Unix
Bash
C
GNU
Systems

Software Systems

Lectures Week 5

Introduction to C

Control Structures – STDIO.H – Arrays and Strings

Prof. Joseph Vybihal

Computer Science

McGill University

Part 1

C Control Structures & Variables

Built-in C Language Types

DESCRIPTION	RESERVED WORD	BITS	RANGE
Integer	short	8	- 128 to + 127
	int	16	+/- 32,768
	long	32	+/- 2,147,483,648
Floating Point	float	32	+/- 3.4 x 10^{38} with 7 significant digits
	double	64	+/- 1.7 x 10^{308} with 15 significant digits
Boolean	short, int, long	(0 is false, other true)	
Character	char, unsigned short int	8	0 to 256
String	char *	32	address in memory (special case of pointer)
Pointers	TYPE*	32	address in memory

Bit sizes depend on CPU

Variable Declaration

Syntax:

```
SCOPE   MODIFIER   TYPE  VAR_NAME;
SCOPE   MODIFIER   TYPE  VAR_NAME = VALUE;
SCOPE   MODIFIER   TYPE  VAR1, VAR2, … , VARn;
```

Where:

SCOPE	- static, extern or it is not used
MODIFIER	- unsigned,short, long or not used
TYPE	- one of the built-in types
VAR_NAME	- must start with a character, any word not beginning with a number or a reserved symbol (like +, -). It is case sensitive: for, For, fOr

Variable Declaration

- int x;
- int x, y, z;
- int x = 5, y, z = 2;
- short int a = -2;
- unsigned short int b = 4;
- char c = 4, d =‘x’;

Constant Declarations

- In C, a variable can be declared as constant.
- The value of a constant is initialized when the variable is declared. That value cannot be changed.

```
int const a = 1;
const int a = 2;
```

typedef Declaration

The typedef command allows for the creation of custom type names. This makes the program more readable.

```
typedef int scalefactor;   // a simple example

int main() {

    scalefactor a;

    a = 10;
    printf("The scale factor is:%d", a);
}
```

typedef Declaration

```
typedef int boolean;
int true=1, false=0;

boolean isValid = false;
```

Operators

Math

Assuming integer math

+	add	x = 5 + 2;	// x becomes 7
-	subtract	x = 5 – 2;	// x becomes 3
*	multiply	x = 5 * 2;	// x becomes 10
/	divide	x = 5 / 2;	// x becomes 2
%	modulo	x = 5 % 2;	// x becomes 1
++	increment	x = x++;	// if x=5 then x=6
--	decrement	x = x--;	// if x=5 then x=4
+=	increment by	x += 3;	// if x=5 then x=8
-=	decrement by	x -= 3;	// if x=5 then x=2
*=	multiply by	x *= 3;	// if x=5 then x=15
/=	divide by	x /= 3;	// if x=5 then x=1

Operators

Logical

• <	less than	5 < 10	true
• >	greater than	5 > 10	false
• <=	less and equal	5 <= 5	true
• >=	greater and equal	5 >= 5	true
• ==	equal	5 == 10	false
• !=	not equal	5 != 10	true
• !	Not	! (5 == 10)	true
• &&	and	(5<10)&&(5>10)	false
• ||	or	(5<10)||(5>10)	true

Expressions

With assignment

- VARIABLE = EXPRESSION;
  - x = 5 + y; // x will contain 5 more than y
  - x = 5 > 10; // results in 1 for true, or 0 for false

Without assignment

- (EXPRESSION)
  - if (x < 10) // true when x is less than 10
  - if (x + 2) // true when result is not equal to zero

Notice the low-level features of C, where logical expressions and integer mathematics mix.

Complex expressions

What do the following output?

- x = x + y++; // assume x = 5 and y = 2
- x = x + ++y;

Standard C definition:
- Var++ → increment after solving the expression
- ++Var → increment before solving the expression

- VAR = (CONDITION) ? TRUE_EXPRESSION : FALSE_EXPRESSION;

- x = (y < 10) ? x++ : x--; // assume x = 5 and y = 2

#include<math.h>

- Standard math:
  - double y = sqrt(double);
  - double y = pow(base,exponent);
  - int x = abs(int);
  - double y = fabs(double);
  - double x = floor(double);
  - double x = ceil(double);
- Trigonometry:
  - sin, cos, tan, asin, acos, atan

```
X = sqrt(25);

X = pos(10,2);
```

```
10.9 -> 10.0

10.2 -> 11
```


```
<math.h>:
        HUGE_VAL    /* large double value */
        double acos(double x);
        double asin(double x);
        double atan(double x);
        double atan2(double y,double x);
        double ceil(double x);
        double cos(double x);
        double cosh(double x);
        double exp(double x);
        double fabs(double x);
        double floor(double x);
        double fmod(double x,double y);
        double frexp(double x,int *expptr);
        double ldexp(double x,int N);
        double log(double x);
        double log10(double x);
        double modf(double x,double *yp);
        double pow(double x,double y);
        double sin(double x);
        double sinh(double x);
        double sqrt(double x);
        double tan(double x);
        double tanh(double x);
```

Control Structures

The if-statement

if (CONDITION) SINGLE_STATEMENT;

```
if (x < 10) puts("X is less than 10\n");
```

if (CONDITION) { MULTIPLE_STATEMENTS; }

```
if (x < 10) {
    puts("X is less than 10\n");
    c = getc(stdin);
}
```

The if-statement

if (CONDITION) SINGLE_STATEMENT; else SINGLE_STATEMENT;

```
if (x < 10) puts("X is less than 10\n");
else puts("X is greater than 10\n");
```

if (CONDITION) { MULTIPLE_STATEMENTS; }

else {MULTIPLE_STATEMENTS;}

```
if (x < 10) {
    puts("X is less than 10\n");
    c = getc(stdin);
} else {
    puts("X is greater than 10\n");
}
```

The switch-statement

```
switch(VARIABLE) {
   case VALUE:     MULTIPLE_STATEMENTS;
                   break;
   case VALUE2:    MULTIPLE_STATEMENTS;
                   break;
   default:
        MULTIPLE_STATEMENTS;
}
```

Optional. It designates the end of the case block. If not present executions automatically goes to the next case block without testing the condition.

The VALUE is a constant, it cannot be a variable.

The VARIABLE can only be of type integer or character.

Some new compilers permit strings.

The switch-statement

```
switch(age) {
 case 1:  puts("You are too young,sorry!\n");
          break;
 case 2:
 case 3: puts("Bring a parent.\n");
          break;
 default:
    puts("You can drive the car.\n");
}
```

We are assuming that the variable AGE is an integer.
If the value is a 1 then the first case is activated only.
If the value is a 2 or 3 then the second case is activated.
All other integer values are handled by the default case.

The switch-statement

```
switch(gender) {
 case 'm':
 case 'M':
        puts("Girls are only welcome.\n");
        break;
 case 'f':
 case 'F':
        puts("Welcome.\n");
        break;
 default:
    puts("Please enter an F or an M.\n");
}
```

The while-loop

while (CONDITION_IS TRUE) SINGLE_STATEMENT;

while (CONDITION_IS_TRUE) { MULTIPLE_STATEMENTS; }

```
int x = 0;
while (x < 10) x++;
```

```
int x = 10;
while(x--);
```

```
int y = 20;
while (y > 0) {
 puts("Hi!");
 y--;
}
```

The do-while-loop

do SINGLE_STATEMENT; while (CONDITION_IS_TRUE);

do { MULTIPLE_STATEMENTS; } while (CONDITION_IS_TRUE);

```
char gender;
do {
     puts("Gender (M or F): ");
     gender = getc(stdin);
} while (gender!='M' && gender!='F');
```

The for-loop

for (START; CONDITION; EXPRESSION) SINGLE_STATEMENT;

for (START; CONDITION; EXPRESSION) { MULTIPLE_STATEMENTS; }

```
int x;
for(x=0; x<10; x++) puts("Hi!");
```

START and EXPRESSION are comma-separated lists.

START, CONDITION, and EXPRESSION are optional !!

```
int x, y;
char c;
for(x=0, y=10, c=' '; x<10 && c != 'x'; x+=2, y--) {
 puts("Hi");
 c = getc(stdin);
}
```

```
for(;;);

for(;x<10;) x--;
```

Part 2

STDIO.H & STDLIB.H

STDIO.H

Defines three forms of I/O:

- Console
- Stream
- Files

Today's lecture will cover Console I/O and Stream I/O

Forms of I/O

Console I/O

- Input and output focused on the keyboard and screen.
  - Other forms: mouse, touch screen.
- Related to the computer the user is interacting with directly.

Stream I/O

- An abstraction.
- A logical or physical device that transmits/consumes n bytes of data, one byte at a time, in a continuous sequence over time.

File I/O

- Reading and writing to a file on disk.
  - This is a special case of stream I/O

STDIN, STDOUT, STDERR

Three standard streams exist in Unix and Linux:

- The input stream (stdin)
  - All C language commands can accept input from stdin.
  - By default stdin is attached to the keyboard.
  - The stdin can be redirected to other input sources.
- The output stream (stdout)
  - All C language commands can write to stdout.
  - By default stdout is attached to the screen.
  - The stdout can be redirected to other output sinks.
- The error stream (stderr)
  - All run-time errors and C error commands write to stderr.
  - By default stderr is attached to the screen.
  - The stderr can be redirected to other output sinks.

Example

We have seen:

c = getc(stdin);

One single character is extracted from the stdin stream, whatever is currently there, and returned.

Example

We have seen:

Bash-prompt $ ls > filename

There is one command: **ls**

Normally the output would show on screen.

The > symbol changes stdout attaching it temporarily to filename.

Example

We have seen:

Bash-prompt $ ls | more

There are two commands: **ls** and **more**.

The stdout for **ls** and the stdin for **more** are attached together.

Stdio Library

<stdio.h>:

constants

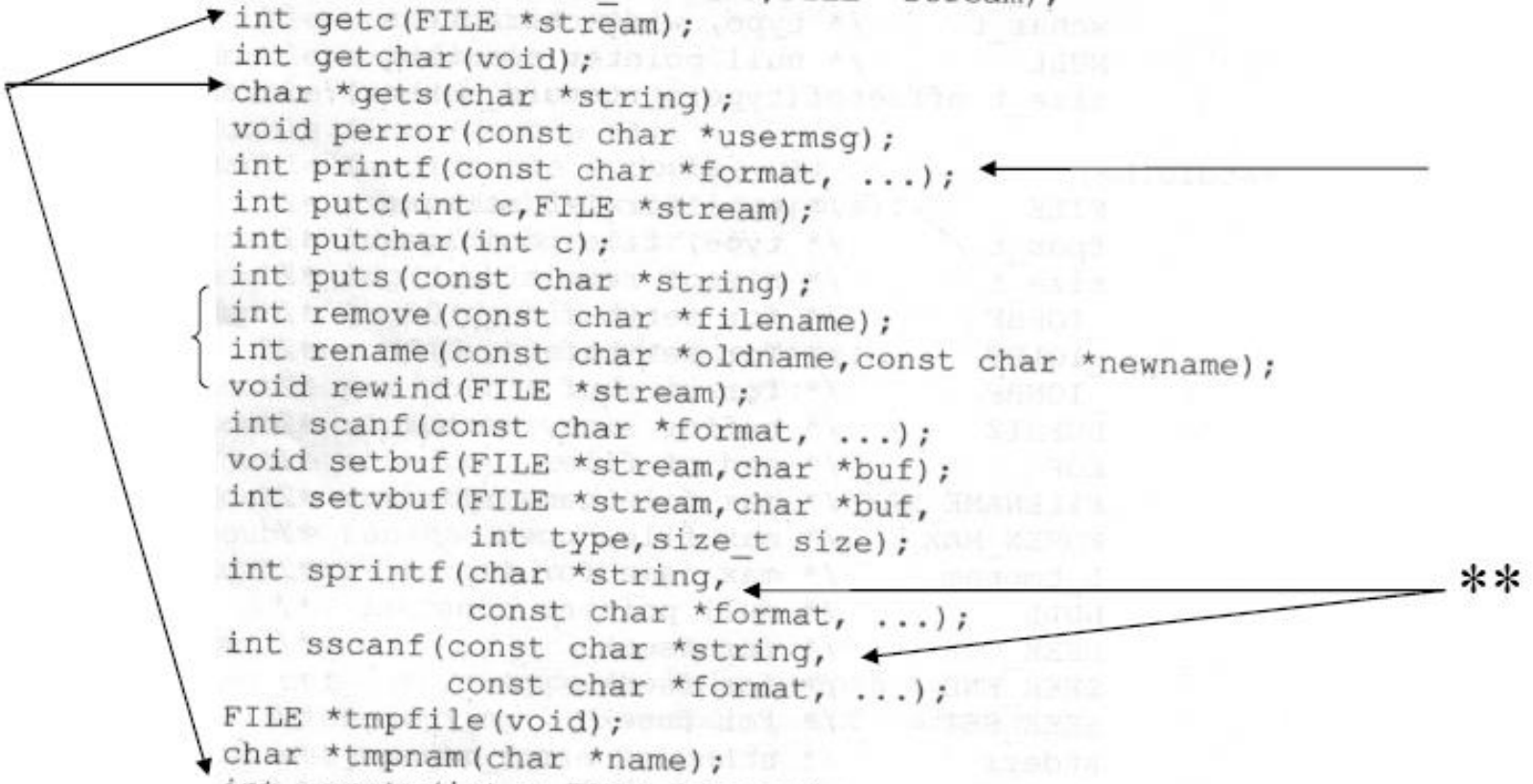
```
FILE            /* type for I/O streams    */
fpos_t          /* type, file position     */
size_t          /* sizeof result           */
_IOFBF          /* for setvbuf             */
_IOLBF          /* for setvbuf             */
_IONBF          /* for setvbuf             */
BUFSIZ          /* buffer size, setbuf     */
EOF             /* end of file             */
FILENAME_MAX    /* max file name size      */
FOPEN_MAX       /* max files open          */
L_tmpnam        /* max name for tmpnam     */
NULL            /* null pointer constant   */
SEEK_CUR        /* for fseek               */
SEEK_END        /* for fseek               */
SEEK_SET        /* for fseek               */
stderr          /* standard error stream   */
stdin           /* standard input stream   */
stdout          /* standard output stream  */
TMP_MAX         /* max tmpnam files        */
```

functions

```
void clearerr(FILE *stream);
int fclose(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
int fflush(FILE *stream);
int fgetc(FILE *stream);
int fgetpos(FILE *stream,fpos_t *pos);
char *fgets(char *string,int n,FILE *stream);
FILE *fopen(const char *name,const char *options);
int fprintf(FILE *stream,const char *format, ...);
int fputc(int c,FILE *stream);
int fputs(const char *string,FILE *stream);
size_t fread(void *ptr,size_t size,
             size_t count,FILE *stream);
```

Stdio Library

```
FILE *freopen(const char *name,
              const char *options,
              FILE *stream);
int fscanf(FILE *stream,const char *format, ...);
int fseek(FILE *stream,long offset,int origin);
int fsetpos(FILE *stream,const fpos_t *pos);
long ftell(FILE *stream);
size_t fwrite(const void *ptr,size_t size,
              size_t count,FILE *stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *string);
void perror(const char *usermsg);
int printf(const char *format, ...);
int putc(int c,FILE *stream);
int putchar(int c);
int puts(const char *string);
int remove(const char *filename);
int rename(const char *oldname,const char *newname);
void rewind(FILE *stream);
int scanf(const char *format, ...);
void setbuf(FILE *stream,char *buf);
int setvbuf(FILE *stream,char *buf,
            int type,size_t size);
int sprintf(char *string,
            const char *format, ...);
int sscanf(const char *string,
           const char *format, ...);
FILE *tmpfile(void);
char *tmpnam(char *name);
int ungetc(int c,FILE *stream);
int vfprintf(FILE *stream,const char *format,
             va_list ap);
int vprintf(const char *format,va_list ap);
```



Inportant STDIO.H functions

- getc, putc, and puts
- getchar, putchar
- fgets
- printf
- scanf
- sprintf
- sscanf

- The file functions are for another lecture…

The getc and puts functions

We have seen these before:

- int getc(STREAM);
  - Where STREAM is stdin, from a file, or other input source
  - It returns the ASCII integer code for the character inputted
    - int asci = getc(stdin);

- int puts(STRING)
  - Where STRING is a series of characters
  - The string is printed to the screen (console)
  - It returns an error code
    - puts(“Hello”);
    - int c = puts(“Hello”);
    - if (puts(“Hello”) == 0) ….

The putc function

Single character stream output:

- int putc(CHARACTER, STREAM);
  - Where STREAM is stdout, to a file, or other output sink
  - Where CHARACTER is a char or int ASCII value
  - It returns error code
    - int errorcode = putc('a', stdout);
    - putc(x, stdout);

The getchar function

This is a console command:

- int getchar(void);
  - The void indicates that there are no arguments.
  - It returns the ASCII of the character read, or error code.
  - It stores the user's entire input into a buffer and then returns a single character from that buffer each time getchar is used.
  - Once the buffer is empty, all the characters have been returned/removed, the function once again stops to read characters into its buffer.
  - When the user presses enter input to the buffer ends.
- Example:
  - int c = getchar(); // user enters: My name is Bob<CR>. Int c = 'M'.
    - Each subsequent call to getchar returns the next char: 'y', then ' ', 'n',…
    - Until there are no more characters
    - When buffer is empty it reads from the console.

Key & Screen Example

```
#include <stdio.h>

int main(void)
{
   char c = '0';

   puts("Input characters until x: ");

   while(c != 'x' && c != 'X')
   {
      c = getchar();
      if (c >= '0' && c <= '9') continue;

      putchar(c); // echo what was read in
   }

   return 0;  // no errors
}
```

What does this do?

The putchar function

This is a console command:

- int putchar(CHARACTER);
  - It returns error code.
  - Outputs the CHARACTER to the screen.
- Example:
  - int errorcode = putchar('A');
  - putchar('B');

The fgets function

This is a stream command:

- POINTER fgets(ARRAY, LIMIT, STREAM);
  - Returns
    - On success a POINTER to the ARRAY.
    - On failure a POINTER to NULL.
    - If no data, or at the end of data, returns POINTER to NULL.
  - Reads at most LIMIT characters from STREAM and store in ARRAY as ASCII codes.
  - The finction fgets inserts a \0 at the end of the stream to indicate the last character. The '\0' character is called the null character.
- Example:
  - char array[30];
  - fgets(array, 29, stdin);
  - char *x = fgets(array, 29, stdin);
  - if (x == NULL) …

I/O Example

```
#include <stdio.h>

int main(void)
{
   char name[30];
   char gender;

   puts("Input your name: ");
   fgets(name, 29, stdin);

   puts("Gender: ");
   gender = getchar();

   puts("Welcome, ");
   puts(name);
   puts(".");

   return 0;
}
```

What does this do?

The printf function

Important console function:

- int printf(STRING, OPTIONAL_ARGUMENTS);
  - Outputs and formats all types of data
  - Returns
    - On success the number of arguments printed to screen.
    - On failure a zero.
  - STRING is the text displayed to the screen.
    - STRING contains escape-character symbols \ and %
    - Example STRING: “I am 12 years old” // simple string, no new line
    - Example STRING: “I am 12 years old \n” // string with new line
    - Example STRING: “I am %d years old \n” // integer inserted into string
- Example:
  - printf(“I am 12 years old.\n”);
  - printf(“I am %d years old.\n”, 12);
  - printf(“I am %d years old.\n”, age);

Escape characters

Escape characters are used for formatting:

- The backslash (\) character
  - \n new line
  - \t tab
  - \a bell
  - \b backspace with no erase
  - \r carriage return
  - \\ backslash
- The percentage (%) character formats variables
  - Format: % SIGN SIZE TYPE
    - %: required
    - SIGN: + -, optional
      - + = normal justification, - = reverse justification
    - SIZE: integer, optional
    - TYPE: d,c,f,s, required
      - d = integer, c = character, f = float, s = string

Examples

```
int age = 12;
printf("I am %d years old.\n", age);
printf("I am %10d years old.\n", age);
printf("I am %-10d years old.\n", age);
```

Numbers are right justified.

Characters and strings are left justified.

Examples

```
float age = 12.5;
printf("I am %f years old.\n", age);
printf("I am %5.1f years old.\n", age);
```

%5.1 → _ _ _ . _

The scanf function

Important console function:

- int scanf(STRING, &VARIABLES);
  - Reads all types of data
  - Returns
    - On success the number of arguments read from keyboard.
    - On failure a zero.
  - STRING is the text format expected from the keyboard.
    - Follows all the same rules we saw with printf
  - VARIABLES
    - At least one variables must be present
    - Leading &
      - Required for regular variables
      - Not used for pointers
- Example:
  - scanf("%d", &age); // reading an integer number into variable age

%s and scanf & printf

In printf %s prints until CR

In scanf %s reads until CR or space

To read an entire sentence use fgets.

Example

```
#include<stdio.h>
int main(void) {
  char name[30]; int age;

  printf("Enter your name: ");
  scanf("%s", name);

  printf("Enter your age: ");
  scanf("%d", &age);

  printf("Welcome %s, you are %d years old.\n", name, age);

  return 0;
}
```

Example

```
#include<stdio.h>
int main(void) {
  int a, b, c;

  printf("Enter three numbers with spaces: ");
  scanf("%d %d %d", &a, &b, &c);

  printf("You entered %d, %d, and %d.\n", a, b, c);

  return 0;
}
```

Important

- The function scanf does not check for the size of the array.
  - If the user enters more characters than the size of the array, C does not crash, it will accept even the extra characters without changing the size of the array, resulting in interesting side effects.
  - This is a system feature allowing programmers to have freedom in accessing and manipulating memory will fewer imposed language rules.

Important

- All input: gets, getc, scanf, etc. do not handle mixed input correctly due to the carriage return issue.
  - %c and %s (or characters and strings) accept the enter key (or carriage return) as a valid input character, and so will read it into the variable.
  - %d and %f (or numbers) only accept numerical values ignoring all other characters.
    - %d will not accept the letter ‘a’ as a valid integer (same for %f)
    - %d and %f will not accept the enter key as a valid number
  - This leads to an interesting usage problem…

Usage Problem

```
#include<stdio.h>
int main(void) {
  int a;
  char array[30];

  // This works
  scanf("%s", array);
  scanf("%d", &a);

  return 0;
}
```

The string entered is saved in array and the integer entered is saved in the variable a.

```
#include<stdio.h>
int main(void) {
  int a;
  char array[30];

  // This fails
  scanf("%d", &a);
  scanf("%s", array);

  return 0;
}
```

The integer entered is stored in the variable a, but the carriage return is still in the buffer. The array reads the carriage return.

Usage Problem Solution

Use a temporary variable to store the carriage return.

- The garbage array captures the carriage return
- The scanf with the array will now wait for the user to input their information

```
#include<stdio.h>
int main(void) {
  int a;
  char array[30];
  char garbage[10];

  // This fails
  scanf("%d", &a);
  scanf("%s", garbage);
  scanf("%s", array);

  return 0;
}
```

The sscanf and sprintf Functions

These two functions are identical to scanf and printf except they do not print to the screen or read from the keyboard:

- sprint(CHAR_ARRAY, STRING, VARIABLES);
- sscanf(CHAR_ARRAY, STRING, VARIABLES);

For sprintf the output goes to CHAR_ARRAY

For sscanf the input comes from CHAR_ARRAY

The sscanf and sprintf Functions

```
char array[100];
int a, b, c;

// Developers assume that users do not follow instructions
printf("Please enter three numbers and press enter at the end: ");
scanf("%s", array);

// If the user input something incorrectly program won't crash, instead zeros
// will be assigned to the offending variable(s)
sscanf(array, "%d %d %d", &a, &b, &c);
```

The sscanf and sprintf Functions

```
char array[100];
char name[30];
int age;
float salary;

// Build a formatted string in memory before you output it
sprintf(array, “Employee: %s, Salary= %6.2f, Age= %3d”, name, salary, age);
```

#include <stdlib.h>

Important Elements

- NULL = 0
- EXIT_FAILURE = 1
- EXIT_SUCCESS = 0
- int x = rand(void); // 0 to RAND_MAX

32767

- int system(string)
- float x = atof(string)
- int y = atoi(string)
- int z = abs(int)
- void exit(int)

Example

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

int main(void)
{
   int randomValue, result, factor = 10;

   randomValue = rand();

   result = factor * randomValue;

   return EXIT_SUCCESS;
}
```

Example

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

int main()
{
    char beingCareful[300];
    int age;
    float salary;

    printf("What is your age?: ");
    gets(beingCarefule);

    age = atoi(beingCareful);

    printf("What is your salary?: ");
    gets(beingCareful);

    salary = atof(beingCareful);

    return EXIT_SUCCESS;
}
```

Example

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

int main(void)
{
   char string[200];

   puts("Please input a command: ");
   gets(string);

   system(string);
}
```

system("ls");
system("./program");
system("cd docs;cp a b; ls");


```
int errorcode = system("./a.out path filename");

// Usage 1 - error / status messages

if (x != EXIT_SUCCESSS)

// Usage 2 - passing messages back

switch(x)
{
case 0: // message 1
case 1: // message 2
case 2: // message 3
}
```

The actual value return by system depends on your OS, but it is based on the shell's error codes.

functions

```
void abort(void);
int abs(int i);
int atexit(void (*wrapfunc)(void));
double atof(const char *s);
int atoi(const char *s);
long atol(const char *s);
void * bsearch(const void *key,
               const void *table,
               size_t N, size_t keysize,
               int (*compar)(const void *,
                             const void *));
void *calloc(size_t N,size_t size);
div_t div(int top,int bottom);
void exit(int status);
void free(void *ptr);
char *getenv(const char *name);
long labs(long n);
ldiv_t ldiv(long top,long bottom);
void *malloc(size_t size);
int mblen(const char *mb,size_t N);
size_t mbstowcs(wchar_t *wcstring,
                const char *mbstring,
                size_t N);
int mbtowc(wchar_t *wc,const char *mb,size_t N);
void qsort(void *table,size_t N,size_t size,
           int (*compar)(const void *,
                         const void *));
int rand(void);
void *realloc(void *oldp,size_t size);
void srand(unsigned seed);
double strtod(const char *s,char **ptr);
long strtol(const char *s,char **ptr,int base);
unsigned long strtoul(const char *s,
                      char **ptr,int base);
int system(const char *command);
size_t wcstombs(char *mbstring,
                const wchar_t *wcstring,
                size_t N);
int wctomb(char *mb, wchar_t wc);
```

**

**

later

**

Part 3

Characters, Arrays and Strings

Character

- A single ASCII integer code
- Syntax:
  - `char VAR1;`
  - `char VAR2 = 'SYMBOL';`
  - `char VAR3 = CODE;`

Example:

```
char x;
char y = 'A';
char z = 92;
```

- Expressions:
  - `char VAR1 = 'SYMBOL1' + 'SYMBOL2'`
    - `char x = 'A' + 'B';`
    - `char y = 'A' + 2;`
  - `char VAR2 = VAR1--;`
    - `char z = x--;`
    - `char z = z - 'A';  // what does this give?`

#include<ctype.h>

- Case manipulation:
  - int c = toupper(int);
  - int c = tolower(int);
- Character testing:
  - int x = isalpha(int);
  - int x = isalphanum(int);
  - int x = isdigit(int);

```
if(toupper(c)=='X')
```

```
if(isalpha(c))
```

Note: char is in int.

<ctype.h>:

```
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

String

- A contiguous static set of characters ending with the NULL character.
- Syntax:
  - `char *VAR1;`
  - `char *VAR2 = "SYMBOLS";`

Example:

```
char *s;
char *s="Bob";
```

- Expressions:

```
char *s1 = "Bob";
char *s2 = "Mary";
s1 = s2; // s1 and s2 equal to "Mary"
printf("My name is %s\n", s1);
```

Important

- A contiguous static set of characters ending with the NULL character.
- Special features:

```
char *s = "Bob";
```

Variable.
It can be assigned to another string.

Static.
The characters that make up this string can never change.

Why does this not work?

```
char *x = "                    ";

scanf("%s", x);
```

Counting Characters

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main()
{
    char *message=”Hi there 123”;
    int digits=0, letters=0, other=0, i;
    char c;

    for(i=0; i<strlen(message); i++)
    {
        c = *(message+i);

        if (isalpha(c)) letters++;
        else if (isdigit(c)) digits++;
        else other++;
    }
}
```

interesting

Arrays

Syntax:

- TYPE NAME [SIZE];
- TYPE NAME [COLS][ROWS];
- TYPE NAME [COLS][ROWS][LAYERS];
- TYPE NAME [COLS][ROWS][LAYERS][CUBES];
- Etc.

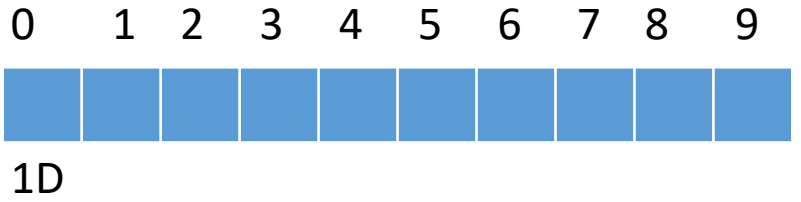
Multidimensional arrays are easy to create an manipulate in C.

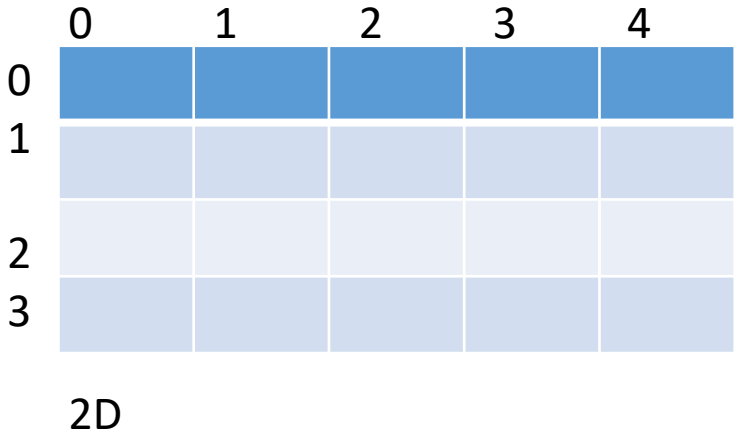
C arrays are variables, which means we can write to arrays and read from arrays.

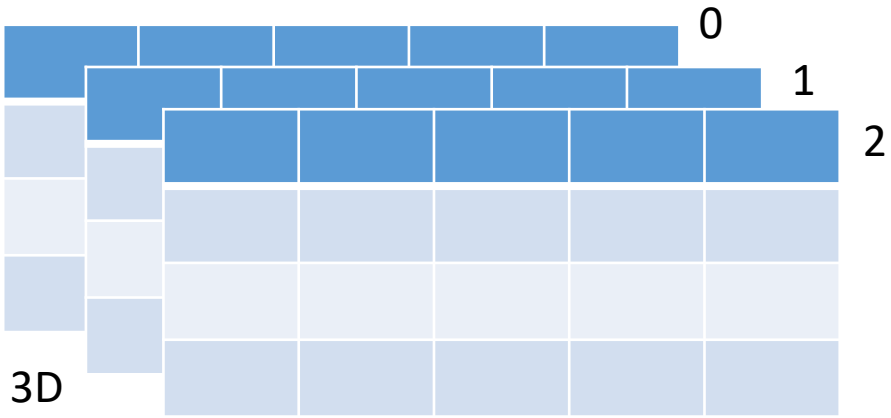
Arrays

Syntax:

- TYPE NAME [SIZE];
  - int data[100];
  - char name[30];
- TYPE NAME [COLS][ROWS];
  - int picture[100][200];
- TYPE NAME [COLS][ROWS][LAYERS];
  - char world[100][100][50];







Find a value

```
#include <stdio.h>
int main(void) {
    int numbers[5] = {5, 10, 15, 20, 25};
    int n, x;

    scanf("%d", &n);

    for(x=0; x<5; x++) {
      if (numbers[x] == n) { puts("Found\n"); break; }
    }

    if (x == 5) puts("Not found\n");
    return 0;
}
```

Multiply

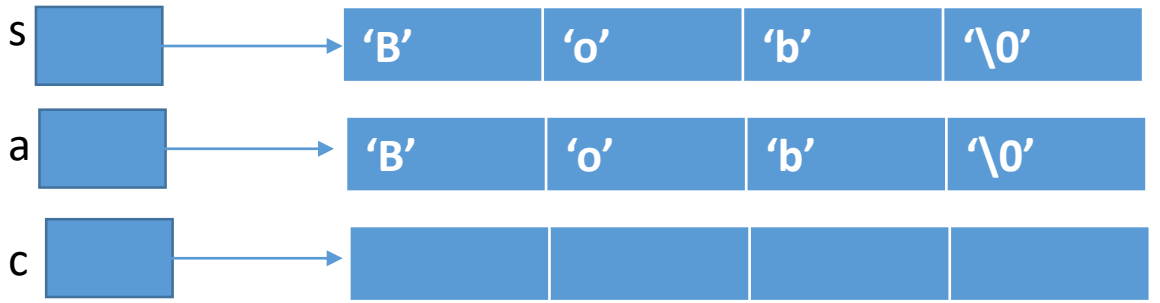
```
#include <stdio.h>
int main(void) {
    int vector[5] = {5, 10, 15, 20, 25}, result[5];
    int matrix[5][2] = { {1,2,3,4,5}, {6,7,8,9,0} };
    int a, b, multsum;

    for (a=0; a<5; a++) {
      multsum = 0;
      for (b=0; b<2; b++) {
            multsum += (vector[a] * matrix[a][b]);
      }
      result[a] = multsum;
    }
    return 0;
}
```

String vs Array

```
char *s = “Bob”;
char a[] = {‘B’, ‘o’, ‘b’, ‘\0’};
char c[4];
scanf(“%s”, c);
```

What do these look like physically in memory?

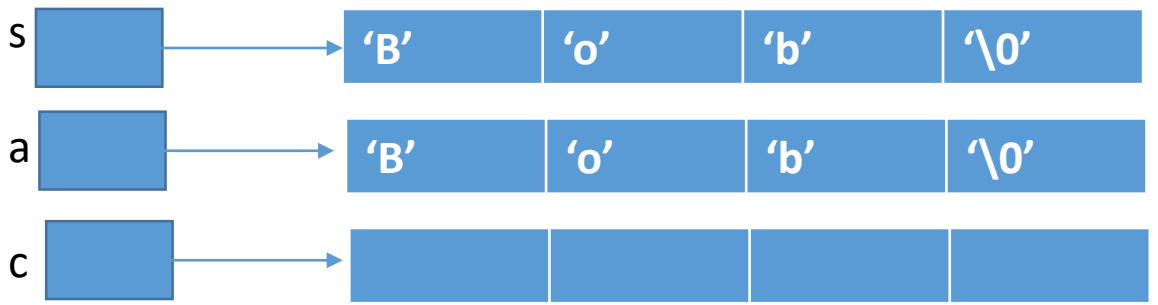


If the user entered at the scanf Bob then the last structure would look like the previous two.

Important

```
char *s = “Bob”;
char a[] = {‘B’, ‘o’, ‘b’, ‘\0’};
char c[4];
scanf(“%s”, c);
```

What do these look like physically in memory?



Static.
It is a string.

Variable.
They can be assigned to another structure of similar shape!!

Variable.
Each cell of the array is a variable and can be assigned a different value.

Example

Caesar cipher.

An ancient cryptographic technique.

Algorithm:

1. Read in a sentence from user (message)
2. Read in an integer number from user (key)
3. Character += key, for every char in message