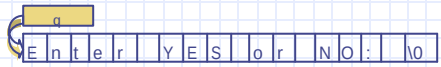
Strings:

- A string is an array of characters, terminated with a trailing null character, '\0'.

```
char *s = "CIS 2520";
char *q;
    q = "Enter YES or NO: ";
```



String Variables:

- A string variable is created by allocating an array of characters:

```
char s[5] = {'s', 't', 'o', 'p', '\0'};
char t[3];
    t[0] = 'N';
    t[1] = 'O';
    t[2] = '\0';

// must explicitly provide the space
// for trailing null.
```


Problem Solving: Counting Words:

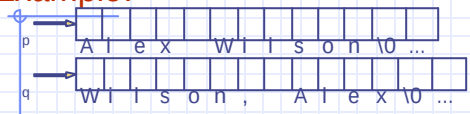
```
#include <ctype.h>

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;
}
```


Standard String Functions:

- Commonly used standard string functions: string.h

```
char *strcat(s1, s2)  //cancatenate s2 to s1, return s1
char *strcpy(s1, s2) // copy s2 to s1, return s1
 int strcmp(s1, s2)   // compares s1 and s2 (-, 0, +)
size_t strlen(s)      // return num of chars (not '\0')
char* strchr(s, c)    // return a pointer to the first c in s
char* strstr(s1, s2) // return a ptr to the first s2 in s1
char* strspn(s1, s2) // return length of prefix of s1
                      // consisting of characters in s2
                      // and more ...
```


Example:




```
char* bp;
  if ((bp = strchr(p, ' ')) == NULL) error(...); // print error msg
  strcpy(q, bp + 1);    // make the last name first
  strcat(q, ", ");      // append a comma
  *bp = '\0';           // terminate p at the end of first name
  strcat(q, p);         // put first name last
```


Additional String Functions:

- stdio library provides two functions, extensions to printf and scanf, allow I/O from/to a string: sprintf and sscanf.

```
char* bp;
  if ((bp = strchr(p, ' ')) == NULL) error(...); // print error msg
  strcpy(q, bp + 1);    // make the last name first
  strcat(q, ", ");      // append a comma
  *bp = '\0';           // terminate p at the end of first name
  strcat(q, p);
  // more compact with sprintf
  *bp = '\0';
  sprintf(q, "%s, %s", bp + 1, p);
```

Example: strcpy

```
// version 1:
char* strcpy(char* s1, char* s2){
    int i;
    for (i = 0; (s1[i] = s2[i]) != '\0'; i++);
    return s1;
}
// version 2:
char* strcpy(char* s1, char* s2){
    char* tp = s1;
    while(*tp++ = *s2++);
    return s1;
}
```


Pointer version *vs.* Array version:

```
#include <stdio.h>
int getline1(char line[]; int max){ // array version
    int c, i = 0;
    while ((c = getchar()) != '\n' && c != EOF)
        if (i < max) line[i++] = c;
    line[i] = '\0';
    return (c==EOF) ? –1 : i;
}
Int getline2(char *p, int max){ // pointer version
    int c;
    char *sp = p, *ep = p + max;
    while ((c = getchar()) != '\n' && c != EOF)
        if (p < ep) *p++ = c;
    *p = '\0';
    return (c==EOF) ? –1 : p – sp;
}
```


Prototypes for useful string functions:

- Standard library provides a set of useful string functions, but they tend to be difficult to use and hard to remember.
- You might redefine these utilities by your own package on top of the standard library.
- These prototypes might be less general but significantly clearer and easier to understand than code that directly uses library functions.

Example:

```
// A package header file (sutils.h)
#include <string.h>
char *firstNonBlank(char* s);
char *lastNonBlank(char* s);
void stringLower(char *s);
void stripBlanks(char* s);
int emptyString(char* s);
int stringEqual(char* s1, char*s2);
```



```
// An implementation (sutils.c)
#include "sutils.h"
static const char *Spaces = " \t"; // a string of two chars: blank and tab

int stringEqual(char* s1, char* s2){ return strcmp(s1, s2) == 0;}

int emptyString(char* s){ return *s == '\0'; }

char* firstNonBlank(char* s){ return s + strspn(s, Spaces);}

void stringLower(char* s){ // lowercase string
    for ( ; *s; s++) *s = tolower(*s);
}
```



```
char* lastNonBlank(char* s){ // return ptr to just past last non-blank
    int len = strlen(s), i = len;
    while (--i > 0)
        if (s[i] !=' ' && s[i] != '\t') return &s[i+1];
    return s + len;
}

void stripBlanks(char* s){ // remove leading/trailing blanks
    *(lastNonblank(s)) = '\0';
    strcpy(s, firstNonBlank(s));
}
```

Arrays of Pointers:

- 2-D arrays contain the same number of elements in each row. For example:

```
char days[][10] = {
    {'m','o','n','d','a','y','\0'},
    {'t','u','e','s','d','a','y','\0'},
    ...
};
```

Space is wasted in the rows containing shorter strings. Can we build an array whose rows can vary in length?

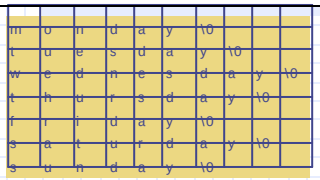
Ragged Array:

- A ragged array is an array of pointers where each entry in the array is a pointer to a string (arbitrary length). For example:

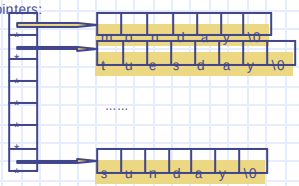
```
char *days[] = { "monday", "tuesday", "wednesday",
"thursday", "friday", "saturday", "sunday" };
```

The compiler allocates an array containing 7 elements and assigns each element a pointer to the corresponding string.

2-D Array:



Array of pointers:



Accessing Elements:

- Elements in a ragged array can be access in either pointer-based access or array-based access. For example,

```
char *days[] = { "monday", "tuesday", "wednesday",
"thursday", "friday", "saturday", "sunday" };
```

Suppose we want to access the third char of "wednesday", the following are equivalent:

```
days[2][2]
*(days[2] + 2)
*(*(days + 2) + 2)
```

`days` is the base-address of the array of pointers

`days[k]` is the pointer to the kth string

Self-Test:

```
char *days[] = { "monday", "tuesday",
  "wednesday", "thursday", "friday",
  "saturday", "sunday" };
```

Expression:	Value:
*(days[2] + 4)	'e'
days[3][5]	'd'
days[4]	"friday"
*(days[5])	's'
days[6][6]	'\0'
**days	'm'

Command-line Arguments:

- When we run a program, we often provide some initial values the program will work with. For Example:

```
$myprogram filename 222
```

- Two arguments, `argc` and `argv`, can be used with the `main()` to communicate with the operating system, where `argc` is the number of arguments on the command-line, and `argv` is an array of pointers to strings containing the various arguments.
- For the above example:

```
argc == 3
argv[0] == "myprogram"
argv[1] == "filename"
argv[2] == "222"
```

Passing Arguments to main():

```
#include <stdio.h>
int main(int argc, char *argv[]){
    int k;
    printf("argc = %d\n", argc);
    for (k = 0; k < argc; k++)
        printf("argv[%d] = %s\n", k, argv[k]);
}
```

```
$ mytest –o try this
argc = 4
argv[0] = mytest
argv[1] = -o
argv[2] = try
argv[3] = this
```


Self-Test:

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

Expression:	Value:
strlen(s1)	25
strlen(s2 + 8)	9
strcmp(s1, s2)	negative int

Statement:	what is printed
printf("%s", s1 + 10);	big sky country
strcpy(s1 + 10, s2 + 8);	
strcat(s1, "s!");	
printf("%s", s1);	beautiful brown cows!

Self-Test:

```
char *p[2][3] = {{"abc", "defg", "hi"},
                 {""jklmno", "pqrstuvw", "xyz"}};
```

Expression:	Equivalent expression:	Value:
***p	p[0][0][0]	'a'
**p[1]	p[1][0][0]	'j'
**(p[1] + 2)	p[1][2][0]	'x'
((p + 1) + 1)[7]	?	error
(*(*(p + 1) + 1))[7]	p[1][1][7]	'w'
*(p[1][2] + 2)	p[1][2][2]	'z'

The precedence of [] is higher than *

Unions:

- A Union may contain one of many different types of values, but can store only one value at a time.
- A Union type is defined by a keyword union with an optional union tag and the alternative names and types it may hold.

```
// a union which may contain either a char, an integer or a double
union number {
        char c;
        int   n;
        double d;
};
union number x1, x2, x3;
```


Union variables:

- Union variables are declared the same as structure variables. We also use the dot operator to access a union's individual fields. Note: don't assign to one field of a union and then access another.

```
x1.c = 'A';
x2.n = 123;
x3.d = 99.87;
if (x1.d == 56.2) { // wrong, x1 holds a char now }
x1.d = 55.3; // change to another alternative
if (x1.d == 56.2) {// OK   }
```


Enumerated Types:

- Enumerated type is used to specify a small range of possible values.
- A enumerated type is defined by giving the keyword enum followed by an optional type designator and a brace-enclosed list of identifiers.

```
enum color {BLUE, RED, WHITE, BLACK};
 enum myType { STRING = 2, INTEGER = 0, REAL};
```

- The list of ids represent a list of constants equal to their position in the list; or user may assign special values to ids in the list. The default value for item is 1 more than the item preceding it.

A Union and Structure Example:

```
typedef  enum {AUTO, BOAT, PLANE, SHIP} vFlag;

typedef struct { /* structure for an automobile */
    int tires;
    int fenders;
    int doors;
} autoType;

typedef struct { /* structure for a boat or ship */
    int displacement;
    char length;
} boatType;
```



```
typedef struct {
    char engines;
    int wingspan;
} planeType;

typedef struct {
    vFlag vehicle_flag; /* what type of vehicle? */
    int weight; /* gross weight of vehicle */
    union { /* type-dependent data */
        autoType    car;        /* part 1 of the union */
        boatType    boat;       /* part 2 of the union */
        planeType   airplane;   /* part 3 of the union */
        boatType    ship;       /* part 4 of the union */
    } vehicle_type;
    int value; /* value of vehicle in dollars */
    char owner[32]; /* owners name */
} vehicle;
```



```
void print_auto(vehicle *  pv){
   printf("Auto weight = %d\n", pv->weight);
   printf("# of tires: %d\n, # of fenders: %d\n, # of doors: %d\n",
         pv->vehicle_type.car.tires,
         pv->vehicle_type.car.fenders,
         pv->vehicle_type.car.doors);
    printf("Cost = $%d\n", pv->value);
    printf("Owned by %s\n", pv->owner); /* owners name */
}

void print_plane(vehicle *  pv){ ...};
void print_boat(vehicle *  pv){ ...};
void print_ship(vehicle *  pv){ ...};
```



```
void print_vehicle(vehicle *  pv){
  switch (pv->vehicle_flag){
      case AUTO:     print_auto(pv);
                     break;
      case PLANE:    print_plane(pv);
                     break;
      case BOAT:     print_boat(pv);
                     break;
      case SHIP:     print_ship(pv);
                     break;
      default:       printf("Unknown vehicle type\n");
  }
}
```



```
void main() {
vehicle  v[2], *piper_cub;
    v[0].vehicle_flag = AUTO;
    v[0].weight = 2742;              /* with a full gas tank */
    v[0].vehicle_type.car.tires = 5;  /* including the spare */
    v[0].vehicle_type.car.doors = 2;
    /* ... other initialization of v[0] */
    v[1].vehicle_flag = BOAT;
    v[1].value = 3742;               /* trailer not included */
    v[1].vehicle_type.boat.length = 20;
    /* ... other initialization of v[1] */
    piper_cub = (vehicle*) malloc(sizeof(vehicle));
    piper_cub->vehicle_flag = PLANE;
    piper_cub->vehicle_type.airplane.wingspan = 27;
    /* ... other initialization of piper_cub */
    print_vehicle(v);
    print_vehicle(&v[1]);
    print_vehicle(piper_cub); /* piper_cub is a pointer to vehicle */
}
```