CZ1106 Problem Solving and Computation II

Module Credit: 4

Duration: Semester 2 - January to April 2007.

Exam Date: 26-Apr-2007 (Morning).

Lecture: Every Wednesday, 12nn to 2pm, LT22.

Tutorial and Lab: TBD at S13-04-13 (CSD Lab), alternate week basis. PC will be used.

Lecturer: Dr Tay Seng Chuan
Senior Lecturer, Department of Physics
Head, IT Unit, Dean's Office
SM2/SM3 Programme Coordinator, Dean's Office
Assistant Hall Master, Temasek Hall

E-mail: scitaysc@nus.edu.sg

Office: S16-02 – Dean's Office at Level 2

Appointment: All are welcome and I advise you to pre-arrange the time and place due to my other commitments. Just give a call or email in advance.

Telephone: 65168752

URL: http://www.physics.nus.edu.sg/~phytaysc

1. Objective

This is the continuation of CZ1102. So my job is to teach you how to solve scientific problems and to perform computation based on your foundation in programming language. In these 14 weeks, I will first teach you more advance programming topics such as **command-line argument, pointers, bit manipulation, dynamic memory allocation, and binary files**, followed by **data structures and its used on numerical and non-numerical algorithms**.

2. What you will achieve at the end of the Course

You will be able to embark on solving computational problems for the rest of your NUS candidature, and in your research and working careers.

3. Course Conduct

- Lab Assignments (30%), Term Test (30%) – closed book, and Final Examination (40%) – closed book.
- Students are expected to work on the tutorial question and lab practical questions before attending the classes. Source files are to be downloaded in your thumb drive or diskette at least 2 days before attending your lesson. Bring them to your lab classes – you cannot trust that the network is always working during your lab session.
- Students should access to course website at least once a week.

4. References

I write **lecturenotes for compulsory reading**. The following contains a list of references:

1. C for Scientists and Engineers, Richard Johnsonbaugh, Martin Kalin, Prentice Hall, ISBN 0-13-320334-4. (40 copies have been ordered in Co-op – behind LT27)

2. Advanced C, Peter D. Hipson, SAM Publishing, ISBN 0-672-30168-7.

3. C – An Introduction with Advanced Applications, Prentic Hall, ISBN 0-13-480781-2.

5. Software

Turbo-C will be used. Both CSD labs at S13-04 are installed with Turbo-C. Lab will also be open from 6pm to 9pm starting from 3rd week.

Chapter 1

Passing Arguments to main Function from Command Line

1. Introduction

- As you have learnt in your first programming course, you have seen that a C program starts and ends with the function **main**.

- Just likes any function, the **main** function can also accept arguments.

- All functions in C, including **main**, have exactly the same structure :
  - a name and an optional argument list.
  - at least one block of executable code (which may be empty)
  - may optionally return a value.

- Program execution starts from the function **main**.

- To make our programs more flexible, we will learn the method to pass arguments to **main** function from command line.

Consider a program used to encode a text file:

If a character is capital letter, encode it as follows :

Actual Character	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Encoded Character	K L M N O P Q R S T U V W X Y Z A B C D E F G H I J

If a character is small letter, encode it as follows :

Actual Character	a b c d e f g h i j k l m n o p q r s t u v w x y z
Encoded Character	z y x w v u t s r q p o n m l k j i h g f e d c b a

Other characters remain unchanged. Store the ciphered text (encoded data) in a file.

Also, write another program to read the ciphered text and decipher it. Display the plain text on the screen.

We want to make use of this program to encode a text file named monkey.inf, and store the ciphered text in donkey.inf. Here's the test-run result.

Input File (monkey.inf)

```
When I call my worker to do something for me, I also give him
some stuff so that he can do the job.
If I ask you to compute the area of a circle, I will have to
give you the radius.
```

Output File (donkey.ouf)

```
Gsvm S xzoo nb dlipvi gl wl hlnvgsrmt uli nv, S zohl trev srn
hlnv hgfuu hl gszg sv xzm wl gsv qly.
Su S zhp blf gl xlnkfgv gsv zivz lu z xrixov, S droo szev gl
trev blf gsv izwrfh.
```


Program

```
/* AC1-1.C */

#include<stdio.h>

main()
{
  FILE *indata,*outdata;
  char this1;
  indata=fopen("monkey.inf","r");
  outdata=fopen("donkey.ouf","w");
  while (fscanf(indata,"%c",&this1)==1)
  {
    if (this1>='A' && this1<='Z')
      fprintf(outdata,"%c", (this1 - 2*'A'+'K')%26+ 'A') ;
    else
      if (this1>='a' && this1<='z') fprintf(outdata,"%c",'z'-(this1-'a'));
      else
           fprintf(outdata,"%c",this1);
  }

  fclose(indata);
  fclose(outdata);

  return 0;
}
```

What is the Concern (not really a problem)

- Program is not flexible; the input and output filename.ext are all hard-coded.
- Suppose you have 100 messages (stored in 100 separate files) to be encoded before transmission, then you will have to edit and re-compile your program 100 times. By that time most probably your messages are already ***out-dated !!!!***

Solution !!!

- *Make the input and output filename.ext as variables.*
- *When you run the program from the command line, also pass in the input and output files.*

argc : number of arguments
argv : an array of string pointers to character string

In the subsequent discussion, filename refers to the descriptor of a file, e.g., a:\\monkey.inf or monkey.inf).

Program (flexible version, encode.c)

```
/* AC1-2.C */

#include<stdio.h>

main(int argc, char *argv[])
{
  FILE *indata,*outdata;
  char this1;

  indata=fopen(argv[1],"r");
  outdata=fopen(argv[2],"w");

  while (fscanf(indata,"%c",&this1)==1)
  {
    if (this1>='A' && this1<='Z')
      fprintf(outdata,"%c", (this1-2*'A'+'K')%26+'A');
    else
      if (this1>='a' && this1<='z')
        fprintf(outdata,"%c",'z'-(this1-'a'));
      else
        fprintf(outdata,"%c",this1);
  }

  fclose (indata);
  fclose (outdata);

  return 0;
}
```

How to use this program ?

. Edit and compile the program to generate the object code.

. Run the object code to generate the executable code. If the source file is named encode.c, the default executable code will be saved in encode.exe, but you can change the name if it is desired.

. Suppose the executable file is named encode.exe. At the prompt sign, use this command to encode a plain file:

c:\> **encode monkey.inf donkey.ouf**

which means that we run the encode program using monkey.inf (plain text) as an input file, and store the output in donkey.ouf (ciphered text).

If an agent has 100 text files (secret0.inf, secret1.inf, ... , secret99.inf) to be encoded, he can simply use the following instructions with the encode.exe :

encode secret0.inf send0.ouf
encode secret1.inf send1.ouf
encode secret2.inf send2.ouf
. .
. .
encode secret99.inf send99.ouf

No recompilation !!!! You cannot afford to compile your program when it is urgent.

So we have addressed the concern.

2. The Command-Line Interface

. In this course we are interested in *argc* and *argv*.

. Others arguments may be passed as well but we will not discuss them in this course.

. *argc* is an **int** and contains the number of arguments found.

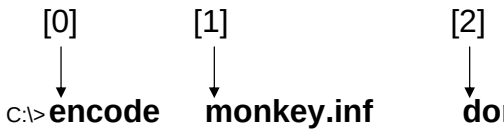
. *argv* is **an array of string pointers** to these arguments.

. The format of a command line instruction is as follows :

progname arg1 arg2 ... arg*n*

- progname is the filename of the executable code.
- arg1 through arg*n* are optional arguments.
- Command-line argument is deemed to be a non-white-space character string.
- Consecutive arguments are separated by multiple spaces and/or horizontal tabs.
- Arguments are always considered to be **character strings**. (e.g., the integer argument 12345 is interpreted as ***string*** "12345").

Example :



argc = 3 (number of command-line arguments)
argv[0] → "encode\0"
argv[1] → "monkey.inf\0"
argv[2] → "donkey.ouf\0"

- **Take note that argv[0] to argv[2] are pointers to string characters.**
- '\0' **is a string terminator and is automatically appended to the end.**
- → **means "it is pointing to ...".**
- '\0' ***is one character by itself.***

Example :

c:\>**encode secret7.inf send7.ouf**

argc = 3
argv[0] → "encode\0"
argv[1] → "secret7.inf\0"
argv[2] → "send7.ouf\0"

A command-line text processor

textpro infile.inf outfile.ouf /spacing=2 /lm=1 /rm=80

argc = 6
argv[0] → "textpro\0"
argv[1] → "infile.inf\0"
argv[2] → "outfile.ouf\0"
argv[3] → "/spacing=2\0"
argv[4] → "/lm=1\0"
argv[5] → "/rm=80\0"

The advantage of this approach is that the same raw document file can be formatted quite differently just by running textpro with different combinations of command-line arguments. The raw (source) file need not be changed!!

Sales Talk!! Anyway, there are serious uses of command-line arguments.

Now we study the flexible version of encode.c again.

C:\> encode monkey.inf donkey.ouf

Program (flexible version)

```
/* AC1-2.C */

#include<stdio.h>

main(int argc, char *argv[])
{
  FILE *indata,*outdata;
  char this1;

  indata=fopen(argv[1],"r");
  outdata=fopen(argv[2],"w");

  while (fscanf(indata,"%c",&this1)==1)
  {
    if (this1>='A' && this1<='Z')
      fprintf(outdata,"%c", (this1-2*'A'+'K')%26+'A');
    else
      if (this1>='a' && this1<='z')
        fprintf(outdata,"%c",'z'-(this1-'a'));
      else
        fprintf(outdata,"%c",this1);
  }

  fclose(indata);
  fclose(outdata);

  return 0;
}
```

---

encode.c

In this example I have not made use of **argc**, I will show you the use of argc in another program.

3. A Case Study

The next programe convert a number in base 10 to its equivalent in base b where is not greater than 8 (for the ease of explanation. You can modify the program to any base value.).

For examples :

Command Line	Response
convert 30 /base=4	134
convert 35 /base=2	100011
convert -13 /base=4	-31
convert 65 /base=7	122
convert 169 /base=9	Base should be less than 9!
convert 13	Wrong Format! Use this instruction : convert dd...d /base=d

Let's do it manually first.

$30_{(10)}$ = _____ (4)

c:\> **convert 30 /base=4**

Program

```
/* AC1-3.C */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define SIZE 10

main(int argc, char *argv[])
{
  int digit, base, remainder, i;
  char symbol[SIZE];

  if (argc!=3)
  {
     printf("Wrong Format! Use this instruction:\n");
     printf("convert dd...d /base=d\n");
     exit(1);
  }

  if (strlen(argv[2] ) != 7)
  {
    printf("Base Format Error!\n");
    exit(1);
  }

  if (*(argv[2]+6) > '8')
  {
    printf("Base should be less than 9!\n");
    exit(1);
  }
```



```
  base = *(argv[2]+6)-'0';
  digit = atoi(argv[1]);
  printf("\t\t");
  if (digit<0)
  {
    putchar('-');
    digit = -digit;
  }

  i=0;
  do
  {
    remainder = digit%base;
    symbol[i] = remainder + '0';
    i++;
    digit = digit/base;
  } while(digit>0);

  do
  {
    i--;
    putchar(symbol[i]);
  } while(i!=0);
  putchar('\n');

  return 0;
}
```

Screen Output

C:\> *convert 30 /base=4*
132

C:\> *convert 35 /base=2*
100011

C:\> *convert -13 /base=4*
-31

C:\> *convert 100 /base=5*
400

C:\> *convert 65 /base=7*
122

C:\> *convert 169 /base=9*
Base should be less than 9!

C:\> *convert 13*
Wrong Format! Use this instruction :
convert dd...d /base=d

Chapter 2

Bit Operations

1. Decimal System

Example : $N = 152_{10}$.

- Subscript "$_{10}$" means base 10.
- Each character is a digit with a corresponding weight.

$N = 1 \times 10^2 + 5 \times 10^1 + 2 \times 10^0 = 152_{10}$

2. Binary System

- Computers are built from devices which behave like switches having only two possible states (and not ten possible states), i.e. OFF or ON, which may be represented as either 0 or l. Thus computers use the binary system (base 2) to represent numbers.
- Weights are powers of two instead of powers of ten.

Example :

$N = 110_2$
$= 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
$= 4 + 2 + 0$
$= 6_{10}$

Decimal Sum

carry digits: **11**

```
  137
+  69
-----
  206
```

Binary Sum

Addition Rules:

$0_2 + 0_2 = 0_2$, $1_2 + 0_2 = 0_2 + 1_2 = 1_2$, and $1_2 + 1_2 = 10_2$

Binary Addition:

carry digits: 1

```
  1110
+ 1001
------
 10111
```

Subtraction in base 2 involves borrowing *two*s : (Actually, we borrow one, but its weight is 2)

carry digits: 22 / -1-1

```
  100
-  11
-----
  001
```


Data Representation used in Computer

- Computers use the binary system to represent numbers internally.
- Binary digits are also called **BIT**s. Bits may exist in the computer as *electrical voltages*, for example +5 Volts might be used to represent a binary 1, and -5 Volts may be used to represent a binary 0. Alternatively they may exist as *charge on capacitors*, a charged capacitor representing a binary 1, and an uncharged capacitor representing a binary 0.
- Bits are stored in groups of 8, called Byte.
- Word varies from one computer to the others. In this course, we assume that 1 word = 2 bytes.
- By our convention the computer uses the leftmost binary digit to determine the sign (+ or -) of a number: the number being positive if the leftmost bit is zero and negative if the leftmost bit is a one.
- Suppose our computer uses 8 bits (i.e., 1 byte) to represent integers. In this course, **the 8 bits are numbered 0 through 7 from right to left in our convention**! Bit number 7 is the Most Significant Bit (MSB) and bit number 0 is the Least Significant Bit (LSB).

MSB							LSB
7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	1

To remember, think of the amount of money you have : $1239 where 9 is the least significant number (only $9), and 1 is most significant number ($1000!!)

The number represented here is positive (because bit-7 is 0), and has the value:

MSB							LSB
7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	1

$1 \times 2^6 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0$
$= 64 + 8 + 4 + 1$
$= 77_{10}$

- The biggest positive integer that may be represented using 8-bit two's complement notation is $+127_{10}$ (01111111_2).
- What is the most negative number represented by this notation ? We will see.

How to represent -77_{10} as a binary byte?

- We use two's complement notation for -ve numbers.
- First, take the binary representation of 77 and toggle all the 1s to 0s, and all the 0s to 1s to obtain the one's complement notation.

So,
$+77_{10} = 01001101_2$

becomes,
10110010_2 (after toggling all bits)

Next, add 1 to the one's complement notation to obtain the two's complement notation.

Now, the number becomes

MSB							LSB
7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	1

So,

$-77_{10} = 10110011_2$ in two's complement notation.

How does computer calculate +77 + (-77) ?

carry digits 11111111

```
  01001101  (+77)
+ 10110011  (-77)
 100000000
```

Binary digit is carried over into position number 8, but it doesn't matter because we only use 8 bits (bits 0 to 7) !!!

Decimal Value of a Two's Complement Notations

For a two's complement 8-bit integer, the decimal value can be computed as follows :

N = (number represented by bits 0 to 6) - (bit 7 x 128)

In general, for a n-bit integer,

N = [number represented by bits 0 to (n-2)] - [bit (n-l) x 2^{n-1}]

MSB							LSB
n-1	n-2	..	..	3	2	1	0
1	0	1	1	0	0	1	1

$10110011_2 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^1 + 1 \times 2^0 - 1 \times 2^7$

$= 32_{10} + 16_{10} + 2_{10} + 1_{10} - 128_{10}$

$= 51_{10} - 128_{10}$

$= -77_{10}$

So, the most negative number we can represent is -128, i.e., 10000000_2 for a 8-bit integer.

Unsigned Integer

If we assumed that numbers were always positive, we could use the MSB to represent value and

our biggest number would be $11111111_2 = 255_{10}$,

the smallest value is $00000000_2 = 0$, and

the number is called an **unsigned integer** because the MSB is no longer used as a sign bit. Now the **MSB carries a weight for unsigned integer**.

By default, integer is signed, unless stated otherwise.

Overflow Problems in Arithmetic

Assume 8-bit integer:

carry digits: 111

$$\begin{array}{rl} & 01110000 \ (112_{10}) \\ + & 00110000 \ (+48_{10}) \\ \hline & 10100000 \end{array}$$

sign bit is set !!!

According to our two's complement notation,

the result is negative !!

Why and what is the cause ???

Another way to run into trouble is to add together two negative numbers whose sum is less than -128_{10}.

For instance $(-111_{10}) + (-39_{10})$.

carry digits: 1 1 1

$$\begin{array}{rll} & 10010001 & (-111_{10}) \\ + & 11011001 & (-\ 39_{10}) \\ \hline & 01101010 & \end{array}$$

sign bit is not set

The result is positive !!!

CARRY bit and **OVERFLOW** bit are use to monitor addition.

- The carry bit acts as bit number 8 and catches any carry over from bit number 7.
- In the last example, after the computer had performed the addition the carry bit would be set (1). If there is no carry out from bit 7 the carry bit will be clear (0).
- The overflow bit is there to tell the computer when an addition has gone wrong i.e., when the result cannot fit in the byte it overflows.

The following two digits are used to set the overflow bit :

(i) the carry digit (C_7) from bit 7 (sign bit) to the carry bit

(ii) the carry digit (C_6) from bit 6 to bit 7

C_7	C_6	Overflow Bit
0	0	0
0	1	1
1	0	1
1	1	0

The relation is known as Exclusive-OR (XOR), meaning "one ***and only*** one". The overflow bit is the XOR of C_6 and C_7. If the overflow bit is set after an addition, the sum cannot be represented by only 8 bits.

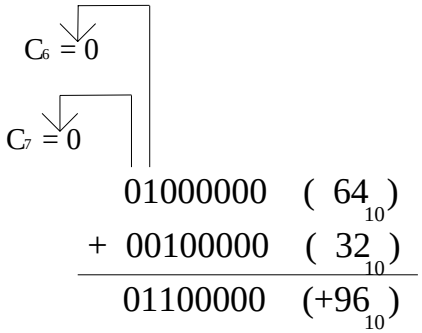
To understand, consider the 2-bit two's complement notations with small numbers: 00 (0), 01 (1), 10 (-2), 11 (-1).

Carry Bit:	0 0	0 1	1 0	0 0
	0 0	0 1	1 0	1 1
	0 1	0 1	1 1	1 1
	0 1	1 0	1 1	1 0

Consider 8 bits:

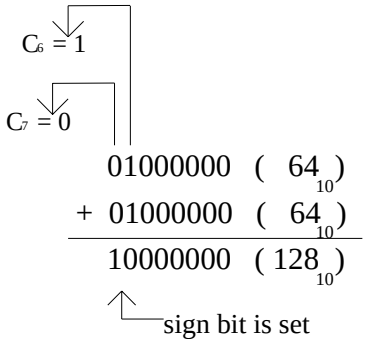
Example for case 1 :

$64_{10} + 32_{10} = 96_{10}$ (no overflow)



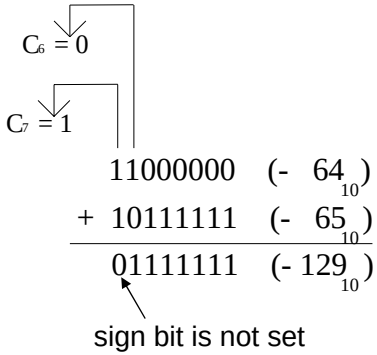
Example for case 2 :

$64_{10} + 64_{10} = 128_{10} > 127_{10}$ (overflow)

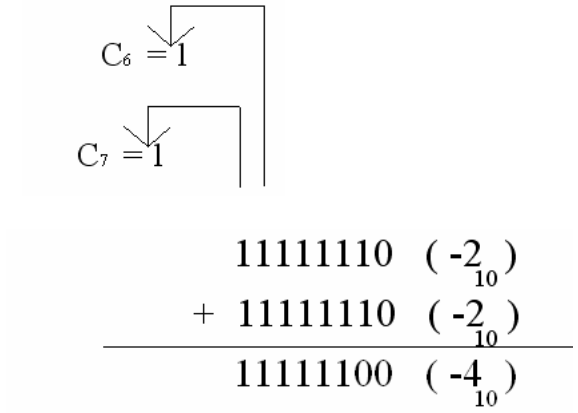


Example for case 3 :

$(-64_{10}) + (-65_{10}) = -129_{10} < -128_{10}$ (overflow)



Example for case 4 : $(-2_{10}) + (-2_{10}) = -4_{10}$ (no overflow)



3. Octal System

$135_8 = 1 \times 8^2 + 3 \times 8^1 + 5 \times 8^0$
$= 64_{10} + 24_{10} + 5_{10}$
$= 93_{10}$

Convert from Binary to Octal

Group of 3 bits !!

$011010101_2 = 325_8$

4. Hexadecimal System

Digits : 0 1 2 3 4 5 6 7 8 9 A B C D E F
(Small letter a b c d e f can be used.)

$AB6_{16} = 10 \times 16^2 + 11 \times 16^1 + 6 \times 16^0$

$= 2560_{10} + 176_{10} + 6_{10}$

$= 2742_{10}$

Conversion of Decimal Numbers to Hexadecimal Numbers

Example : $106_{10} = ??_{16}$

106 / 16 = 6 remainder 10_{10} (A_{16})
6 / 16 = 0 remainder 6_{10} (6_{16})

i.e. $106_{10} = 6A_{16}$

Binary to Hexadecimal

Group of 4 bits

$1010111011110011_2 = AEF3_{16}$

If the number of digits in the binary number is not a multiple of four, we must append the leading zeros to the left hand side. Thus

$11011_2 = \underline{000}11011_2 = 0001\ 1011_2 = 1B_{16}$

When adding and subtracting hexadecimal numbers remember to carry or borrow sixteens!

ASCII Codes

. ASCII is only a 7 bit code, for 128 characters, the 8th bit (bit-7) is used for error checking during data transmission.

. IBM have extended the ASCII codes to include a special set of characters.

	MSBs							
LSBs	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC_1	!	1	A	Q	a	q
0010	STX	DC_2	“	2	B	R	b	r
0011	ETX	DC_3	#	3	C	S	c	s
0100	EOT	DC_4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	O	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Character	ASCII Code
0	0110000
1	0110001
. . .	. . .
9	0111001
:	0111010
A	1000001
B	1000010
. . .	. . .
Z	1011010
[	1011011
\	1011100

The first 32 characters are known as control characters. These are used to control some action of the computer and are not printable characters like the remaining 224 characters. Some points to notice about the table are :

- The decimal digits 0 to 9 have ASCII codes in the range 48_{10} to 57_{10} or 30_{16} to 39_{16} which may be easier to remember because the last digit of the code in hexadecimal is the same as the character represented).
- Uppercase (capital) letters have codes in the range 65_{10} to 90_{10} (i.e. 41_{16} to $5A_{16}$). Lowercase (small) letters have codes in the range 97_{10} to $l22_{10}$ (i.e. 61_{16} to $7A_{16}$). To convert an uppercase character to a lower case character we need to add 32_{10} (20_{16}) to the ASCII code. To convert lowercase to uppercase we would subtract 32_{10} (20_{16}) from the ASCII code.

‘A’ = 65, ‘a’=97