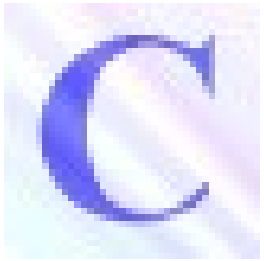
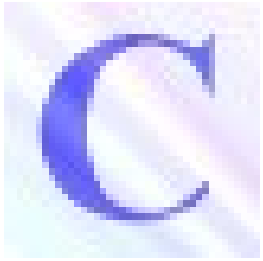


Computer Programming

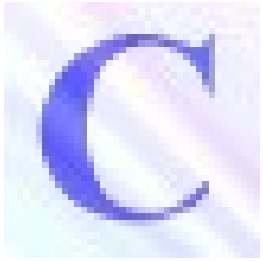
” Make everything as simple as possible, but not simpler.”

Albert Einstein



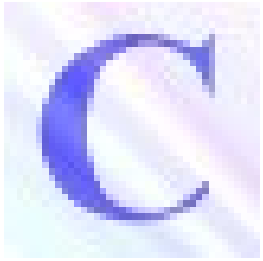
Outline

- Functions
  - Structure of a function
  - Function invocation
  - Parameter passing
  - Variable scope
  - Functions for character processing



Functions

- Functions
  - Modularize a program
  - All variables declared inside functions are local variables
  - Known only in function defined
- Parameters
  - Communicate information between functions
  - Local variables
- Benefits
  - Divide and conquer
  - Manageable program development
  - Software reusability
  - Use existing functions as building blocks for new programs
  - Abstraction - hide internal details (library functions)
  - Avoids code repetition

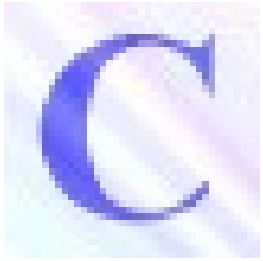


Structure of a function

- Structure:

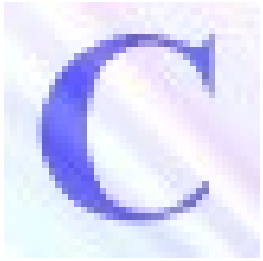
```
return_value_type name(formal_parameter_list)
{
   declarations
   statements
}
```

  - `name`: any valid identifier
  - `return-value-type`: data type of the result (default `int`)
  - First row is called a function *header*
  - Formal parameter list may contain:
    - No parameters, i.e. header is just `return_value_type identifier();` or `return_value_type identifier(void);`
    - One or more parameters, separated by commas. A formal parameter is specified by: `type identifier`



Structure of a function

- Two categories of functions:
  - Functions returning a value using "`return expression;`"
  - Functions not returning a value: use just "`return;`". `return_value_type` is replaced by `void`
- Returning control
  - If nothing returned
    - upon encounter of `return;`
    - or, until reaches right brace
  - If something returned
    - Upon encounter of `return expression;`



Function prototype

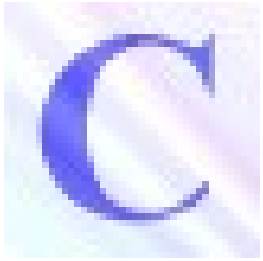
- Function prototype:
  - Function name
  - Parameters - what the function takes in
  - Return type - data type function returns (default **`int`**)
  - Used to validate functions
  - Obtained by copying the header and appending a semicolon
  - Formal parameter names (not types!) can be omitted
  - Prototype only needed if function definition comes after use in program

```
int maximum( int, int, int );
```

    - Takes in 3 **`int`**s
    - Returns an **`int`**

Function arguments

- *Parameters*: appear in definitions
- *Arguments*: appers in function calls
  - In C arguments are passed **by value**
- Argument conversion rules. The compiler has:
  - *encountered a prototype prior* the call: the value of each argument is implicitly converted to the type of the corresponding parameter as if by assignment
  - *NOTencountered a prototype prior* the call: default argument promotions, i.e. float=>double; char & short=>int



Array arguments

- If arg is one-dimensional array: size may be left unspecified

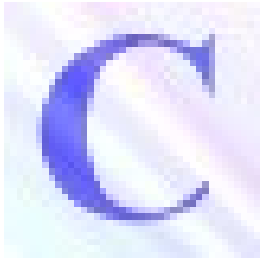
```
int f(int a[]){ // no length specified
   …
}
```

  - **Note**: cannot use `sizeof` to get the length of the array

- Variable length array arguments (C99):

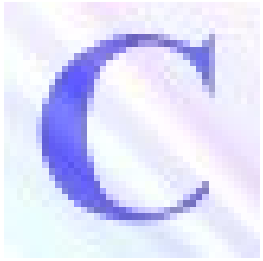
```
int f(int n, int a[n]){
   …
}
```

  - **Note**: length arg, `n,` must precede array



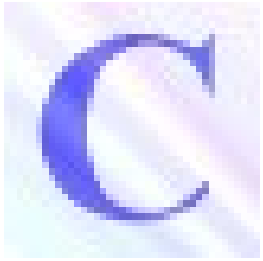
Header files

- Header files: contain prototypes for library functions
  - Standard library functions prototypes are found in header files (e.g. `stdio.h, stdlib.h, math.h, string.h`, etc.).
  - Load with `#include <filename.h>`
- Custom header files
  - Create a file with function prototypes (and macros if any)
  - Save as `filename.h`
  - Load in other files with `#include "filename.h"`
  - Reuse functions



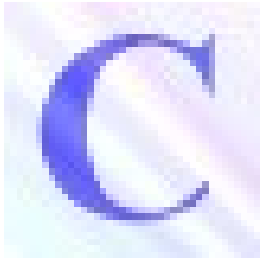
Some of the standard library headers.

Header	Explanation
<assert.h>	Contains macros and information for adding diagnostics that aid program debugging.
<ctype.h>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
<errno.h>	Defines macros that are useful for reporting error conditions.
<float.h>	Contains the floating-point size limits of the system.
<limits.h>	Contains the integral size limits of the system.
<locale.h>	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.
<math.h>	Contains function prototypes for math library functions.
<setjmp.h>	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.



Some of the standard library headers.

Header	Explanation
<signal.h>	Contains function prototypes and macros to handle various conditions that may arise during program execution.
<stdarg.h>	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
<stddef.h>	Contains common type definitions used by C for performing calculations.
<stdio.h>	Contains function prototypes for the standard input/output library functions, and information used by them.
<stdlib.h>	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
<string.h>	Contains function prototypes for string-processing functions.
<time.h>	Contains function prototypes and types for manipulating the time and date.

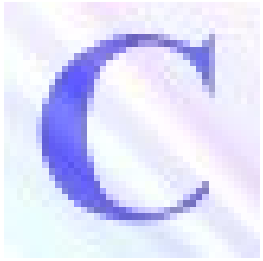


Calling Functions

- Function returning no value:

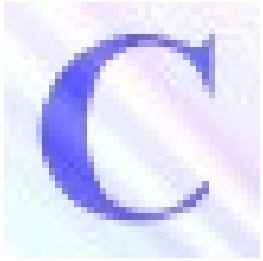
```
name(effective_parameter_list);
```

- Function returning a value
  - As above, loose returned value
  - As an operand of an expression, returned value used in expression evaluation
- Correspondence between formal and effective parameters is *positional*
  - If an effective parameter type is different than the formal parameter type, an automatic conversion occurs



Calling Functions

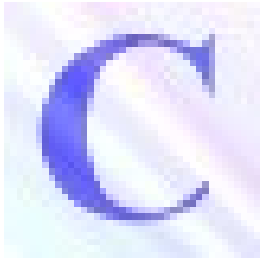
- Used when invoking functions
- Call by *value*
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by *reference* (in C++ only)
  - Passes original argument
  - Changes in function affect original
  - Only used with trusted functions



Calling Functions. Call by value. Example

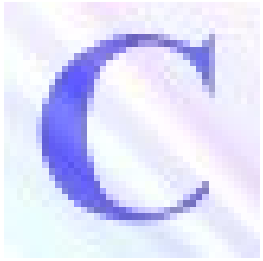
```
#include <stdio.h>
#include <stdlib.h>
/* swap values of a and b */
void swap(int a, int b)
{
   int aux;
   printf("\nin swap upon entry: a=%d b=%d\n", a, b);
   aux = a; a = b; b = aux;
   printf("\nin swap at exit: a=%d b=%d\n", a, b);
}
int main()
{
   int a=3, b=2;
   printf("\nin main before invoking swap: a=%d b=%d\n", a, b)
   swap(a, b);
   printf("\nin main after invoking swap: a=%d b=%d\n", a, b)
   system("pause");
   return 0;
}
```

swap will have no effect in main() as parameters are passed by value



Calling Functions. Call by value using pointers. Example

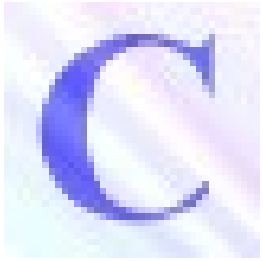
```
#include <stdio.h>
#include <stdlib.h>
/* swap values of a and b */
void swap(int *a, int *b)
{
   int aux;
   printf("\nin swap upon entry: a=%d b=%d\n", *a, *b);
   aux = *a; *a = *b; *b = aux;
   printf("\nin swap at exit: a=%d b=%d\n", *a, *b);
}
int main()
{
   int a=3, b=2;
   printf("\nin main before invoking swap: a=%d b=%d\n", a, b);
   swap(&a, &b);
   printf("\nin main after invoking swap: a=%d b=%d\n", a, b);
   system("pause");
   return 0;
}
```



Calling functions

- Notes:
  - When an effective parameter is the name of an array the function can change the elements of that array
  - For *single-dimensional* arrays, a formal parameter can be declared as:
    - an array: `type formal_param_name[]`
    - a pointer: `type *formal_param_name`
    - The two are equivalent, in the body you can use the indexed variable `formal_param_name[index]`
  - When values of array elements or the referred value should not be changed by a function use pointer to constant construct for formal parameters

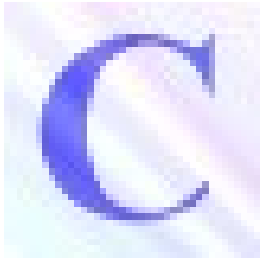
```
const type *identifier
```



Calling functions. Call by reference

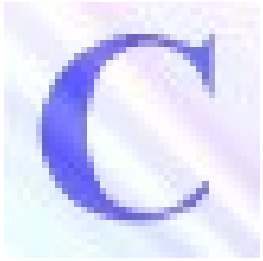
- Exists only in C++. Uses & after type spec
- Identical to PASCAL call by reference
- Example:

```
#include <stdio.h>
#include <stdlib.h>
/* swap values of a and b */
void swap(int& a, int& b)
{
    int aux;
    printf("\nin swap upon entry: a=%d b=%d\n", a, b);
    aux = a; a = b; b = aux;
    printf("\nin swap at exit: a=%d b=%d\n", a, b);
}
int main()
{
    int a=3, b=2;
    printf("\nin main before invoking swap: a=%d b=%d\n", a, b);
    swap(a, b);
    printf("\nin main after invoking swap: a=%d b=%d\n", a, b);
    system("pause");
    return 0;
}
```



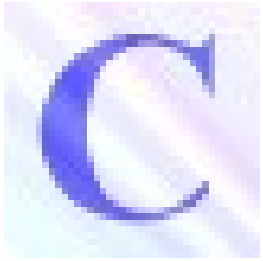
What makes a good Function?

- It is *called several times*
- It helps make the *calling code more compact and more readable*
- It *does just one well-defined task*, and does it well.
- Its *interface* to the rest of the program is *clean and narrow*.



The scope of a name

- The scope of a name is the part of a program over which the name is visible
- The scope defines the section of a program where the name can be legally used
- Examples:
  - a global variable has *global scope:* it is accessible throughout a program
  - a variable defined inside a function has *function scope*: it is only accessible within the defining function



Variable scope

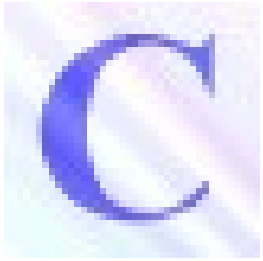
- Global variables
  - Defined at the beginning of a source file
  - Visible from the point of their definition to the end of the file
  - Declaration:

```
type identifier{, identifier};
static type identifier{, identifier};
```

- Extern variables: visible from other source files than the one containing the definition

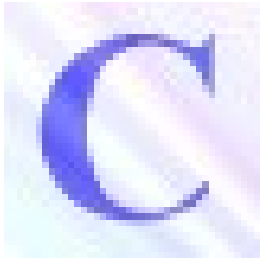
```
extern type identifier{, identifier};
```

  - can be declared:
    - after a function header. Scope is within function
    - at the beginning of a source file: scope is all functions in that file



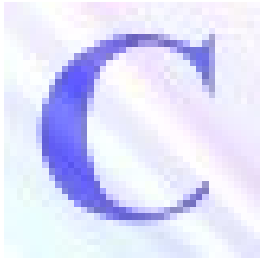
Global variable dangers

- Global variables are useful to share information across functions but they must be used with great caution because:
  - they introduce coupling between different parts of the same program
  - they make a program less readable
  - they make a program less maintainable
  - they may introduce name clashes



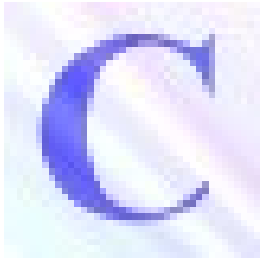
Variable scope

- Local variables
  - declared in a function or in a block
  - scope is that unit
  - categories:
    - automatic
      - allocated on the stack at run time
      - cease to exist upon function return, or when control leaves block
      - example: `int a, b, c; double x;`
    - static
      - allocated by the compiler in a special area
      - persist during program execution
      - cannot be declared extern
      - example: `static int x, y, z;`
    - register
      - allocated in the registers of the processor
      - declared using: `register type variable_name;`



Automatic Variables

- Automatic (or stack) variables are declared at the beginning of a function
- Their scope is the function where they are declared
- Automatic variables exist only while the function is being executed: they are destroyed when the function is exited and re-created when it is re-entered

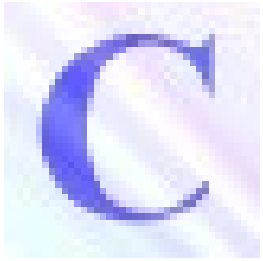


Memory layout for C programs

<table>
<tr><td>High Address</td><td>Args and env vars</td><td>Command line arguments and environment variables</td></tr>
<tr><td rowspan="5"></td><td>Stack
|
V</td><td rowspan="3"></td></tr>
<tr><td>Unused memory</td></tr>
<tr><td>^
|
Heap</td></tr>
<tr><td>Uninitialized Data Segment (bss)</td><td>Initialized to zero by exec.</td></tr>
<tr><td>Initialized Data Segment</td><td>Read from the program file by exec.</td></tr>
<tr><td>Low Address</td><td>Text Segment</td><td>Read from the program file by exec.</td></tr>
</table>

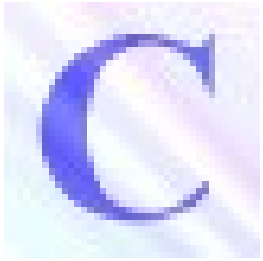
The Basic Call/Return Process

- The following things happen during a C call and return:
  1. The arguments to the call are evaluated and put in an agreed place.
  2. The return address is saved on the stack.
  3. Control passes to the called procedure.
  4. The bookkeeping registers and register variables of the calling procedure are saved so that their values can be restored on return.
  5. The called procedure obtains stack space for its local variables, and temporaries.
  6. The bookkeeping registers for the called procedure are appropriately initialized. By now, the called procedure must be able to find the argument values.
  7. The body of the called procedure is executed.
  8. The returned value, if any, is put in a safe place while the stack space is freed, the calling procedure's register variables and bookkeeping registers are restored, and the return address is obtained and transferred to.



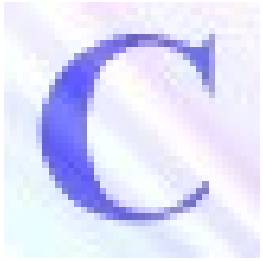
The C Stack

- The C run-time system manages an internal data structure that is called the "C Stack" and which it uses to create and manipulate temporary variables
- The stack is filled and emptied in LIFO order (last-in-first-out)
- The "stack pointer" points to the head of the stack (the last stack location to have been filled)
- The C stack is used to:
  - store automatic variables
  - store function parameters



The C stack. Example

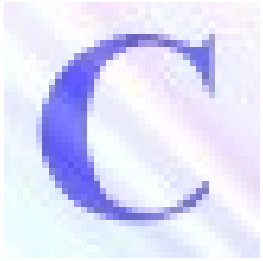
```
void foo (int i, char *name)
{
   char LocalChar[24];
   int LocalInt;
   ...
}
int main(int argc, char *argv[])
{
   int MyInt=1; // stack variable located at ebp-8
   char *MyStrPtr="MyString"; // stack var at ebp-4
   ...
   foo(MyInt,MyStrPtr); // call foo function
   ...
}
```



The C stack. Example

- Sample stack frame `foo()` that takes two arguments and contains four local variables.
  - The low memory is at the top of the stack, so in this illustration the stack grows toward lower memory.

`foo(int i, char* name)`			
Address	**Value**	**Description**	**Len**
0x0012FF4C	?	Last Local Variable - Integer - LocalInt	4
0x0012FF50	?	First Local Variable - String - LocalChar	24
0x0012FF68	0x12FF80	Calling Frame of Calling Function: main()	4
0x0012FF6C	0x401040	Return Address of Calling Function: main()	4
0x0012FF70	1	Arg: 1st argument: MyInt (int)	4
0x0012FF74	0x40703C	Arg: 2nd argument: Pointer toMyString (char *)	4

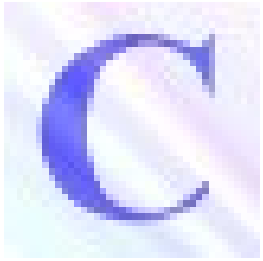


Static automatic variables

- Automatic variables are destroyed after the function returns. What can be done to preserve their value across function calls?
- Automatic variables that are declared `static` have permanent storage: their value is preserved across function calls
- Example: a function that maintains a permanent storage for float values:

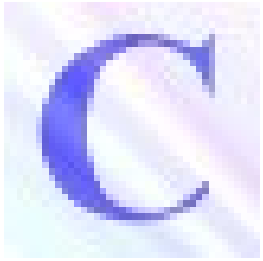
```
int store(float val)                    /* name and return value */
                                        /* parameter declaration */
{
   static float storageArray[100]; /* local variable */
   static int nOfEntries=0;         /* local variable */

   storageArray[nOfEntries]=val;   /* store new value */
   nOfEntries++;                   /* increment counter */
   return nOfEntries;              /* return */
}
```



Scope rules (I)

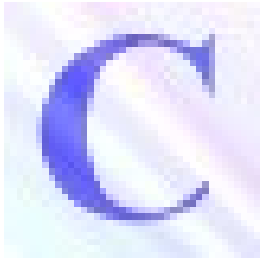
- File scope
  - Identifier defined outside function, known in all functions
  - Global variables, function definitions, function prototypes
- Function scope
  - Can only be referenced inside a function body
  - Only labels (`start:` `case:`, etc.)



Scope rules (II)

- Block scope
  - Identifier declared inside a block
    - Block scope begins at declaration, ends at right brace
  - Variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if same variable name

- Function prototype scope
  - Identifiers in parameter list
  - Names in function prototype optional, and can be used anywhere



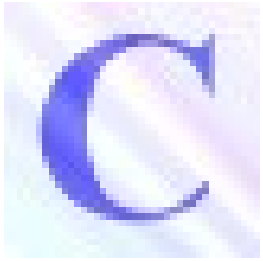
Strings

- A character string is stored in an single-dimensional array of type **char**
- The last character in the array is the ASCII NUL ('\0');
- The name of the array is a constant pointer to the first element
- Example:

```
char string[]="Character string";
```

stored in (first row: indices in `string`; second row: hex values of bytes):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
43	68	61	72	61	63	74	65	72	20	73	74	72	69	6e	67	00



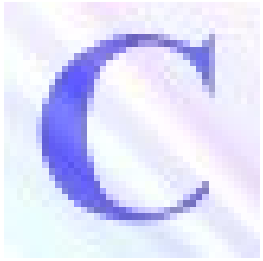
Strings

- Relations (for the previous example):
  - `string[i]`, with $i \in [0,15]$ is the ASCII code for the i th character of the string
  - `*(string+i)`, with $i \in [0,15]$ is the ASCII code for the i th character of the string
  - `*(string+i)≡string[i]`
- Character string array declaration:
  - `char *arr[]={"string0", "string1", "string2", ..., "stringn"}`
  - `arr[i]`, for $i \in [0,n]$ is a pointer to `"stringi"`,
  - `arr[i]` can be printed with `printf("%s\n", arr[i]);`

Character handling library

- Prototypes in `ctype.h`
- Note that in C, **true** means <> 0, and **false** means 0.

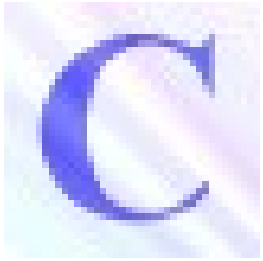
Prototype	Description
int isdigit(int c)	Returns **true** if **c** is a digit and **false** otherwise.
int isalpha(int c)	Returns **true** if **c** is a letter and **false** otherwise.
int isalnum(int c)	Returns **true** if **c** is a digit or a letter and **false** otherwise.
int isxdigit(int c)	Returns **true** if **c** is a hexadecimal digit character and **false** otherwise.
int islower(int c)	Returns **true** if **c** is a lowercase letter and **false** otherwise.
int isupper(int c)	Returns **true** if **c** is an uppercase letter; **false** otherwise.
int tolower(int c)	If **c** is an uppercase letter, **tolower** returns **c** as a lowercase letter. Otherwise, **tolower** returns the argument unchanged.



Character handling library

- Prototypes in `ctype.h`

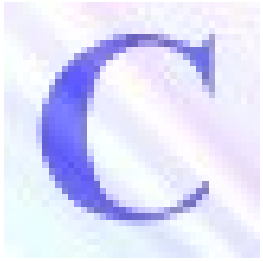
Prototype	Description
int toupper(int c)	If **c** is a lowercase letter, **toupper** returns **c** as an uppercase letter. Otherwise, **toupper** returns the argument unchanged.
int isspace(int c)	Returns **true** if **c** is a white-space character—newline (**'\n'**), space (**' '**), form feed (**'\f'**), carriage return (**'\r'**), horizontal tab (**'\t'**), or vertical tab (**'\v'**)—and **false** otherwise
int iscntrl(int c)	Returns **true** if **c** is a control character and **false** otherwise.
int ispunct(int c)	Returns **true** if **c** is a printing character other than a space, a digit, or a letter and **false** otherwise.
int isprint(int c)	Returns **true** value if **c** is a printing character including space (**' '**) and **false** otherwise.
int isgraph(int c)	Returns **true** if **c** is a printing character other than space (**' '**) and **false** otherwise.



String Conversion Functions

- Conversion functions
  - Prototypes in `<stdlib.h>` (general utilities library)
  - Convert strings of digits to integer and floating-point values

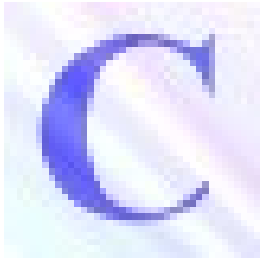
Prototype	Description
double atof(const char *nPtr)	Converts the string **nPtr** to **double**.
int atoi(const char *nPtr)	Converts the string **nPtr** to **int**.
long atol(const char *nPtr)	Converts the string **nPtr** to long **int**.
double strtod(const char *nPtr, char **endPtr)	Converts the string **nPtr** to **double.**
long strtol(const char *nPtr, char **endPtr, int base)	Converts the string **nPtr** to **long**.
unsigned long strtoul(const char *nPtr, char **endPtr, int base)	Converts the string **nPtr** to **unsigned long**.



String Manipulation Functions of the String Handling Library

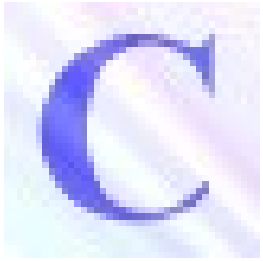
- String handling library has functions to
  - Manipulate string data
  - Search strings
  - Tokenize strings
  - Determine string length

Function prototype	Function description
char *strcpy(char *s1, const char *s2)	Copies string **s2** into array **s1**. The value of **s1** is returned.
char *strncpy(char *s1, const char *s2, size_t n)	Copies at most **n** characters of string **s2** into array **s1**. The value of **s1** is returned.
char *strcat(char *s1, const char *s2)	Appends string **s2** to array **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
char *strncat(char *s1, const char *s2, size_t n)	Appends at most **n** characters of string **s2** to array **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.



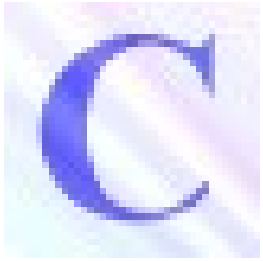
Comparison Functions of the String Handling Library

- Comparing strings
  - Computer compares numeric ASCII codes of characters in string
- `int strcmp( const char *s1, const char *s2 );`
  - Compares string `s1` to `s2`
  - Returns a negative number (`s1` < `s2`), zero (`s1` == `s2`), or a positive number (`s1` > `s2`)
- `int strncmp( const char *s1, const char *s2,size_t n );`
  - Compares up to `n` characters of string `s1` to `s2`
  - Returns values as above



Search Functions of the String Handling Library

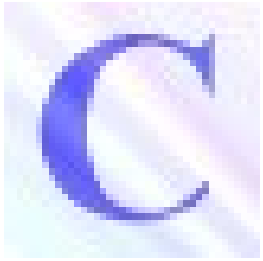
Function prototype	Function description
char *strchr(const char *s, int c);	Locates the first occurrence of character **c** in string **s**. If **c** is found, a pointer to **c** in **s** is returned. Otherwise, a **NULL** pointer is returned.
size_t strcspn(const char *s1, const char *s2);	Determines and returns the length of the initial segment of string **s1** consisting of characters not contained in string **s2**.
size_t strspn(const char *s1, const char *s2);	Determines and returns the length of the initial segment of string **s1** consisting only of characters contained in string **s2**.
char *strpbrk(const char *s1, const char *s2);	Locates the first occurrence in string **s1** of any character in string **s2**. If a character from string **s2** is found, a pointer to the character in string **s1** is returned. Otherwise, a **NULL** pointer is returned.



Search Functions of the String Handling Library. Example

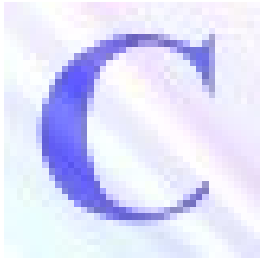
- Find the first occurrence of a given character in a character string. Return a pointer to the character if found, NULL otherwise.

```
char *FindCharInString(char *str, int ch)
{
   if (str == NULL) return NULL;
   while (*str != 0 && *str != ch)
   {
       str++;
   }
   if (*str == ch)
       return str;
   return NULL;
}
```



Memory Functions of the String- handling Library

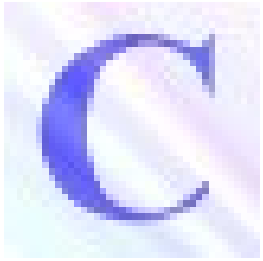
- Memory Functions
  - Prototypes in `<stdlib.h>`
  - Manipulate, compare, and search blocks of memory
  - Can manipulate any block of data
- Pointer parameters are `void *`
  - Any pointer can be assigned to `void *`, and vice versa
  - `void *` *cannot* be dereferenced
    - Each function receives a size argument specifying the number of bytes (characters) to process



Memory Functions of the String-handling Library

Note. In the table below, "object" refers to a block of data

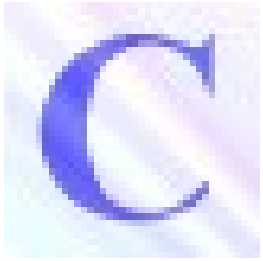
Prototype	Description
void *memcpy(void *s1, const void *s2, size_t n)	Copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. A pointer to the resulting object is returned.
void *memmove(void *s1, const void *s2, size_t n)	Copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. The copy is performed as if the characters are first copied from the object pointed to by **s2** into a temporary array, and then copied from the temporary array into the object pointed to by **s1**. A pointer to the resulting object is returned.



Memory Functions of the String-handling Library

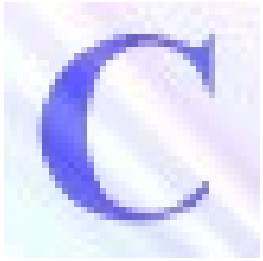
Note. In the table below, "object" refers to a block of data

Prototype	Description
int memcmp(const void *s1, const void *s2, size_t n)	Compares the first **n** characters of the objects pointed to by **s1** and **s2**. The function returns **0**, less than **0**, or greater than **0** if **s1** is equal to, less than or greater than **s2**, respectively.
void *memchr(const void *s, int c, size_t n)	Locates the first occurrence of **c** (converted to **unsigned char**) in the first **n** characters of the object pointed to by **s**. If **c** is found, a pointer to **c** in the object is returned. Otherwise, **0** is returned.
void *memset(void *s, int c, size_t n)	Copies **c** (converted to **unsigned char**) into the first **n** characters of the object pointed to by **s**. A pointer to the result is returned.



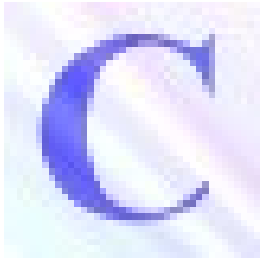
Other Functions of the String Handling Library

- **`char *strtok(char *newstring, const char *delimiters)`**
  - Splits **`newstring`** into tokens by making a series of calls to the function **`strtok`**.
  - Split occurs when a character in **`delimiters`** is encountered
  - The string to be split up is passed as the **`newstring`** argument on the first call only. The **`strtok`** function uses this to set up some internal state information.
  - Subsequent calls to get additional tokens from the same string are indicated by passing a null pointer as the **`newstring`** argument. Calling **`strtok`** with another non-null **`newstring`** argument reinitializes the state information.



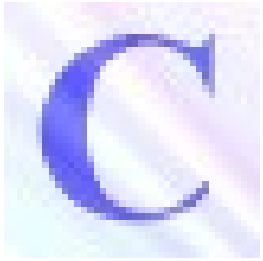
Other Functions of the String Handling Library

- **`char * strdup (const char *s)`**
  - Copies the null-terminated string `s` into a newly allocated string. The string is allocated using `malloc`. If `malloc` cannot allocate space for the new string, `strdup` returns a null pointer. Otherwise it returns a pointer to the new string.
- **`char *strerror( int errornum );`**
  - Creates a system-dependent error message based on `errornum`
  - Returns a pointer to the string
- **`size_t strlen( const char *s );`**
  - Returns the number of characters (before ASCII NUL) in string `s`



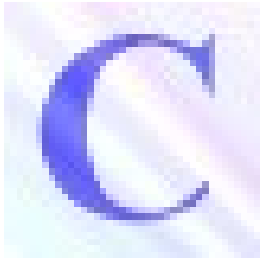
Examples

- Demo for string handling library



Reading

- King: chapters 9, 13
- Prata: chapters 9, 11
- Deitel: chapters 5, 8



Summary

- Functions
  - Structure of a function
  - Function invocation
  - Parameter passing
  - Functions as parameters
  - Variable scope
  - Functions for character processing