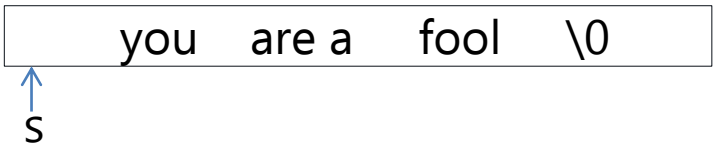
Strings

- declaration
  - char *s = “abcde”;
  - char s[] = “abcde”;
  - char s[] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘\0’};




```
#include <ctype.h>

int word_cnt(char *s)
{
   int       cnt = 0;
   while (*s != '\0') {
             while (isspace(*s))  ++s;
             if (*s != '\0') {
                ++cnt;
                while (!isspace(*s) && *s != '\0')  ++s;
             }
   }
   return cnt;
}
```

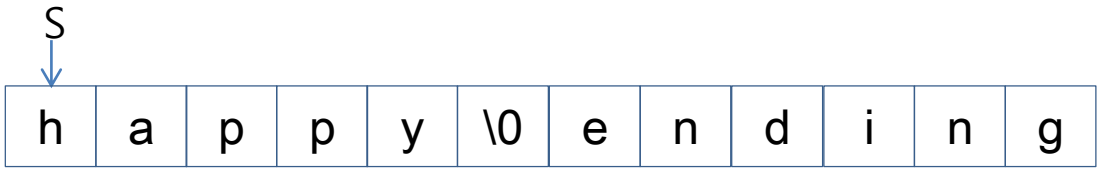
String Functions

- ANSI C Lib contains many useful functions
  - char *strcat(char *s1, const char *s2);
    - result is in *s1
    - what if there is no space after s1?
    - strncat(dest, src, n);
    - char * const a; /* can change char but NOT pointer */
    - return value == s1 /* why do we need it */
  - int strcmp(const char *s1, const char *s2);
    - returns negative, zero, positive depending on the lexicographical order
    - strncmp(s1, s2, n)

- char *strcpy(char *s1, const char *s2);
  - copy s2 to s1
  - what if s2 is longer than s1?
  - strncpy(dest, src, strlen(dest)); /* safe */

- size_t strlen(const char *s);
  - size_t is usually unsigned int

- char *strchr(char *s, int c) /* find c in s */
- char *strtok(char *s, char *delimiters) /* tokenize s */
  - strtok("- This, a sample string.", " ,.-") will generates
  - This a sample string
- char *strstr(char *s, char *pat) /* find pat in s */

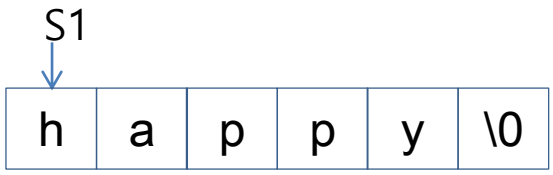


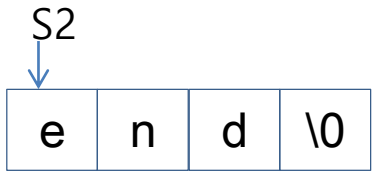

```
unsigned strlen(const char *s)
{
    register int    n;

    for (n = 0; *s != '\0'; ++s)
        ++n;
    return n;
}
```

```
char   *strcat(char *s1, const char *s2)
{
    register char   *p = s1;

    while (*p)
        ++p;
    while (*p++ = *s2++)
        ;
    return s1;
}
```





Declarations and initializations

```
char    s1[] = "beautiful big sky country",
        s2[] = "how now brown cow";
```

Expression	Value
`strlen(s1)`	25
`strlen(s2+8)`	9
`strcmp(s1, s2)`	negative integer
Statements	**What gets printed**
`printf("%s", s1 + 10)`	big sky country
`strcpy(s1 + 10, s2 + 8)`	
`strcat(s1, "s!")`	
`printf("%s", s1)`	beautiful brown cows!

Arrays of Pointers

- char *w[N];
  - an array of pointers
  - each pointer is to char (string!)

- ragged array
  - char *p[2] = {“abc”, “1234567890”};

Arguments to main( )

- int **main**(int argc, char **argv)

- argc and argv are used for main()
  - argc is the number of arguments
  - argv is an array of pointers
    - argv[0] is the name of the main program
    - then naturally, argc >= 1

my_echo.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int   i;

    printf("argc = %d\n", argc);
    for (i = 0; i < argc; ++i)
       printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

```
$ my_echo midterm is on Thursday
```

argc = 5

argv[0] = my_echo

argv[1] = midterm

argv[2] = is

argv[3] = on

argv[4] = Thursday

Functions as Arguments

- a function name can be passed as an argument
- think a function name as a pointer (like an array)
- (*f)(x)
  - f is a pointer to a function
  - *f is a function
  - (*f)(x) is a call to the function
- if you are still confused, just follow the example

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

double    f(double);
double    sum_square(double (*)(double), int, int);
```

```c
#include "sum_sqr.h"

int main(void)
{
   printf("%s%.7f\n%s%.7f\n",
      " First computation: ", sum_square(f, 1, 10000),
      "Second computation: ", sum_square(sin, 2, 13));
   return 0;
}
```

```c
double sum_square(double f(double), int m, int n)
{
   int      k;
   double   sum = 0.0;

   for (k = m; k <= n; ++k)
      sum += f(k) * f(k);
   return sum;
}
```

```c
double f(double x)
{
   return 1.0 / x;
}
```

Functions as Arguments

- prototypes

```
double sum_square(double f(double x), int m, int n);
double sum_square(double f(double), int m, int n);
double sum_square(double f(double), int, int);
double sum_square(double (*f)(double), int, int);
double sum_square(double (*)(double), int, int);
```

const volatile

- const int N = 3;
  - N cannot be changed after initialization
  - N cannot be used for array definition like
    - int k[N];

- with pointers

```
int *const ptr;        /* ptr is const*/
const int* ptr;        /* integer is const */
int const *ptr;        /* integer is const*/
```

volatile int *p;

telling a compiler NOT to change order, cacheing or anything related to the integer since it may be changed by an external event

```
void SendCommand (int * hwstatus, int command, int data)
 {
    // wait while the gadget is busy:
    while (hwstatus == isbusy)
    {
       // do nothing here.
    }
    // set data first:
    hwdata = data;
    // writing the command starts the action:
    hwcommand = command;
}
```

Storage Classes

- Every variable and functions in C has two arrributes: type and storage class

- Storage Classes

  auto extern register static

auto

- the most common class
  - variables defined inside a function
  - variables defined outside a function are global
- default class – you may omit it

- the memory space is allocated/released when the function is invoked/exited

- when a function is reentered, the previous values are unknown

external

- global
- they may be defined somewhere else (in another file)
- they never disappear
  - transmit values across functions
- they may be hidden by re-declaration, but they are not destroyed

```c
#include <stdio.h>

int   a = 1, b = 2, c = 3;
int   f(void);

int main(void)
{
   printf("%3d\n", f());
   printf("%3d%3d%3d\n", a, b, c);
   return 0;
}

int f(void)
{
   int   b, c;

   a = b = c = 4;
   return (a + b + c);
}
```

main.c

```
#include <stdio.h>

int   a = 1, b = 2, c = 3;      /* external variables */
int   f(void);

int main(void)
{
   printf("%3d\n", f( ) );
   printf("%3d%3d%3d\n", a, b, c);
   return 0;
}
```

fct.c

```
int f(void)
{
   extern int   a;              /* look for it elsewhere */
   int          b, c;

   a = b = c = 4;
   return (a + b + c);
}
```

```
CC      =  gcc
CFLAGS  =  -Wall
EXEC    =  a.out
INCLS   =
LIBS    =

OBJS    =  main.o  fct.o

$(EXEC): $(OBJS)
        @echo "linking ..."
        @$(CC)  $(CFLAGS)  -o $(EXEC)  $(OBJS)  $(LIBS)

$(OBJS):
        $(CC)  $(CFLAGS)  $(INCLS)  -c  $*.c

relink:
        @echo "relinking ..."
        @$(CC)  $(CFLAGS)  -o $(EXEC)  $(OBJS)  $(LIBS)
```

* read Makefile tutorial

register

- allocate this variable on a register
- to speed up the execution
- not always possible to find a register
- tricky for memory-IO operations

static

- to preserve the value even after the function exits
  - extern does the same
- to control visibility of variable and functions
  - "static extern" - visible only within the same source file

```
void f(void)
{
   static int cnt = 0;

   ++cnt;
   if (cnt %2 == 0)
            .....
   else
            ....
}
```

```
static int seed = 100;
/* static extern – external, but invisible from other files */

int random(void)
{
   seed = 25173 * seed + 13849;
            ....
}
```

```
/* function g( ) can be seen only within this file */
static int g(void)
{

        ....
}



void f(int a)
{
        .....
        ....
}
```