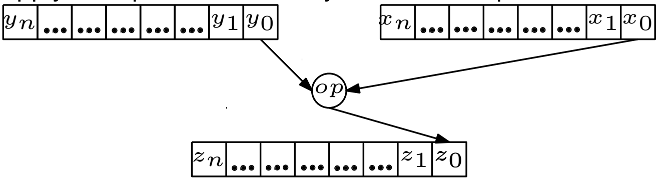
Bitwise Operations

- Bitwise operations
  - are boolean operations, either binary or unary
  - take integer 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



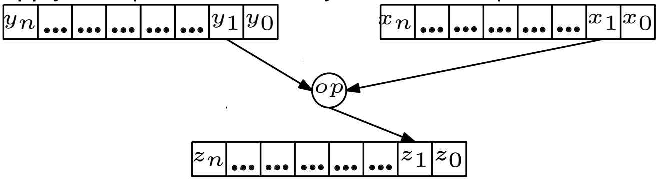
Bitwise Operations

- Bitwise operations
  - are boolean operations, either binary or unary
  - take integer 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



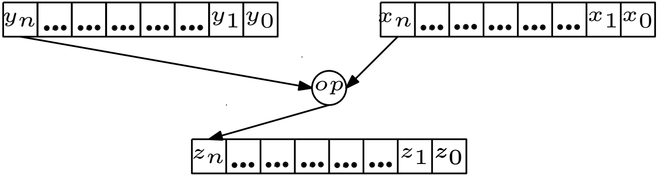
Bitwise Operations

- Bitwise operations
  - are boolean operations, either binary or unary
  - take integer 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 integer 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



  - The CPU performs all these bitwise operations in parallel

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: Use with bit masks

- Testing if MS bit is 1:

```
uchar mask = 0x80;      // 10000000b
...
if ( flags & mask )     // test value of flags MS bit
```

- What if you want to test if the MS bit is 0?
  - The dual:

    ```
    mask = ~mask;             // mask becomes 01111111b
    if( flags | mask )        // could use: flags | ~mask
    ```

    does not work! Why?
- Other operations:

```
flags = flags | mask;   // set flags MS bit
flags ^= mask;          // toggle flags MS bit
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 any integer 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 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 rational 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;
```

source: CMU SEI

Problems Source: CMU SEI

Let:

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

Problems Source: CMU SEI

Let:

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

Question: What is the value of `result_8`?

Problems Source: CMU SEI

Let:

```
uchar port = 0x5a;
uchar 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**
`port`	`0x5a`
`~port`	`0xa5`
`(~port)>>4`	`0x0a`
`result_8`	`0x0a`

Problems Source: CMU SEI

Let:

```
uchar port = 0x5a;
uchar 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
`port`	`0x5a`	`0x0000005a`
`~port`	`0xa5`	`0xffffffa5`
`(~port)>>4`	`0x0a`	`0xfffffffa`
`result_8`	`0x0a`	`0xfa`

Problems Source: CMU SEI

Let:

```
uchar port = 0x5a;
uchar 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
`port`	`0x5a`	`0x0000005a`	`0x0000005a`
`~port`	`0xa5`	`0xffffffa5`	`0xffffffa5`
`(uint_8)`	N/A	N/A	`0xa5`
`(~port)>>4`	`0x0a`	`0xfffffffa`	`0x0a`
`result_8`	`0x0a`	`0xfa`	`0x0a`

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

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

The cast tells the compiler to handle the complement as an unsigned 8 bit integer, and the right shift works as expected

Size of C' Integer Types and <stdint.h>

- To facilitate code portability, C does not specify the size of integer types
  - The range of a given type, say `int`, varies from one platform to another
- However, sometimes we need to use a particular range, independently of the platform
  - E.g. the registers of an I/O controller have a size that is independent of the platform it is integrated into
- In this case, you should use the `<stdint.h>` which includes a set of `typedefs` specifying the size and the signedness of different integer types:

```
uint8_t      int8_t
uint16_t     int16_t
uint32_t     int32_t
uint64_t     int64_t
```

E.g.:

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

Further Reading

- INT02-C. Understand integer conversion rules

Contents

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**

Strings and Pointers in C

- In C, a string is stored as a sequence 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);
```

- String literals are constants not variables. The following is **WRONG**:

```
char *p;
[...]   /* p's initialization; */
HELLO = p;   /* This is similar to: 5 = n,
                                with n an integer variable *
```