

Lecture 4

- Review
- Control flow
- I/O
 - Standard I/O
 - String I/O
 - File I/O

Blocks

- Blocks combine multiple statements into a single unit.
- Can be used when a single statement is expected.
- Creates a local scope (variables declared inside are local to the block).
- Blocks can be nested.

```
{  
  int x=0;  
  {  
    int y=0; /*both x and y visible */  
  }  
  /*only x visible */  
}
```

Conditional blocks

if ... else..else if is used for conditional branching of execution

```
if ( cond )  
{  
    /*code executed if cond is true */  
}  
else  
{  
    /*code executed if cond is false */  
}
```

Conditional blocks

switch..case is used to test multiple conditions (more efficient than if else ladders).

```
switch ( opt )
{
  case ' A' :
    /*execute   if  opt == 'A' */
    break ;
  case ' B' :
  case ' C' :
    /*execute   if  opt == 'B'  ||  opt == 'C' */
  default  :
}
```

Iterative blocks

- **while** loop tests condition before execution of the block.
- **do..while** loop tests condition after execution of the block.
- **for** loop provides initialization, testing and iteration together.

6.087 Lecture 4 – January 14, 2010

- Review
- **Control flow**
- I/O
 - Standard I/O
 - String I/O
 - File I/O

goto

- **goto** allows you to jump **unconditionally** to arbitrary part of your code (within the same function).
- the location is identified using a label.
- a label is a named location in the code. It has the same form as a variable followed by a ':'

```
start :  
{  
    if ( cond )  
        goto outside ;  
    /*some code */  
    goto start ;  
}  
outside :  
/*outside    block */
```

Spaghetti code

Dijkstra. *Go To Statement Considered Harmful*.
Communications of the ACM 11(3),1968

- Excess use of **goto** creates *sphagetti code*.
- Using **goto** makes code harder to read and debug.
- Any code that uses **goto** can be written without using one.

error handling

Language like C++ and Java provide exception mechanism to recover from errors. In C, **goto** provides a convenient way to exit from nested blocks.

```
for (...)
{
    for (...)
    {
        if (error_cond)
            goto error;
        /*skips 2 blocks */
    }
}
error:

cont_flag = 1;
for (..)
{
    for (init; cont_flag; iter)
    {
        if (error_cond)
        {
            cont_flag = 0;
            break;
        }
        /*inner loop */
    }
    if (!cont_flag) break;
    /*outer loop */
}
```

6.087 Lecture 4 – January 14, 2010

- Review
- Control flow
- I/O
 - Standard I/O
 - String I/O
 - File I/O

Preliminaries

- Input and output facilities are provided by the standard library `<stdio.h>` and not by the language itself.
- A text stream consists of a series of lines ending with `'\n'`. The standard library takes care of conversion from `'\r\n'` - `'\n'`
- A binary stream consists of a series of raw bytes.
- The streams provided by standard library are **buffered**.

Standard input and output

int putchar(**int**)

- putchar(c) puts the character c on the *standard output*.
- it returns the character printed or EOF on error.

int getchar()

- returns the next character from *standard input*.
- it returns EOF on error.

Standard input and output

What does the following code do?

```
int main ()
{
    char c;
    while ((c=getchar ())!= EOF)
    {
        if ( c>=' A'  && c<= ' Z' )
            c=c-' A' +' a' ;
        putchar (c);
    }
    return  0;
}
```

To use a file instead of standard input, use '`<`' operator (*nix).

- Normal invocation: `./a.out`
- Input redirection: `a.out < file.txt`. Treats `file.txt` as source of standard input. This is an OS feature, not a language feature.

Standard output:formatted

int printf (**char** format[],arg1,arg2 ,...)

- printf() can be used for formatted output.
- It takes in a **variable** number of arguments.
- It returns the number of characters printed.
- The format can contain literal strings as well as format specifiers (starts with %).

Examples:

```
printf ( "hello world\n" );  
printf ( "%d\n" ,10);/ *format: %d (integer),argument:10 */  
printf ( "Prices:%d and %d\n" ,10 ,20);
```

printf format specification

The format specification has the following components

%[flags][width][. precision][length]<type>

type:

type	meaning	example
d,i	integer	printf ("%d",10); /*prints 10 */
x,X	integer (hex)	printf ("%x",10); /*print 0xa */
u	unsigned integer	printf ("%u",10); /*prints 10 */
c	character	printf ("%c",' A '); /*prints A */
s	string	printf ("%s","hello"); /*prints hello*/
f	float	printf ("%f",2.3); /*prints 2.3 */
d	double	printf ("%d",2.3); /*prints 2.3 */
e,E	float(exp)	1e3,1.2E3,1E-3
%	literal %	printf ("%d %%",10); /*prints 10%*/

printf format specification (cont.)

%[flags][width][. precision][modifier]<type>

width:

format	output
printf ("%d",10)	"10"
printf ("%4d",10)	bb10 (b:space)
printf ("%s","hello")	hello
printf ("%7s","hello")	bbhello

printf format specification (cont.)

%[flags][width][. precision][modifier]<type>

flag:

format	output
printf ("%d, %+d, %+d",10,-10)	10,+10,-10
printf ("%04d",10)	0010
printf ("%7s","hello")	bbhello
printf ("% -7s","hello")	hellobb

printf format specification (cont.)

%[flags][width][. precision][modifier]<type>

precision:

format	output
<code>printf ("%.2f, %.0f", 1.141, 1.141)</code>	1.14,1
<code>printf ("%.2e, %.0e", 1.141, 100.00)</code>	1.14e+00,1e+02
<code>printf ("%.4s", "hello")</code>	hell
<code>printf ("%.1s", "hello")</code>	h

printf format specification (cont.)

`%[flags][width][. precision][modifier]<type>`

modifier:

modifier	meaning
h	interpreted as short. Use with i,d,o,u,x
l	interpreted as long. Use with i,d,o,u,x
L	interpreted as double. Use with e,f,g

Digression: character arrays

Since we will be reading and writing strings, here is a brief digression

- strings are represented as an array of characters
- C does not restrict the length of the string. The end of the string is specified using 0.

For instance, "hello" is represented using the array
{'h','e','l','l','\0'}.

Declaration examples:

- `char str []="hello";` **compiler takes care of size**
- `char str[10]="hello";` **make sure the array is large enough**
- `char str []={'h','e','l','l',0};`

Note: use `\` if you want the string to contain `"`.

Digression: character arrays

Comparing strings: the header file `<string.h>` provides the function `int strcmp(char s[], char t [])` that compares two strings in dictionary order (lower case letters come **after** capital case).

- the function returns a value <0 if `s` comes before `t`
- the function return a value 0 if `s` is the same as `t`
- the function return a value >0 if `s` comes after `t`
- `strcmp` is case sensitive

Examples

- `strcmp("A", "a") / * <0 *`
- `strcmp("IRONMAN", "BATMAN") / * >0 *`
- `strcmp("aA", "aA") / * $=0$ *`
- `strcmp("aA", "a") / * >0 *`

Formatted input

int scanf(**char***format ,...) is the input analog of printf.

- scanf reads characters from standard input, interpreting them according to format specification
- Similar to printf , scanf also takes variable number of arguments.
- The format specification is the same as that for printf
- When multiple items are to be read, each item is assumed to be separated by white space.
- It returns the number of **items** read or EOF.
- **Important:** scanf ignores white spaces.
- **Important:** Arguments have to be address of variables (pointers).

Formatted input

int scanf(**char***format ,...) is the input analog of printf.

Examples:

printf ("%d",x)	scanf("%d",&x)
printf ("%10d",x)	scanf("%d",&x)
printf ("%f",f)	scanf("%f",&f)
printf ("%s",str)	scanf("%s",str) / *note no & required*
printf ("%s",str)	scanf("%20s",str) / *note no & required*
printf ("%s %s",fname,lname)	scanf("%20s %20s",fname,lname)

String input/output

Instead of writing to the standard output, the formatted data can be written to or read from character arrays.

int sprintf (**char** string [], **char** format[],arg1,arg2)

- The format specification is the same as printf.
- The output is written to string (does not check size).
- Returns the number of character written or negative value on error.

int sscanf(**char** str [], **char** format[],arg1,arg2)

- The format specification is the same as scanf;
- The input is read from str variable.
- Returns the number of items read or negative value on error.

File I/O

So far, we have read from the standard input and written to the standard output. C allows us to read data from text/binary files using `fopen()`.

`FILE*fopen(char name[],char mode[])`

- mode can be "r" (read only),"w" (write only),"a" (append) among other options. "b" can be appended for binary files.
- `fopen` returns a **pointer** to the file stream if it exists or `NULL` otherwise.
- We don't need to know the details of the `FILE` data type.
- **Important:** The standard input and output are also `FILE*` datatypes (`stdin,stdout`).
- **Important:** `stderr` corresponds to standard error output(different from `stdout`).

File I/O(cont.)

int fclose(FILE *fp)

- closes the stream (releases OS resources).
- fclose() is automatically called on all open files when program terminates.

File input

int getc(FILE*fp)

- reads a single character from the stream.
- returns the character read or EOF on error/end of file.

Note: getchar simply uses the standard input to read a character. We can implement it as follows:

```
#define getchar() getc(stdin)
```

char[] fgets(**char** line [], **int** maxlen,FILE*fp)

- reads a single line (upto maxlen characters) from the input stream (including linebreak).
- returns a pointer to the character array that stores the line (read-only)
- return NULL if end of stream.

File output

int putc(**int** c, FILE *fp)

- writes a single character c to the output stream.
- returns the character written or EOF on error.

Note: putchar simply uses the standard output to write a character. We can implement it as follows:

```
#define putchar(c) putc(c,stdout)
```

int fputs(**char** line [], FILE*fp)

- writes a single line to the output stream.
- returns zero on success, EOF otherwise.

int fscanf(FILE *fp,**char** format[],arg1,arg2)

- similar to scanf,sscanf
- reads items from input stream fp.

Command line input

- In addition to taking input from standard input and files, you can also pass input while invoking the program.
- *Command line parameters* are very common in *nix environment.
- So far, we have used `int main()` as to invoke the main function. However, main function can take arguments that are populated when the program is invoked.

Command line input (cont.)

`int` main(`int` argc,`char`*argv[])

- argc: count of arguments.
- argv[]: an array of pointers to each of the arguments
- note: the arguments include the name of the program as well.

Examples:

- `./cat a.txt b.txt` (argc=3,argv[0]="cat" argv[1]="a.txt" argv[2]="b.txt")
- `./cat` (argc=1,argv[0]="cat")

MIT OpenCourseWare
<http://ocw.mit.edu>

6.087 Practical Programming in C

January (IAP) 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.