Command Line Arguments

Bitwise Operators

Command Line Arguments in C

- Passed to the `main()` function
- `argc` (Argument Count) stores the number of command-line arguments passed
- `argv` (Argument Vector) is an array of character pointers listing all the arguments
- `argv[argc]` is a NULL pointer
- `argv[0]` holds the name of the program
- `argv[1]` first argument
- `argv[argc-1]` last argument

Command Line Arguments

- Parameters/arguments supplied to the program
- A way to influence the behavior during runtime

```c
#include <stdio.h>

// defining main with arguments
int main(int argc, char* argv[]) {
    printf("You have entered %d arguments:\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

```
gcc main.c -o main
```

```
./main geeks for geeks
```

Convert String to Other Types

```c
int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }
    return 0;
}

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s arg1 arg2\n", argv[0]);
        return 1;
    }
}

int value = atoi(argv[1]); // Convert string to integer
```

- Applications: file manipulation or program configuration
- Enhance program flexibility
- More versatile C programs

```
./prog "f1" f2 f3 "this one" -y

argc = 6
argv[0] = prog
argv[1] = f1
argv[2] = f2
argv[3] = f3
argv[4] = this one
argv[5] = -y
```

Bitwise Operators

- Used to manipulate bits and use unsigned integers
- Data manipulations are machine dependent
- Data is represented as sequence of bits
- Value of a single bit is 1 or 0
- Mask operand an integer value with specific bits
    - Used to check or manipulate targeted bits

There are only 10 type of people in the world. Those who understand binary and those who don't.

What is Binary?

- Number system with just two digits (0 and 1)
- Each digit position represents a power of 2
- Rightmost position is 2^0, next left 2^1, then 2^2, etc.
- Also called "base 2" number system
- Binary essential concept for computer science
- Computers operate on binary digits
- Conversion to/from decimal important skill

Binary 1010	Decimal	Decimal 10	Binary
0 x 2^0	= 0	10 / 2	= 5 rem 0
1 x 2^1	= 2	5 / 2	= 2 rem 1
0 x 2^2	= 0	2 / 2	= 1 rem 0
1 x 2^3	= 8	1 / 2	= 0 rem 1
Decimal	= 10	Binary	= 1010

Decimal 29	Binary	Decimal 11101	Decimal
29 / 2	= 14 rem 1	1 x 2^0	= 1
14 / 2	= 7 rem 0	0 x 2^1	= 0
7 / 2	= 3 rem 1	1 x 2^2	= 4
3 / 2	= 1 rem 1	1 x 2^3	= 8
1 / 2	= 0 rem 1	1 x 2^4	= 16
Binary	= 11101	Decimal	= 29

8 Bit Binary Number

- 0110 0101 =
  - 2^6 + 2^5 + 2^2 + 2^0 =
  - 64 + 32 + 4 + 1 = 101 base 10
- 0001 0001 = 2^4 + 2^0 = 16 + 1 = 17 base 10

Representation

- 32-bit `0110 0011 1001 1111 0111 1110 1101 0101`
- 64-bit `0110 0011 1101 1111 0111 1110 0101 0101`
  `0110 0011 1001 1111 0111 1110 0101 0100`

Bitwise Operators

Operator	Use	Description
&	Bitwise AND	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are both 1.
~	One's Compliment	All 0 bits are set to 1 and all 1 bits are set to 0.

NOTE: The combination of pointers and bit-level operators makes C useful for many low level applications.

Bitwise Operators

Operator	Use	Description
|	Bitwise Inclusive OR	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are both 1.
^	Bitwise Exclusive OR (XOR)	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are different.

& bitwise AND vs. && logical AND | *bitwise OR vs.* || *logical OR*

Bitwise Operators

Operator	Use	Description
<<	Left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from the right with 0 bits.
>>	Right Shift	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent when the left operand is negative.

& bitwise AND vs. && logical AND | *bitwise OR vs.* || *logical OR*

Using AND to Display Bits

```
unsigned int displayMask = 1 << 31;
// 10000000 00000000 00000000 00000000

for (unsigned int c = 1, c <= 32; c++) {
    if (value & displayMask)
        { printf("1"); }
    else
        { printf("0"); }
        value <<= 1;
}
```

System Dependent

- bitMask shifted 31 to place 1 in the leftmost bit
- The loop should iterate 32 times
- Assumed unsigned ints are always stored in 32 bits
- Computers use 32-bit or 64-bit word hardware
- Use symbolic constant `CHAR_BIT` (defined `<limits.h>`)
- Program is more cross platform compatible

```
#include <limits.h>
// Number of left shifts
CHAR_BIT * sizeof(unsigned int) - 1;
// Number of iterations
CHAR_BIT * sizeof(unsigned int);
```

Bitwise Shifting

- 2 shift right is equivalent to division by 4
  - x = 0000 0010 (binary) == 2 (decimal)
  - x >> 2
  - x = 00000000 == 0 (decimal)
- 2 shift left is equivalent to a multiplication by 2
  - x = 00000010 (binary) == 2 (decimal)
  - x << 2
  - x = 00001000 == 8 (decimal)

NOTE: For fast multiplications or division by 2^n.

Bit Fields

- Specify the number of bits stored
- Member of a structure or union
- Enable better memory utilization
- Must be declared as int or unsigned int
- Declare unsigned int or int member name : number of bits

```
struct bitCard {
    unsigned int face : 4; // Values 0-15 available
    unsigned int suit : 2; // Values 0-3 available
    unsigned int color : 1; // Values 0-1 available
};
```

Practical Application

- Disk drive controller a 16-bit status register
  - Represented by a single unsigned short integer
  - Contain multiple values packed together

```
struct disk_register {

  unsigned is_write : 1;
  // 1 bit flag - 0=read op, 1=write op
  unsigned error : 1;
  // 1 bit error flag
  unsigned sector : 7;
  // 7 bit sector number being read/written
  unsigned reserved : 3;
  // Reserved/unused
} status;
```

```
struct DISK_REGISTER  {
   unsigned intready:1;
   unsigned error_occured:1;
   unsigned disk_spinning:1;
   unsigned write_protect:1;
   unsigned head_loaded:1;
   unsigned error_code:8;
   unsigned track:9;
   unsigned sector:5;
   unsigned command:5;
};

struct DISK_REGISTER *disk_reg =
 (struct DISK_REGISTER *) DISK_REGISTER_MEMORY;

/* Define sector and track to start read */
disk_reg->sector = new_sector;
disk_reg->track = new_track;
disk_reg->command = READ;
```

```
/* wait until operation done, ready will be true */
while ( ! disk_reg->ready ) ;

/* check for errors */
if (disk_reg->error_occured) {
// interrogate disk_reg->error_code for error type
  switch (disk_reg->error_code)
    ......
}
```

- Bit fields are a convenient way to express many difficult operations. However, bit fields do suffer from a lack of portability between platforms.

Enumeration Constants

- Set of integers represented by identifiers
- Create symbolic names for integer values
- Improve code readability and maintainability
- Providing meaningful names for constants
- Values start with 0 unless specified
- Incremented by 1

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

int main() {
  const char *monthNames[] = {"", "January",
  "February", "March", "April", "May",
   "June", "July", "August", "September",
    "October", "November", "December"};

  for (enum months month = JAN;
   month <= DEC; month++) {
    printf("%2d%11s\n", month, monthNames[month]);
  }

  return 0;
}
```

Questions?