OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

FAKULTÄT FÜR INFORMATIK

Communication and Networked Systems

C Crashkurs

TI 2 im SS 2017

Prof. Dr. Mesut Güneş

Literatur:

- B. Kernighan, D. Ritchie: *The C-programming language*, Second Edition, Prentice-Hall, 1988.
- R. Sedgewick: *Algorithms in C*, Third edition, Addison-Wesley, 1998.
- Unix man-Pages
- Internet: www.cppreference.com, http://www.cplusplus.com und http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/

Dank an Ralf Wienzek, RWTH Aachen, 1. Version WS 2006/2007

Inhaltsübersicht

- Historie
- Hauptunterschiede zu Java
- Datentypen
- Operatoren
- Kontrollstrukturen
- Präprozessor
- Debugging

Historie

Weiterentwicklung von BCPL und B

- BCPL: Ende der 60er Jahre von Martin Richards zum Bau von Betriebssystemen und Compilern entwickelt
- B: Ken Thompson erstellte 1970 mit B das erste UNIX System

C

- 1972 von Dennis Ritchie in den Bell Laboratories entwickelt
- Wurde zur Entwicklung des UNIX-Betriebssystems verwendet
- Zunächst durch den Klassiker „The C Programming Language“ von Brian Kernighan und Dennis Ritchie beschrieben und 1989 vom amerikanischen ANSI-Institut standardisiert
- Häufig nicht als Hochsprache angesehen, da maschinennahe Programmierung möglich
- Hohe Flexibilität, kleiner Sprachumfang (ANSI-C hat nur 32 Schlüsselwörter)

Hauptunterschiede zu Java

Merkmale	C	Java
Programmier-Paradigma	funktions-orientiert / prozedural	objekt-orientiert
Basis-Programmiereinheit	Funktion	Klasse
Kompilierung vs. Interpretation	Kompilierung, generiert Maschinencode `gcc hello.c`	Interpretation, generiert JVM bytecode `javac Hello.java`
Abstraktionsebene	Low-level language	High-level language
Speicheradressierung	Pointer (direkte Handhabung)	Referenzen (keine Manipulation der Pointer erlaubt)
Speicherverwaltung	durch Programmierer	automatisch (garbage collection)
Einbinden von Bibliotheken	Präprozessor `#include <stdio.h>`	Imports `import java.io.File`
Strings	‚\0' terminiertes Character Array	String Datentyp
I/O Funktionen	`printf()` `scanf()`	`System.out.print` `System.in.read`
Fehlerbehandlung	durch Programmierer (Debugging mit printf)	Exception handling

Sprachumfang

- C besteht aus der Sprache C und der C-Bibliothek
- Die Sprache ist rein prozedual
- Der Funktionsumfang der C-Bibliothek umfasst nur wenige, essentielle Funktionen
- Alle anderen Funktionen kommen von Dritten
  z.B. libpthread, libxml2, libcurl
- Ausnahme: GNU C Library (glibc) umfasst auch alle POSIX Betriebssystemfunktionen (z.B. sockets, fork, pipe)

Ein einfaches Programm

```
#include <stdio.h>            // Funktionsdeklarationen zur Ein-/Ausgabe einbinden

int maxi( int a, int b );     // Funktionsdeklaration

int global_max = 0;           // Globale Variable global_max (mit 0 initialisiert)

/*****************************
 * Hauptprogramm
 *****************************/
int main( int argc, char *argv[] ) {    // Hier beginnt die Programmausführung
    printf( "Hello World.\n" );    // Ausgabe von "Hello World." auf der Standardausgabe
    global_max = maxi( 1,2 );      // Aufruf der Funktion maxi
    return 0;                      // Rückgabewert des Programms
}

/*********************************
 * Funktion zur Maximumberechnung
 *********************************/
int maxi( int a, int b ) {         // Funktionsdefinition
    if (a>b)
        return a;
    else
        return b;
}
```

Datentypen

- **Elementare Typen**

Typ	*übliche* Größe	*minimaler* Wertebereich	Beispiele
char	8 Bit	-127 ... 127	'a', '&', ' ', '\n','\0'
unsigned char		0 ... 255	
_Bool [1]	8 Bit	wahr und falsch	0,1[1]
int	16 Bit	-32767 ... 32767	Dezimal: 0,-7,42 Oktal: -07,010 Hexadez.: 0x2f, 0xa
unsigned int		0 ... 65535	
(unsigned) short int	16 Bit	wie int	
long int	32 Bit	-2147483647 ... 2147483647	
unsigned long int		0 ... 4294967295	
long long int	64 Bit	-9223372036854775807 ... 9223372036854775807	
unsigned long long int		0 ... 18446744073709551615	
float	32 Bit	auf 6 Stellen genau	0.0, -1.7, 31415e-4
double	64 Bit	auf 10 Stellen genau	
long double	128 Bit	auf 10 Stellen genau	

[1] Mit `#include <stdbool>` auch *`bool`* mit den Konstanten *`true`* und *`false`*

Datentypen – Verbunde, Arrays I-III

- **Verbunde**

```
struct [Verbundname] {
   Datentyp VariablenName;
   Datentyp VariablenName;
   …
} [Variablenliste];
```

- **Beispiel**

```
struct punkt {
   int x;
   int y;
};
```

- **Variablendefinition**

```
struct punkt p;

struct {
   struct punkt oLinks, uRechts;
} r1, r2;
```

- **Zugriff**

```
p.x = 7;
r1.obenLinks.y = 42;
```

Datentypen – Verbunde, Arrays II-III

- **Arrays**

```
Datentyp Varname[Anz. der Elemente];
```

  - Arrays in C sind immer 0-basiert:

    `int a[n]` definiert die Elemente `a[0], …, a[n-1].`

  - Die Elemente eines Arrays befinden sich in einem zusammenhängenden Speicherbereich, d.h. sie befinden sich an aufeinanderfolgenden Adressen.

- **Definition**

```
char c[26];
float noten[4]={1.0, 2.0, 3.0, 4.0};
int tabelle [2][5] = {{1,2,3,4,5},
                      {9,8,7,6,5}};
```

- **Zugriff auf Elemente**

```
c[0] = 'a';
c[25] = 'z';
tabelle[1][3] = 7;
```

Datentypen – Verbunde, Arrays III-III

- **Union**

```
union [Unionsname] {
   Datentyp VariablenName;
   Datentyp VariablenName;
   …
} [Variablenliste];
```

- **Beispiel**

```
union inhalt {
   int wert;
   char *zeiger;
};
```

- **Variablendefinition**

```
union inhalt c;

struct nachricht {
   int empfaenger_id;
   union {
      int wert;
      char *zeiger;
   };
}
```

- **Zugriff**

```
c.wert = 7;
c.zeiger = „foo“;
```

Datentypen – enum, typedef I-II

- **enum**

  Aufzählung von Konstanten

  Integer-Variablen, die nicht jeden beliebigen Wert annehmen dürfen, sondern auf eine begrenzte Anzahl von Werten beschränkt sind. Diese Werte werden über Namen angesprochen.

```
enum [Name] {Element_1, …, Element_N} [Variablenliste];
```

- **Beispiel**

```
enum Wochentag {So, Mo, Di, Mi, Do, Fr, Sa};
enum Richtung { NORD, OST=90, SUED=180, WEST=270};

enum Wochentag heute;
heute = Di;
```

Datentypen – enum, typedef II-II

- **typedef**

  Neuer Bezeichner für bereits bestehenden einfachen Datentypen

```
typedef Datentyp NeuerName;
```

- **Beispiel**

```
typedef enum Wochentag wtag;
wtag morgen;
morgen = Mi;

typedef struct {
    char name[80];
    int matrikelnummer;
} tStudent;
tStudent ralf;
ralf.matrikelnummer = 123456;
```

- **Hinweis**

  Der Header <stdint.h> stellt architektur-abhängige typedefs bereit, z.B.:

```
typedef unsigned int uint32_t;
```

Datentypen – Zeiger

- **Ein Zeiger ist eine Variable, die eine Speicheradresse beinhaltet. Die Größe und das Format einer Adresse ist abhängig vom Betriebssystem.**

- **Definition**

  `Datentyp *Variablenname;`

- **Beispiele**

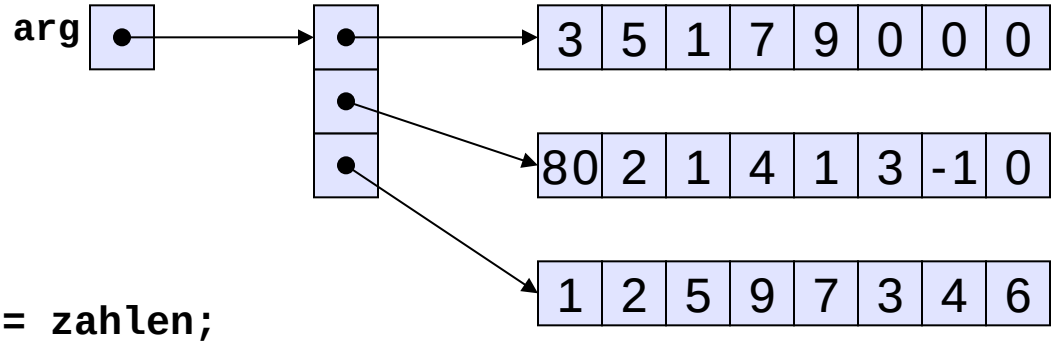
```
int i = 7;
int *p_i;                  // int-Zeiger
int *p_j=NULL;             // Zeiger mit NULL initialisiert
struct punkt *p_punkt;     // Zeiger auf eine Verbundvariable

p_i = &i;                  // & ist Adressoperator:
                           // p_i speichert Adresse von i
p_j = p_i;
*p_j = *p_j + 1;           // Dereferenzierung -> i hat nun den Wert 8!

(*p_punkt).x = 9;          // Beide Anweisungen
p_punkt->x = 9;            // bewirken das gleiche
```

Datentypen – Zeiger und Arrays

- **Eine Array-Variable ist nichts anderes als ein Zeiger auf das erste Element:**

```
int zahlen[8] = {1,2,5,9,7,3,4,6};
```


```
int arg[3][8];
```




```
int *p = zahlen;
*p == p[0];
*(p+1) == p[1];

int **q = arg;          // Zeiger auf einen Zeiger
p = q[1];
```

Operatoren

- **Arithmetische Operatoren**
  - `+` Addition
  - `-` Subtraktion, Negation
  - `*` Multiplikation
  - `/` Division
  - `%` Divisionsrest (Modulo)
- **Bit-Operatoren**
  - `&` Bitweises UND
  - `|` Bitweises ODER
  - `^` Bitweises XOR
  - `~` Invertieren
- **Shift-Operatoren**
  - `<<` Links-Shift
  - `>>` Rechts-Shift
- **Zeiger-Operatoren**
  - `&` Adressoperator
  - `*` Dereferenzierung
- **Vergleichsoperatoren**
  - `==` Gleich
  - `!=` Ungleich
  - `<` Kleiner
  - `<=` Kleiner oder gleich
  - `>` Größer
  - `>=` Größer oder gleich
- **Logische Verknüpfungen**
  - `&&` Logisches UND
  - `||` Logisches ODER
  - `!` Negation

Typkonvertierungen

- **Implizite Konvertierung**
  - Enthält ein Ausdruck Variablen oder Konstanten verschiedener Datentypen, wird der Typ einzelner Operanden automatisch umgewandelt. Dabei wird der Operand mit kleinerem Wertebereich an den größeren Datentyp angepasst.

- **Explizite Konvertierung**
  - Der Programmierer gibt durch Voranstellen des Datentyps an, in welchen Typ die nachfolgende Variable oder Konstante umgewandelt werden soll.

- **Beispiele**
  - **float f = 1.0/3**
    ergibt 0.333333: erster Operand ist Gleitkommazahl, d.h. implizite Konvertierung von 3 zu 3.0 und Durchführung einer Fließkommadivision
  - **float f = 1/3**
    ergibt 0.0: ganzzahlige Division von 1 durch 3 ergibt 0, implizite Konvertierung von 0 zu 0.0
  - **float f = (int)0.333333 * 3**
    ergibt 0.0: explizite Konvertierung von 0.333333 zu 0, anschließende Multiplikation mit 3 und impliziter Konvertierung zu 0.0
  - **float f = (float)1/3**
    ergibt 0.333333: explizite Konvertierung von 1 zu 1.0, implizite Konvertierung von 3 zu 3.0 und Durchführung einer Fließkommadivision
  - **float f = (float)(1/3)**
    ergibt 0.0: ganzzahlige Division von 1 durch 3 ergibt 0, explizite Konvertierung von 0 zu 0.0

Kontrollstrukturen

- **if - Anweisungen**

```
if (Bedingung)
    Anweisung;
[else
    Anweisung;]
```

- **Beispiele**

```
if (a > b)
  max = a;
else
  max = b;
```

```
if (arg[0]=='-')
  if  (arg[1]=='v') {
    version=1;
    printf("Version 1\n“);
  } else
    printf("Unknown option.\n“);
```

- **Verschachtelte if-Anweisungen**

```
if (Bedingung1)
  Anweisung1;
else if (Bedingung2)
  Anweisung2;
else if (Bedingung3)
  Anweisung3;
else
  Anweisung4;
```

Bedingungen 1-3 werden in dieser Reihenfolge geprüft. Sobald eine Bedingung erfüllt ist, wird die entsprechende Anweisung ausgeführt. Ist keine Bedingung erfüllt, wird Anweisung4 ausgeführt.

Kontrollstrukturen

- **switch - Anweisung**
  Auswahl unter mehreren Alternativen.

```
switch (Ausdruck) {
   case konstanter Ausdruck1:
      Anweisung1a;
      Anweisung1b;
      ...
   case konstanterAusdruck2:
      Anweisung2a;
      ...
   ...
   case konstanterAusdruckN:
      AnweisungNa;
      ...
   [default:
      AnweisungDa;
      AnweisungDb;
      ...]
}
```

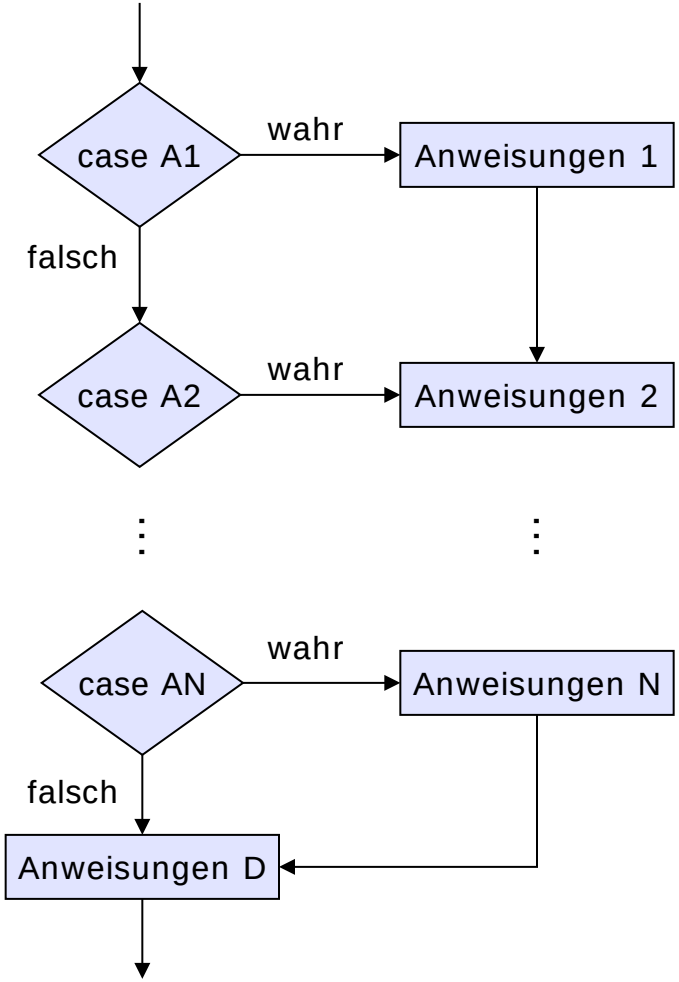
- **Es wird zunächst der Ausdruck nach switch ausgewertet und das Ergebnis mit allen case-Konstanten verglichen. Stimmt der Wert mit einer Konstanten überein, wird die Programmausführung dort bis zur nächsten break-Anweisung fortgeführt. Ist keine Übereinstimmung zu finden, wird zur (optionalen) default-Marke gesprungen. Ist diese nicht vorhanden, werden keine Anweisungen ausgeführt.**

- **Beispiel**

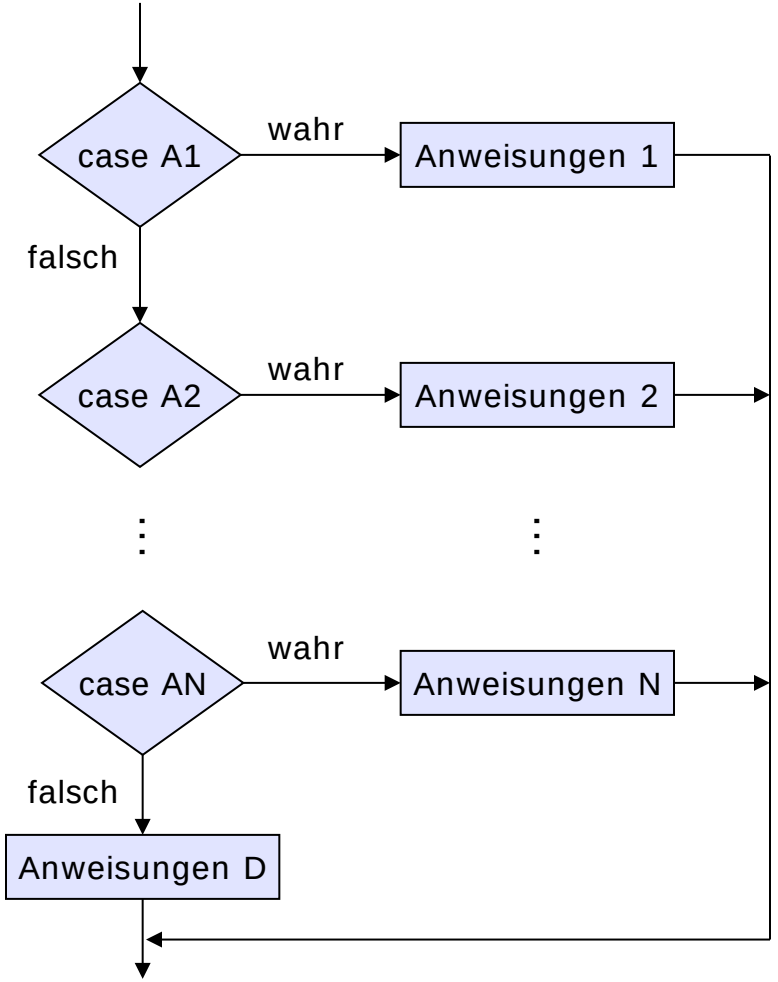
```
char buchstabe;
...
switch ( buchstabe ) {
   case 'a': printf("a\n“);
   case 'b': printf("b\n“); break;
   case 'c': printf("c\n“); break;
   default: printf("Nicht a,b,c\n“);
}
```

Kontrollstrukturen

- **switch ohne break-Anweisungen**



- **switch mit break-Anweisung**



Kontrollstrukturen

- **while-Schleife**

```
while (Bedingung)
    Anweisung;
```

Solange die Auswertung der Bedingung einen Wert ungleich 0 liefert, wird der Anweisungsblock durchlaufen.

- **do-while-Schleife**

```
do
    Anweisung;
while (Bedingung);
```

Die Anweisung wird immer mindestens ein mal ausgeführt. Liefert beim Erreichen des while-Statements die Auswertung der Bedingung einen Wert ungleich 0, wird die Anweisung erneut ausgeführt.

- **for-Schleife**

```
for (AnwInit; Bedingung;
AnwSchleife)
    Anweisung;
```

Beim Erreichen des for-Statements wird zunächst AnwInit ausgeführt. Ist danach die Schleifenbedingung erfüllt, wird der Schleifenkörper durchlaufen. Nach jedem Schleifendurchlauf wird AnwSchleife ausgeführt und falls die Schleifenbedingung noch erfüllt ist, ein weiterer Durchlauf gestartet.

- **Beispiel**

```
int i, sum=0;

for (i=0; i<=10; i=i+1)
    sum = sum+i;

for (i=10; i; i=i-1)
    sum = sum+i;

for (;;)
    puts(„Endlos-Schleife“);
```

Kontrollstrukturen

break

- Anwendbar bei **switch**, **for**-, **while**-und **do-while**-Scheifen
- Verhindert die Ausführung weiterer Anweisungen innerhalb der Schleife
- Führt zum sofortigen Verlassen von for-, while und while-do-Schleifen
- Kann die Lesbarkeit von Programmen erhöhen, da nicht sämtliche Bedingungen in den Schleifenkopf untergebracht werden müssen

- **Beispiel**

```
while (1) {      // Endlosschleife
   ...
   if ( input==0 )
       break;    // Beenden der Schleife
   ...
}
```

continue

- Anwendbar bei **for**-, **while**-, **do-while**-Schleifen
- Bewirkt eine sofortige Rückkehr zur Schleifenanweisung

- **Beispiel**

```
for (i=0; i<7; i++) {
   if ((i==2) || (i==4))
       continue;
   printf( "%i " );
}
```

```
Ergibt:
0 1 3 5 6
```

Präprozessor Direktiven

Präprozessor : Teil des Compilers

Aufgabe: Vorbereitung von Eingabedaten bevor Programm vom Compiler übersetzt wird (Quelltextersetzung)

Beispiele:

- Ersetzen von Kommentaren durch Leerzeichen
- Kopieren von Header-/Quelldateien in Quelltext (#include)
- Einbindung symbolischer Konstanten (#define)
- Bedingtes Übersetzen von Zeilen (z.B. #ifdef)

#include (Einkopieren von Dateien)

```
#include <header>
#include „header“ // im akt. Arbeitsverzeichnis
```

- PP entfernt include-Zeile, ersetzt sie durch Quelltext in header-Datei

Bsp: `#include <stdio.h>`

#define (Makros und Konstanten)

```
#define Bezeicher Ersatzbezeichner
```

- Austausch von Zeichenketten durch Bezeichner

Beispiel Konstante:

```
#include <stdio.h>
#include <stdlib.h>
#define ZAHL 42
int main(void)
{
     printf(„%d\n“,ZAHL);
     return EXIT_SUCCESS;
}
```

Beispiel Makro:

```
#define KLEINER_100(x) ((x) < 100)
```

Debugging mittels printf()

printf(Kontrollstring, arg1, arg2, ...)

- Flexible Ausgabefunktion
- Die Funktion wertet den Kontrollstring aus und formatiert die Argumente entsprechend
- Einfacher Text im Kontrollstring wird unverändert ausgegeben
- Platzhalter im Kontrollstring werden durch die Werte der weiteren Argumente ersetzt:
  - %**d** Dezimalzahl
  - %**x** Hexadezimalzahl
  - %**u** Vorzeichenlose Zahl
  - %**c** ein Zeichen
  - %**s** Zeichenkette
  - %**f** Gleitkommazahl

- **Beispiel**

```
#include <stdio.h>
int main()
{
   int i=42;
   printf("%d als Hex.-Zahl: %x\n“, i, i);
   char *text = "Hallo";
   printf("%s\n„, text);
   return 0;
}
```

```
$ gcc –o printf_bsp printf_bsp.c
$ ./printf_bsp
$ 42 als Hex.-Zahl: 2a
$ Hallo
```

Beispiele

```
#include <stdio.h>
typedef char* STRING;

void func1( int var1, STRING var2 );

int main (int argc, char *argv[] )
{
  int i;
  printf( "Dateiname: %s\n", argv[0] );
  for (i=1; i<argc; i++)
     func1( i, argv[i] );
  return 0;
}

void func1( int var1, STRING var2 )
{
  printf( "%i. Parameter: %s\n", var1, var2 );
}
```

```
$ gcc –o bsp bsp.c
$ bsp
Dateiname: bsp
$ bsp -23 zwei
Dateiname: bsp
1. Parameter: -23
2. Parameter: zwei
$ _
```

Die C-Bibliothek

<assert.h> C Diagnostics Library
<ctype.h> Character handling functions
<errno.h> Errors
<float.h> Characteristics of floating-point types
<iso646.h> ISO 646 Alternative operator spellings
<limits.h> Sizes of integral types
<locale.h> C localization library
<math.h> C numerics library
<setjmp.h> Non local jumps
<signal.h> C library to handle signals
<stdarg.h> Variable arguments handling
<stdbool.h> Boolean type
<stddef.h> C Standard definitions
<stdint.h> Integer types
<stdio.h> C library to perform Input/Output operations
<stdlib.h> C Standard General Utilities Library
<string.h> C Strings
<time.h> C Time Library
<uchar.h> Unicode characters
<wchar.h> Wide characters
<wctype.h> Wide character type

[http://www.cplusplus.com/reference/clibrary/]