6.087 Lecture 4 – January 14, 2010

- 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:
}
```

```c
#include <stdio.h>
#include <time.h>

void run_switch(int loops);
void run_ifelse(int loops);

int main() {

    int loops = 100;
    clock_t start_time, end_time;
    double time_elapsed_switch, time_elapsed_ifelse;

    // Benchmark switch statement
    start_time = clock();
    run_switch(loops);
    end_time = clock();
    time_elapsed_switch = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

    // Benchmark if/else ladder
    start_time = clock();
    run_ifelse(loops);
    end_time = clock();
    time_elapsed_ifelse = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

    printf("Switch statement: %f seconds\n", time_elapsed_switch);
    printf("If/else ladder: %f seconds\n", time_elapsed_ifelse);

    return 0;
}
void run_switch(int loops) {
    int input = 3;
    int output = 0;
    int i, j;
    for (j = 0; j < loops; j++) {
        for (i = 0; i < 100000000; i++) {
            switch (input) {
                case 1:
                    output = 10;
                    break;
                case 2:
                    output = 20;
                    break;
                case 3:
                    output = 30;
                    break;
                case 4:
                    output = 40;
                    break;
                default:
                    output = 0;
                    break;
            }
        }
    }
}

void run_ifelse(int loops) {
    int input = 3;
    int output = 0;
    int i, j;
    for (j = 0; j < loops; j++) {
        for (i = 0; i < 100000000; i++) {
            if (input == 1) {
                output = 10;
            } else if (input == 2) {
                output = 20;
            } else if (input == 3) {
                output = 30;
            } else if (input == 4) {
                output = 40;
            } else {
                output = 0;
            }
        }
    }
}
```

```
Switch statement: 20.723417 seconds
If/else ladder: 22.847126 seconds
```

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.

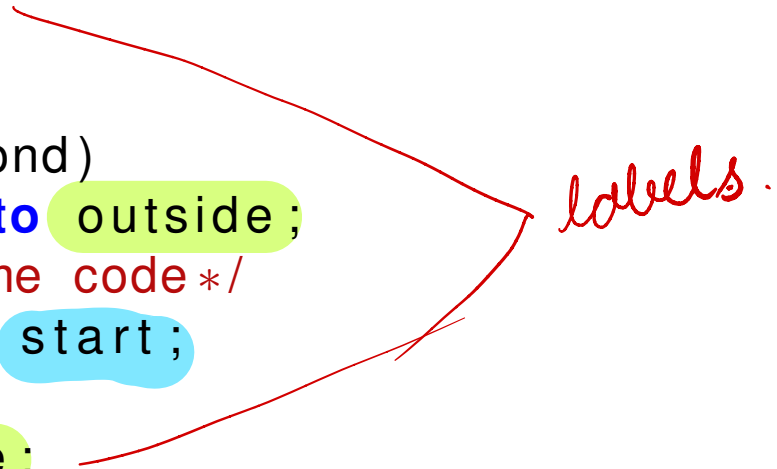
6.087 Lecture 4 – January 14, 2010

- 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:
```

Subroutine to handle exceptions.

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

```c
#include <stdio.h>

int main() {
    int i, j;
    int exit_flag = 0;

    for (i = 1; i <= 10 && !exit_flag; i++) {
        for (j = 1; j <= 10 && !exit_flag; j++) {
            if (i * j > 50) {
                exit_flag = 1;
            }
            else {
                printf("%d * %d = %d\n", i, j, i*j);
            }
        }
    }

    printf("Exited the nested loop at i=%d and j=%d\n", i-1, j-1);

    return 0;
}
```

Without using goto.
- harder if multiple error handling.

```c
#include <stdio.h>

int main() {
    int i, j;

    for (i = 1; i <= 10; i++) {
        for (j = 1; j <= 10; j++) {
            if (i * j > 50) {
                goto exit_loop;
            }
            printf("%d * %d = %d\n", i, j, i*j);
        }
    }

exit_loop:
    printf("Exited the nested loop at i=%d and j=%d\n", i, j);

    return 0;
}
```

Use goto to handle exceptions.
- should be used only if it simplifies code.

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**.

Windows way to create a newline.

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
printf ("%.2f,%.0f,1.141,1.141)	1.14,1
printf ("%.2e,%.0e,1.141,100.00)	1.14e+00,1e+02
printf ("%.4s","hello")	hell
printf ("%.1s","hello")	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

```
printf() format specifications:

type:
    %d       - integer format specifier
    %f       - floating-point format specifier
    %s       - string format specifier
    %c       - character format specifier
    %u       - unsigned integer format specifier
    %o       - octal format specifier
    %p       - pointer format specifier
    %g or %G - general format specifier
    %x or %X - hexadecimal format specifier
    %e or %E - scientific notation format specifier

Flags:
    -   - left-justify output
    +   - always print sign (+ or -)
    #   - alternate form (for example, prefix with 0x or 0X for hex output)
    0   - zero-padding (useful for aligning numbers)
    ' ' - space-pad positive numbers (useful for aligning output)

Width
    Specifies the minimum field width of the output:
        - It can be a positive integer
        - It can be an asterisk (*) to indicate a width argument passed as an additional parameter to printf().

Precision
    Specifies the precision of the output for floating-point number
    - It can be a positive integer
    - It can be an asterisk (*) to indicate a precision argument passed as an additional parameter to printf().
    - For string specifiers, it specifies the maximum number of characters to be printed.

Modifiers:

    hh - for char or unsigned char
    h  - for short or unsigned short
    l  - for long or unsigned long
    ll - for long long or unsigned long long
    j  - for intmax_t or uintmax_t
    z  - for size_t
    t  - for ptrdiff_t
    L  - for long double
```

```c
int x = 42;
double pi = 3.14159265;
char* name = "Alice";
int* ptr = &x;

// basic specifiers
printf("%d\n", x);    // output: 42
printf("%f\n", pi);   // output: 3.141593
printf("%s\n", name); // output: Alice
printf("%p\n", ptr);  // output: 0x7ffee18669a8

// width and precision
printf("%10d\n", x);           // output:         42
printf("%10.2f\n", pi);        // output:      3.14
printf("%-10s\n", name);       // output: Alice
printf("%010d\n", x);          // output: 0000000042
printf("%.3s\n", name);        // output: Ali
printf("%.*f\n", 3, pi);       // output: 3.142

// flags and modifiers
printf("%+d\n", x);            // output: +42
printf("%#x\n", x);            // output: 0x2a
printf("%o\n", x);             // output: 52
printf("%e\n", pi);            // output: 3.141593e+00
printf("%g\n", pi);            // output: 3.14159
printf("%lld\n", 123456789

// Basic format specifiers:
printf("%d", 42);              // prints an integer
printf("%f", 3.14);            // prints a floating-point number
printf("%c", 'a');             // prints a character
printf("%s", "hello");         // prints a string

// Flags:
printf("%+d", 42);             // prints a signed integer with a plus sign
printf("%06d", 42);            // prints an integer with leading zeros
printf("%#x", 42);             // prints a hexadecimal number with a prefix
printf("%-5s", "hi");          // prints a left-justified string with a minimum width of 5

// Modifiers:
printf("%ld", 42L);            // prints a long integer
printf("%hd", 42);             // prints a short integer
printf("%zd", sizeof(int));    // prints a size_t value
printf("%e", 1000.0);          // prints a floating-point number in scientific notation
printf("%p", &x);              // prints a pointer value

// Combining format specifiers:
printf("%-8.3f", 3.14159);     // prints a left-justified floating-point number with a minimum width of 8 and 3 decimal places
printf("%+06d", 42);           // prints a signed integer with a plus sign, leading zeros, and a minimum width of 6
```

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

A-Z < a-z in ASCII.

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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {


    // printf() - prints to stdout
    printf("Hello, world!\n");

    // sprintf() - writes to a string instead of stdout
    char buffer[100];
    sprintf(buffer, "The value of pi is approximately %.2f", 3.14159);
    printf("%s\n", buffer);

    // fprintf() - writes to a file instead of stdout
    FILE *file = fopen("output.txt", "w");
    fprintf(file, "This is written to a file.\n");
    fclose(file);



    // scanf() - reads input from stdin
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    printf("You entered %d.\n", number);

    // sscanf() - reads input from a string instead of stdin
    char input[] = "John Smith 42";
    char name[20];
    int age;
    sscanf(input, "%s %*s %d", name, &age);
    printf("%s is %d years old.\n", name, age);

    // fscanf() - reads input from a file instead of stdin
    file = fopen("input.txt", "r");
    char word[20];
    while (fscanf(file, "%s", word) == 1) {
        printf("%s\n", word);
    }
    fclose(file);
```

```c
    // getc() - reads a single character from stdin
    printf("Enter a single character: ");
    char character = getc(stdin);
    printf("You entered '%c'.\n", character);

    // getchar() - reads a single character from stdin [STRICTLY], but waits for Enter to be pressed
    printf("Press Enter to continue...");
    getchar();
    printf("Continuing.\n");

    // fgets() - reads a line of input from stdin
    char line[100];
    printf("Enter a line of text: ");
    fgets(line, sizeof(line), stdin);
    line[strcspn(line, "\n")] = 0; // remove trailing newline character
    printf("You entered: %s\n", line);




    // putc() - writes a single character to stdout, stderr
    putc('A', stderr);
    printf("\n");

    // putchar() - writes a single character to stdout [STRICTLY]
    char c = 'A';
    putchar(c);  // prints 'A' to stdout

    // fputs() - writes a string to stdout
    fputs("This is a string.\n", stdout);

    return 0;
}
```

```c
int printf(const char *format, ...);
int sprintf(char *str, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);

int scanf(const char *format, ...);
int sscanf(const char *str, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);

int getc(FILE *stream);
int getchar(void);
char *fgets(char *str, int size, FILE *stream);

int putc(int character, FILE *stream);
int putchar(int character);
int fputs(const char *str, FILE *stream);
```

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")

6.087 Practical Programming in C

January (IAP) 2010