



**UNIVERSITÀ  
DEGLI STUDI  
DI TRIESTE**



Dipartimento di  
**Ingegneria  
e Architettura**

## **The C Language**

**A. Carini – Digital System Architectures**

# C language

- One of the most popular programming languages ever developed.
- It was created by a group including Dennis Ritchie and Brian Kernighan at Bell Laboratories between 1969 and 1973 to rewrite the UNIX operating system from its original assembly language.
- C, with C++, C#, Objective C, is the most widely used language in existence.
- Its popularity stems from
  - Availability on a tremendous variety of platforms.
  - Relative ease of use
  - Moderate level of abstraction providing higher productivity than assembly language
  - Suitability for generating high performance programs
  - Ability to interact directly with the hardware
    - C allows the programmer to directly access addresses in memory
- Formally introduced in 1978 by Kernighan and Ritchie's classic book, The C Programming Language.
- In 1989, the American National Standards Institute (ANSI) expanded and standardized the language, which became known as ANSI C, Standard C, or C89.
- International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) updated the standard in 1999, 2011, 2017 to what is called C99, C11, C17 (C18), respectively.

# Welcome to C

- A C program is a text file that describes operations for the computer to perform.
- The text file is *compiled*, converted into a machine-readable format, and run or *executed* on a computer.
- C programs are generally contained in one or more text files that end in “.c”.

```
// Write "Hello world!" to the console
#include <stdio.h>

int main(void){
    printf("Hello world!\n");
}
```

## Console Output

```
Hello world!
```

# C Program Dissection

- A C program is organized into one or more functions.
- Every program must include the **main** function, which is where the program starts executing.
- Most programs use other functions defined elsewhere in the C code and/or in a library.
- The overall sections of the hello.c program are the **header**, the **main function**, and the **body**.
- **Header:** `#include <stdio.h>`
  - The header includes the library functions needed by the program.
  - The program uses the `printf` function, which is part of the standard I/O library, `stdio.h`.
- **Main function:** `int main(void)`
  - All C programs must include exactly one main function.
  - Execution of the program occurs by running the code inside main, called the body of main.
  - The body of a function contains a sequence of *statements*.
  - Each statement ends with a *semicolon*.
  - **Int** denotes that the main function returns an integer that indicates whether the program ran successfully.
- **Body:** `printf("Hello world!\n");`
  - The body contains one statement, a call to the `printf` function. `\n` is a newline character

# Running a C Program

- C programs can be run on many different machines.
  - The program is first compiled on the desired machine using the *C compiler*.
  - We show how to compile and run a C program using **gcc**, which is freely available for download.
  - It runs directly on Linux machines and is accessible under the Cygwin (<https://www.cygwin.com/>) environment or on WSL – Windows for Linux (<https://learn.microsoft.com/it-it/windows/wsl/install>) on Windows.
1. Create the text file, for example hello.c.
  2. In a terminal window, change to the directory that contains the file hello.c and type **gcc hello.c** at the command prompt.
  3. The compiler creates an executable file. By default, the executable is a.exe on Windows and a.out on Linux.
  4. At a command prompt, type **./a.exe** (or ./a.out on Linux) and press return.
  5. “Hello world!” will appear on the screen.

# Compilation

- A compiler is a piece of software that reads a program in a high-level language and converts it into a file of machine code called an executable.
- The overall operation of the compiler is to
  - (1) preprocess the file by including referenced libraries and expanding macro definitions,
  - (2) ignore all unnecessary information such as comments,
  - (3) translate the high-level code into machine language, and
  - (4) compile all the instructions into a single binary executable that can be read and executed by the computer.
- Each machine language is specific to a given processor, so a program must be compiled specifically for the system on which it will run.

# Comments

- C programs use two types of comments:
  - Single-line comments begin with `//` and terminate at the end of the line;
  - multiple-line comments begin with `/*` and end with `*/`.

```
// This is an example of a one-line comment.
```

```
/* This is an example  
of a multi-line comment. */
```

```
// hello.c  
// 1 Jan 2015 Sarah_Harris@hmc.edu, David_Harris@hmc.edu  
//  
// This program prints "Hello world!" to the screen
```

# #define

- Constants are named using the **#define** directive and then used by name throughout the program. These globally defined constants are also called *macros*.

```
#define MAXGUESSES 5
```

- The # indicates that this line in the program will be handled by the pre-processor.
- Before compilation, the preprocessor replaces each MAXGUESSES in the program with 5.
- By convention, #define lines are **located at the top** of the file and identifiers are written in all capital letters.
- Number constants in C by default are decimal but can also be hexadecimal (prefix "0x") or octal (prefix "0"). Binary constants are not defined in C99 but are supported by some compilers (prefix "0b")

```
char x = 37;  
char x = 0x25;  
char x = 045;
```



# #define

```
// Convert inches to centimeters
#include <stdio.h>

#define INCH2CM 2.54

int main(void) {
    float inch = 5.5;    // 5.5 inches
    float cm;

    cm = inch * INCH2CM;
    printf("%f inches = %f cm\n", inch, cm);
}
```

## Console Output

```
5.500000 inches = 13.970000 cm
```

# #include

- Modularity encourages us to split programs across separate files and functions.
- Variable declarations, defined values, and function definitions located in a **header file** can be used by another file by adding the **#include** preprocessor directive.
- Standard libraries that provide commonly used functions are accessed in this way. E.g.,

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

- The “.h” postfix of the include file indicates it is a header file.
- While #include directives can be placed anywhere in the file, they are conventionally located at the top of a C file.
- Programmer-created header files can also be included by using quotation marks (" ") around the file name instead of brackets (< >).

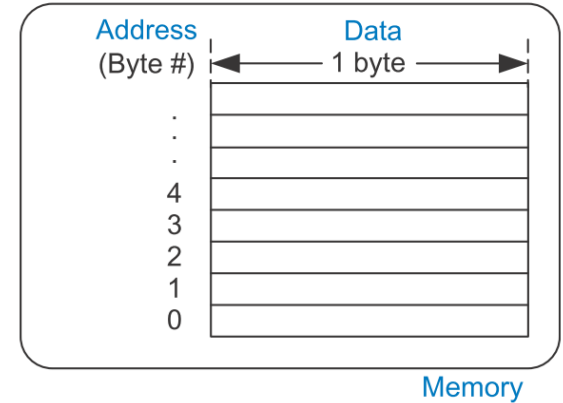
```
#include "myfunctions.h"
```

- At compile time, files specified in brackets are searched for in system directories.
- Files specified in quotes are searched for in the same local directory where the C file is found (or in the specified path relative to the current directory).

# Variables

- Variables in C programs have a type, name, value, and memory location.
- A variable declaration states the type and name of the variable.
- For example, a variable of type **char**, which is a 1-byte type,

```
char x;
```



- Variable names are case sensitive, may not be any of C's reserved words, start with a letter, and include special characters such as `\`, `*`, `?`, or `-`. Underscores (`_`) are allowed.
- C views memory as a group of consecutive bytes, where each byte of memory is assigned a unique number indicating its address.
- A variable occupies one or more bytes of memory, and the address of multiple-byte variables is indicated by the lowest numbered byte.
- The type of a variable indicates whether to interpret the byte(s) as an integer, floating point number, or other type.

# Primitive Data Types

Type	Size (bits)	Minimum	Maximum
char	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
unsigned char	8	0	$2^8 - 1 = 255$
short	16	$-2^{15} = -32,768$	$2^{15} - 1 = 32,767$
unsigned short	16	0	$2^{16} - 1 = 65,535$
long	32	$-2^{31} = -2,147,483,648$	$2^{31} - 1 = 2,147,483,647$
unsigned long	32	0	$2^{32} - 1 = 4,294,967,295$
long long	64	$-2^{63}$	$2^{63} - 1$
unsigned long long	64	0	$2^{64} - 1$
int	machine-dependent		
unsigned int	machine-dependent		
float	32	$\pm 2^{-126}$	$\pm 2^{127}$
double	64	$\pm 2^{-102}$	$\pm 2^{102}$

But in Linux  
64 bits

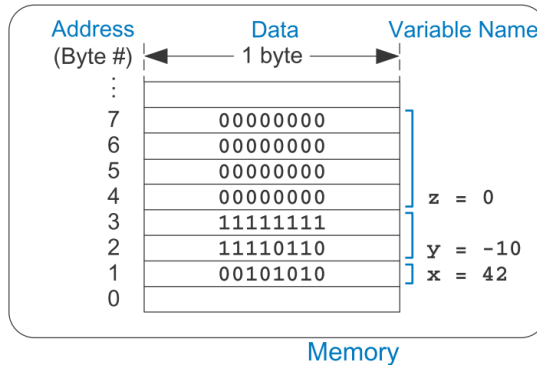
# Primitive Data Types

- **Char** should always be 1 byte (but in some non-conforming machine it has also 16 bit, e.g. TI C54).
  - It is an integer type, that can be used for operations.
  - Characters are associated with integers.
- Sizes of data types are machine dependent, but it is always guaranteed that  
 $\text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)} \leq \text{sizeof(long long)}$

# Example data types

```
// Examples of several data types and their binary representations
unsigned char x = 42;      // x = 00101010
short y = -10;           // y = 11111111 11110110
unsigned long z = 0;     // z = 00000000 00000000 00000000 00000000
```

- Shows the declaration of variables of different types.
- x requires one byte of data, y requires two, and z requires four.
- The program decides where these bytes are stored in memory, but each type always requires the same amount of data.



# Global and Local Variables

- Global and local variables differ in where they are declared and where they are visible.
- A **global variable** is declared outside of all functions, typically at the top of a program, and can be accessed by all functions.
  - Global variables should be used sparingly because they violate the principle of modularity.
- A **local variable** is declared inside a function and can only be used by that function.
  - Two functions can have local variables with the same names without interfering with each other.
  - Local variables are declared at the beginning of a function.
  - They cease to exist when the function ends and are recreated when the function is called again.
  - They do not retain their value from one invocation of a function to the next.

# Global Variables

```
// Use a global variable to find and print the maximum of 3 numbers
int max;           // global variable holding the maximum value

void findMax(int a, int b, int c) {
    max = a;
    if (b > max) {
        if (c > b) max = c;
        else     max = b;
    } else if (c > max) max = c;
}

void printMax(void) {
    printf("The maximum number is: %d\n", max);
}

int main(void) {
    findMax(4, 3, 7);
    printMax();
}
```



# Local Variables

```
// Use local variables to find and print the maximum of 3 numbers

int getMax(int a, int b, int c) {
    int result = a; // local variable holding the maximum value

    if (b > result) {
        if (c > b) result = c;
        else      result = b;
    } else if (c > result) result = c;

    return result;
}

void printMax(int m) {
    printf("The maximum number is: %d\n", m);
}

int main(void) {
    int max;

    max = getMax(4, 3, 7);
    printMax(max);
}
```

# Initializing Variables

- A variable needs to be initialized – assigned a value – before it is read.
- When a variable is declared, the correct number of bytes is reserved for that variable in memory.
- However, the memory at those locations retains whatever value it had last time it was used, essentially a random value.
- Global and local variables can be initialized either when they are declared or within the body of the program.

```
unsigned char x = 42;  
short y = -10;  
unsigned long z = 0;
```

# Operators

- The most common type of statement in a C program is an expression, such as

$$y = a + 3;$$

- An expression involves operators (such as + or \*) acting on one or more operands, such as variables or constants.
- C supports the operators shown in next slides, listed by category and in order of decreasing precedence.
- For example, multiplicative operators take precedence over additive operators.
- Within the same category, operators are evaluated in the order that they appear in the program.
- Unary operators, also called monadic operators, have a single operand; binary operators have two operands; ternary operators have three.

# Operators

Category	Operator	Description	Example
Unary	++	post-increment	<code>a++; // a = a+1</code>
	--	post-decrement	<code>x--; // x = x-1</code>
	&	memory address of a variable	<code>x = &amp;y; // x = the memory // address of y</code>
	~	bitwise NOT	<code>z = ~a;</code>
	!	Boolean NOT	<code>!x</code>
	-	negation	<code>y = -a;</code>
	++	pre-increment	<code>++a; // a = a+1</code>
	--	pre-decrement	<code>--x; // x = x-1</code>
	(type)	casts a variable to (type)	<code>x = (int)c; // cast c to an // int and assign it to x</code>
sizeof()	size of a variable or type in bytes	<code>long int y; x = sizeof(y); // x = 4</code>	
Multiplicative	*	multiplication	<code>y = x * 12;</code>
	/	division	<code>z = 9 / 3; // z = 3</code>
	%	modulo	<code>z = 5 % 2; // z = 1</code>

# Operators

Category	Operator	Description	Example
Additive	+	addition	<code>y = a + 2;</code>
	-	subtraction	<code>y = a - 2;</code>
Bitwise Shift	<<	bitshift left	<code>z = 5 &lt;&lt; 2; // z = 0b00010100</code>
	>>	bitshift right	<code>x = 9 &gt;&gt; 3; // x = 0b00000001</code>
Relational	==	equals	<code>y == 2</code>
	!=	not equals	<code>x != 7</code>
	<	less than	<code>y &lt; 12</code>
	>	greater than	<code>val &gt; max</code>
	<=	less than or equal	<code>z &lt;= 2</code>
	>=	greater than or equal	<code>y &gt;= 10</code>
Bitwise	&	bitwise AND	<code>y = a &amp; 15;</code>
	^	bitwise XOR	<code>y = 2 ^ 3;</code>
		bitwise OR	<code>y = a   b;</code>
Logical	&&	Boolean AND	<code>x &amp;&amp; y</code>
		Boolean OR	<code>x    y</code>

# Operators

Category	Operator	Description	Example
Ternary	? :	ternary operator	<code>y = x ? a : b; // if x is TRUE, // y=a, else y=b</code>
Assignment	=	assignment	<code>x = 22;</code>
	+=	addition and assignment	<code>y += 3; // y = y + 3</code>
	-=	subtraction and assignment	<code>z -= 10; // z = z - 10</code>
	*=	multiplication and assignment	<code>x *= 4; // x = x * 4</code>
	/=	division and assignment	<code>y /= 10; // y = y / 10</code>
	%=	modulo and assignment	<code>x %= 4; // x = x % 4</code>
	>>=	bitwise right-shift and assignment	<code>x &gt;&gt;= 5; // x = x &gt;&gt; 5</code>
	<<=	bitwise left-shift and assignment	<code>x &lt;&lt;= 2; // x = x &lt;&lt; 2</code>
	&=	bitwise AND and assignment	<code>y &amp;= 15; // y = y &amp; 15</code>
	=	bitwise OR and assignment	<code>x  = y; // x = x   y</code>
^=	bitwise XOR and assignment	<code>x ^= y; // x = x ^ y</code>	

# Ternary operator

```
(a) y = (a > b) ? a : b; // parentheses not necessary, but makes it clearer
```

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

- C considers a variable to be TRUE if it is nonzero and FALSE if it is zero.
- Logical and ternary operators, as well as control-flow statements such as if and while, depend on the truth of a variable.
- Relational and logical operators produce a result that is 1 when TRUE or 0 when FALSE.

# Operator examples

Expression	Result	Notes
44 / 14	3	Integer division truncates
44 % 14	2	44 mod 14
0x2C && 0xE //0b101100 && 0b1110	1	Logical AND
0x2C    0xE //0b101100    0b1110	1	Logical OR
0x2C & 0xE //0b101100 & 0b1110	0xC (0b001100)	Bitwise AND
0x2C   0xE //0b101100   0b1110	0x2E (0b101110)	Bitwise OR
0x2C ^ 0xE //0b101100 ^ 0b1110	0x22 (0b100010)	Bitwise XOR
0xE << 2 //0b1110 << 2	0x38 (0b111000)	Left shift by 2
0x2C >> 3 //0b101100 >> 3	0x5 (0b101)	Right shift by 3
x = 14; x += 2;	x=16	
y = 0x2C; // y = 0b101100 y &= 0xF; // y &= 0b1111	y=0xC (0b001100)	
x = 14; y = 44; y = y + x++;	x=15, y=58	Increment x after using it
x = 14; y = 44; y = y + ++x;	x=15, y=59	Increment x before using it

- \*= or += are **compound assignments**
- x += 10; is equivalent to x = x + 10;



# Function calls

- Large programs are divided into functions with well-defined inputs, outputs, and behavior.

```
// Return the sum of the three input variables
int sum3(int a, int b, int c) {
    int result = a + b + c;
    return result;
}
```

- The function declaration begins with the return type, int, followed by the name, sum3, and the inputs enclosed within parentheses(int a, int b, int c).
- Curly braces {} enclose the body of the function, which may contain zero or more statements.
- The return statement indicates the value returned to the caller, i.e., the output of the function.
- A function can only return a single value.

```
int y = sum3(10, 15, 17);
```

- After this call to sum3, y holds the value 42.

# Function calls

- Although a function may have inputs and outputs, neither is required.

```
// Print a prompt to the console
void printPrompt(void)
{
    printf("Please enter a number from 1-3:\n");
}
```

- The keyword **void** before the function name indicates that nothing is returned.
- **void** between the parentheses indicates that the function has no input arguments.

# Function prototype

- A function must be declared in the code before it is called.
- This may be done by placing the called function earlier in the file, with **main** placed **at the end** of the C file after all the functions it calls.
- Alternatively, a **function prototype** can be placed in the program before the function is defined.
  - The function prototype *is the first line of the function*, declaring the return type, function name, and function inputs.
  - It is good style to place prototypes for all of a program's functions near the beginning of the C file or in a header file.

# Function prototype

```
#include <stdio.h>

// function prototypes
int sum3(int a, int b, int c);
void printPrompt(void);

int main(void)
{
    int y = sum3(10, 15, 20);

    printf("sum3 result: %d\n", y);
    printPrompt();
}

int sum3(int a, int b, int c) {
    int result = a+b+c;
    return result;
}

void printPrompt(void) {
    printf("Please enter a number from 1-3:\n");
}
```

## Console Output

```
sum3 result: 45
Please enter a number from 1-3:
```

# Control flow statements

- C provides control-flow statements for conditionals and loops.
- Conditionals execute a statement only if a condition is met: **if**, **if/else**, and **switch/case**
- A loop repeatedly executes a statement as long as a condition is met: **while**, **do/while**, and **for** loops

# If and if/then statements

- An if statement executes the statement immediately following it when the expression in parentheses is TRUE (i.e., nonzero).
- The general format is:

```
if (expression)
    statement
```

```
int dontFix = 0;
if (aintBroke == 1)
    dontFix = 1;
```

- Curly braces, {}, are used to group one or more statements into a compound statement or **block**.

```
// If amt >= $2, prompt user and dispense candy
if (amt >= 2) {
    printf("Select candy.\n");
    dispenseCandy = 1;
}
```

- if/else statements execute one of two statements depending on a condition:

```
if (expression)
    statement1
else
    statement2
```

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

# switch/case statements

- switch/case statements execute one of several statements depending on the value of an expression:

```
switch (variable) {  
    case (expression1): statement1 break;  
    case (expression2): statement2 break;  
    case (expression3): statement3 break;  
    default:             statement4  
}
```

- If the keyword `break` is omitted, execution begins at the point where the condition is `TRUE` and then falls through to execute the remaining cases below it.

```
// Assign amt depending on the value of option  
switch (option) {  
    case 1:  amt = 100; break;  
    case 2:  amt = 50;  break;  
    case 3:  amt = 20;  break;  
    case 4:  amt = 10;  break;  
    default: printf("Error: unknown option.\n");  
}
```

- A switch/case statement is equivalent to a series of nested if/else statements:

```
// Assign amt depending on the value of option  
if (option == 1) amt = 100;  
else if (option == 2) amt = 50;  
else if (option == 3) amt = 20;  
else if (option == 4) amt = 10;  
else printf("Error: unknown option.\n");
```

# while Loops

- while loops repeatedly execute a statement until a condition is not met

```
while (condition)
    statement
```

```
// Compute 9! (the factorial of 9)
int i = 1, fact = 1;

// multiply the numbers from 1 to 9
while (i < 10) { // while loops check the condition first
    fact *= i;
    i++;
}
```



# do/while Loops

- **do/while** loops are like while loops but the condition is checked only after the statement is executed once:

```
do
    statement
while (condition);
```

← The condition is followed by a semi-colon.

```
// Query user to guess a number and check it against the correct number.
#define MAXGUESSES 3
#define CORRECTNUM 7
int guess, numGuesses = 0;
do {
    printf("Guess a number between 0 and 9. You have %d more guesses.\n",
           (MAXGUESSES-numGuesses));
    scanf("%d", &guess);    // read user input
    numGuesses++;
} while ( (numGuesses < MAXGUESSES) & (guess != CORRECTNUM) );
// do loop checks the condition after the first iteration
if (guess == CORRECTNUM)
    printf("You guessed the correct number!\n");
```

# for Loops

- **for** loops, like while and do/while, repeatedly execute a statement until a condition is not satisfied.
- However, **for** loops add *support* for a *loop variable*, which typically keeps track of the number of loop executions.
- The general format of the for loop is

```
for (initialization; condition; loop operation)
    statement
```

- The **initialization** code executes only once, before the for loop begins.
- The **condition** is tested at the beginning of each iteration of the loop. If not TRUE, the loop exits.
- The **loop operation** executes at the end of each iteration.

```
// Compute 9!
int i;    // loop variable
int fact = 1;

for (i=1; i<10; i++)
    fact *= i;
```

# for Loops

- A **for** loop could be expressed equivalently, but less conveniently, as

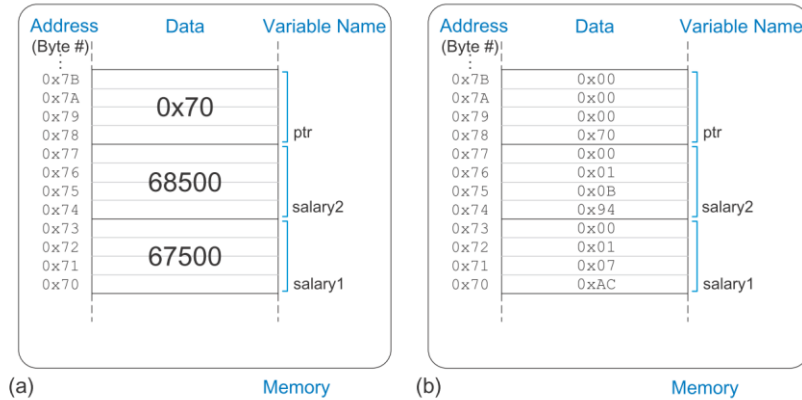
```
initialization;  
while (condition) {  
    statement  
    loop operation;  
}
```

# More data types: pointers

- A **pointer** is the address of a variable.

```
// Example pointer manipulations
int salary1, salary2; // 32-bit numbers
int *ptr;             // a pointer specifying the address of an int variable

salary1 = 67500;      // salary1 = $67,500 = 0x000107AC
ptr = &salary1;       // ptr = 0x0070, the address of salary1
salary2 = *ptr + 1000; /* dereference ptr to give the contents of address 70 = $67,500,
                       then add $1,000 and set salary2 to $68,500 */
```



**Figure eC.3** Contents of memory after C Code Example eC.18 executes shown (a) by value and (b) by byte using little-endian memory

## More data types: pointers

- In a variable declaration, a star (\*) before a variable name indicates that the variable is a pointer to the declared type.

```
int *ptr;           // a pointer specifying the address of an int variable
```

- In using a pointer variable, the \* operator *dereferences* a pointer, returning the value stored at the indicated memory address contained in the pointer.

```
salary2 = *ptr + 1000; /* dereference ptr to give the contents of address 70 = $67,500,  
                        then add $1,000 and set salary2 to $68,500 */
```

- The & operator is pronounced “address of,” and it produces the memory address of the variable being referenced.

```
ptr = &salary1;    // ptr = 0x0070, the address of salary1
```

- Dereferencing a pointer to a non-existent memory location or an address outside of the range accessible by the program will usually cause a program to crash.
- The crash is often called a *segmentation fault*.

## More data types: pointers

- Pointers are particularly useful when a function needs to modify a variable, instead of just returning a value.
- Functions can't modify their inputs directly, but we can make the input a pointer to the variable.
- This is called passing an input variable *by reference* instead of *by value*.
  
- A pointer to address 0 is called a null pointer and indicates that the pointer is not actually pointing to meaningful data. It is written as NULL in a program (with NULL defined in <stddef.h>).

# More data types: pointers

```
// Quadruple the value pointed to by a
#include <stdio.h>

void quadruple(int *a)
{
    *a = *a * 4;
}

int main(void)
{
    int x = 5;

    printf("x before: %d\n", x);
    quadruple(&x);
    printf("x after: %d\n", x);
    return 0;
}
```

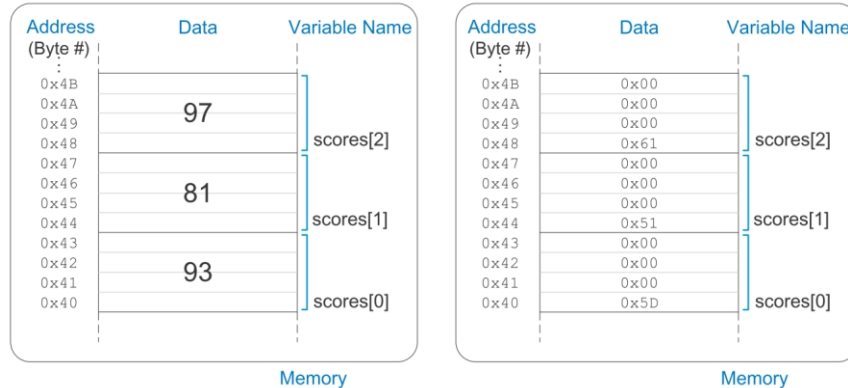
## Console Output

```
x before: 5
x after: 20
```

# More data types: arrays

- An array is a group of similar variables stored in consecutive addresses in memory.
- The elements are numbered from 0 to N-1, where N is the length of the array.

```
long scores[3]; // array of three 4-byte numbers
```



- In C, the *array variable*, in this case `scores`, is a *pointer* to the 1<sup>st</sup> element.
- It is the programmer's responsibility not to access elements beyond the end of the array.



# More data types: arrays

- The elements of an array can be initialized either at declaration using curly braces {},

```
long scores[3]={93, 81, 97}; // scores[0]=93; scores[1]=81; scores[2]=97;
```

- or individually in the body of the code,

```
long scores[3];  
scores[0] = 93;  
scores[1] = 81;  
scores[2] = 97;
```

- In the first case, if there are fewer initializers than the number specified, the missing elements will be zero.
- Each element of an array is accessed using brackets [].

```
// User enters 3 student scores into an array  
long scores[3];  
int i, entered;  
  
printf("Please enter the student's 3 scores.\n");  
for (i=0; i<3; i++) {  
    printf("Enter a score and press enter.\n");  
    scanf("%d", &entered);  
    scores[i] = entered;  
}  
printf("Scores: %d %d %d\n", scores[0], scores[1], scores[2]);
```

# More data types: arrays

- When an array is declared, the length must be constant so that the compiler can allocate the proper amount of memory.
- However, when the array is passed to a function as an input argument, the length need not be defined because the function only needs to know the address of the beginning of the array.

```
// Initialize a 5-element array, compute the mean, and print the result.
#include <stdio.h>

// Returns the mean value of an array (arr) of length len
float getMean(int arr[], int len) {
    int i;
    float mean, total = 0;

    for (i=0; i < len; i++)
        total += arr[i];

    mean = total / len;
    return mean;
}
```

## More data types: arrays

```
int main(void) {
    int data[4] = {78, 14, 99, 27};
    float avg;

    avg = getMean(data, 4);

    printf("The average value is: %f.\n", avg);
}
```

### Console Output

The average value is: 54.500000.

- An array argument is equivalent to a pointer to the beginning of the array. Thus, `getMean` could also have been declared as


```
float getMean(int *arr, int len);
```

- Although functionally equivalent, `datatype[]` is the preferred method for passing arrays as input arguments because it more clearly indicates that the argument is an array.

## More data types: arrays

- A function is limited to a single output, i.e., return variable. However, by receiving an array as an input argument, a function can essentially output more than a single value by changing the array itself.

```
// Sort the elements of the array vals of length len from lowest to highest
void sort(int vals[], int len)
{
    int i, j, temp;
    for (i=0; i<len; i++) {
        for (j=i+1; j<len; j++) {
            if (vals[i] > vals[j]) {
                temp = vals[i];
                vals[i] = vals[j];
                vals[j] = temp;
            }
        }
    }
}
```



```
void sort(int *vals, int len);
void sort(int vals[], int len);
void sort(int vals[100], int len);
```

# Number of elements of an array

- In the function where the array is declared, the number of elements in the array can be found from:

```
sizeof(array) / sizeof(arrayElement)
```

- E.g.,

```
int a[10];  
sizeof(a)/sizeof(int) is 10;
```

- This is useful to determine the number of elements when the array size is deduced by the initialization

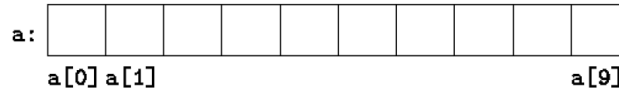
```
int a[]= {1, 2, 100, ..., 5, 2};
```

- Note however that in a function with a parameter `array[]`, `sizeof(array)` is just the size of the pointer to the array, because arrays are passed by reference and the number of elements is unknown.

# Arrays and Pointers

- In C, there is a strong relationship between pointers and arrays.
- Any operation that can be achieved by array subscripting can also be done with pointers.
- The declaration  

```
int a[10];
```
- defines an array of size 10, that is, a block of 10 consecutive objects named  $a[0]$ ,  $a[1]$ , ...,  $a[9]$

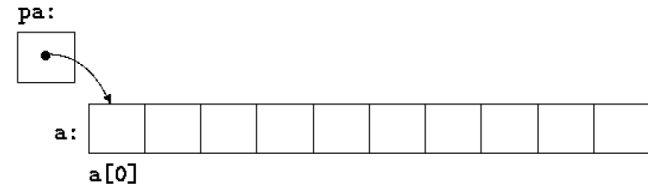


- If  $pa$  is a pointer to an integer, declared as  

```
int *pa;
```
- then the assignment  

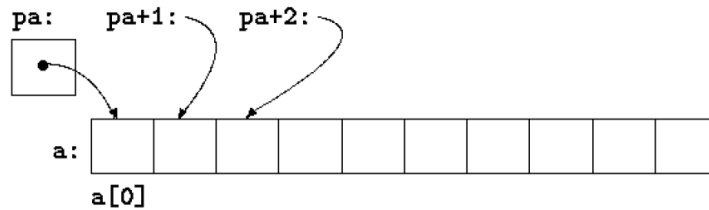
```
pa = &a[0];
```
- sets  $pa$  to point to element zero of  $a$ ; that is,  $pa$  contains the address of  $a[0]$ .  

```
x = *pa;
```
- will copy the contents of  $a[0]$  into  $x$ .



# Arrays and Pointers

- If  $pa$  points to a particular element of an array, then by definition  $pa+1$  points to the next element,  $pa+i$  points  $i$  elements after  $pa$ , and  $pa-i$  points  $i$  elements before. Thus, if  $pa$  points to  $a[0]$ ,
- $*(pa+1)$
- refers to the contents of  $a[1]$ ,  $pa+i$  is the address of  $a[i]$ , and  $*(pa+i)$  is the contents of  $a[i]$ .



- These remarks are true regardless of the type or size of the variables in the array  $a$ .
- The meaning of “adding 1 to a pointer” is that  $pa+1$  points to the next object, and  $pa+i$  points to the  $i$ -th object beyond  $pa$ .

# Arrays and Pointers

- After the assignment  
$$pa = \&a[0];$$
- $pa$  and  $a$  have identical values.
- Since the name of an array is a synonym for the location of the initial element, the assignment  $pa = \&a[0]$  can also be written as  
$$pa = a;$$
- In evaluating  $a[i]$ , C converts it to  $*(a+i)$  immediately; the two forms are equivalent.
- Moreover,  $pa[i]$  is identical to  $*(pa+i)$ .
  
- There is one difference between an array name and a pointer that must be kept in mind:
  - A pointer is a variable, so  $pa=a$  and  $pa++$  are legal.
  - But an array name is not a variable; constructions like  $a=pa$  and  $a++$  are illegal.
  
- Note that the expression  $*p++$  is parsed as  $*(p++)$ , and not as  $(*p)++$ .



# Arrays and Pointers

- When an array name is passed to a function, what is passed is the location of the initial element.
- Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.

```
/* strlen: return length of string s */
int strlen(char *s)
{
    int n;

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

- As formal parameters in a function definition,  
`char s[]` and `char *s`  
are equivalent.

# More data types: arrays

- Arrays may have multiple dimensions.

```
// Initialize 2-D array at declaration
int grades[10][8] = { {100, 107, 99, 101, 100, 104, 109, 117},
                    {103, 101, 94, 101, 102, 106, 105, 110},
                    {101, 102, 92, 101, 100, 107, 109, 110},
                    {114, 106, 95, 101, 100, 102, 102, 100},
                    {98, 105, 97, 101, 103, 104, 109, 109},
                    {105, 103, 99, 101, 105, 104, 101, 105},
                    {103, 101, 100, 101, 108, 105, 109, 100},
                    {100, 102, 102, 101, 102, 101, 105, 102},
                    {102, 106, 110, 101, 100, 102, 120, 103},
                    {99, 107, 98, 101, 109, 104, 110, 108} };
```

- Multi-dimensional arrays used as input arguments to a function must define all but the first dimension.

```
void print2dArray(int arr[10][8]);
void print2dArray(int arr[][8]);
```

## More data types: arrays

```
#include <stdio.h>

// Print the contents of a 10×8 array
void print2dArray(int arr[10][8])
{
    int i, j;

    for (i=0; i<10; i++) {           // for each of the 10 students
        printf("Row %d\n", i);
        for (j=0; j<8; j++) {
            printf("%d ", arr[i][j]); // print scores for all 8 problem sets
        }
        printf("\n");
    }
}

// Calculate the mean score of a 10×8 array
float getMean(int arr[10][8])
{
    int i, j;
    float mean, total = 0;

    // get the mean value across a 2D array
    for (i=0; i<10; i++) {
```

## More data types: arrays

```
    for (j=0; j<8; j++) {
        total += arr[i][j];      // sum array values
    }
}
mean = total/(10*8);
printf("Mean is: %f\n", mean);
return mean;
}
```

- Note that because an array is represented by a pointer to the initial element, C cannot copy or compare arrays using the = or == operators.
- Instead, you must use a loop to copy or compare each element one at a time.

## More data types: characters

- A character (char) is an 8-bit variable.
- It can be viewed either as a two's complement number between  $-128$  and  $127$  or as an ASCII code for a letter, digit, or symbol.
- ASCII characters can be specified as a numeric value (in decimal, hexadecimal, etc.) or as a printable character enclosed in single quotes.
  - The letter A has the ASCII code  $0x41$ ,  $B=0x42$ , etc. Thus, 'A' + 3 is  $0x44$ , or 'D'.

# More data types: characters

**Table eC.4** Special characters

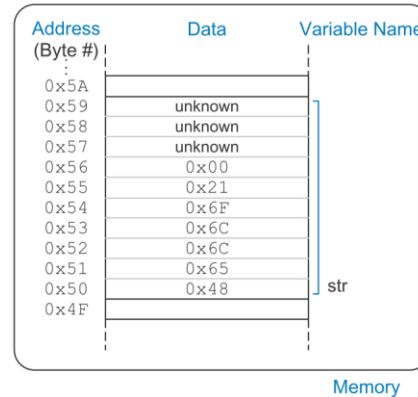
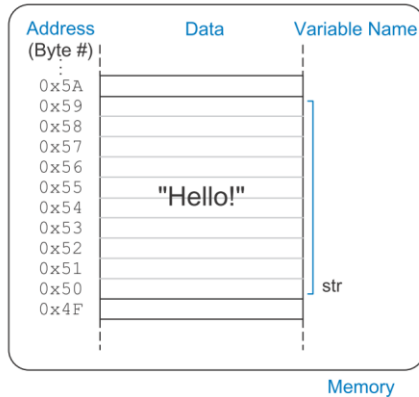
Special Character	Hexadecimal Encoding	Description
<code>\r</code>	0x0D	carriage return
<code>\n</code>	0x0A	new line
<code>\t</code>	0x09	tab
<code>\0</code>	0x00	terminates a string
<code>\\</code>	0x5C	backslash
<code>\"</code>	0x22	double quote
<code>\'</code>	0x27	single quote
<code>\a</code>	0x07	bell

Windows text files use `\r\n` to represent end-of-line while UNIX-based systems use `\n`, which can cause nasty bugs when moving text files between systems.

# More data types: strings

- A string is an *array of characters* used to store a piece of text of bounded but variable length.
- Each character is a byte representing the ASCII code for that letter, number, or symbol.
- The *size of the array* determines the *maximum length* of the string, but the *actual length* of the string could be *shorter*.
- In C, the length of the string is determined by looking for the null terminator at the end of the string.

```
char greeting[10] = "Hello!";
```



**Figure eC.5** The string "Hello!" stored in memory

# More data types: strings

- an alternate declaration of the string *greeting*:

```
char *greeting = "Hello!";  
printf("greeting: %s", greeting);
```

## Console Output

```
greeting: Hello!
```

- Unlike primitive variables, a string cannot be set equal to another string using the equals operator, =.
- Each element of the character array must be copied from the source string to the target string.

```
// Copy the source string, src, to the destination string, dst  
void strcpy(char *dst, char *src)  
{  
  
    int i = 0;  
  
    do {  
        dst[i] = src[i];        // copy characters one byte at a time  
    } while (src[i++]);        // until the null terminator is found  
}
```



# More data types: structures

- In C, structures are used to store a collection of data of various types.
- The general format of a structure declaration is

```
struct name {  
    type1 element1;  
    type2 element2;  
    ...  
};
```

- where *struct* is a keyword indicating that it is a structure, *name* is the structure tag name, and *element1* and *element2* are members of the structure.

```
struct contact {  
    char name[30];  
    int phone;  
    float height; // in meters  
};  
  
struct contact c1;  
  
strcpy(c1.name, "Ben Bitdiddle");  
c1.phone = 7226993;  
c1.height = 1.82;
```

# More data types: structures

- Just like built-in C types, you can create arrays of structures and pointers to structures.

```
struct contact classlist[200];  
classlist[0].phone = 9642025;
```

- It is common to use pointers to structures.
- C provides the *member access operator* -> to dereference a pointer to a structure and access a member of the structure.

```
struct contact *cptr;  
cptr = &classlist[42];  
cptr->height = 1.9; // equivalent to: (*cptr).height = 1.9;
```

# More data types: structures

The table lists the precedence and associativity of C operators. Operators are listed top to bottom, in descending precedence.

Taken from [https://en.cppreference.com/w/c/language/operator\\_precedence](https://en.cppreference.com/w/c/language/operator_precedence)

Precedence	Operator	Description	Associativity	
1	++ --	Suffix/postfix increment and decrement	Left-to-right	
	()	Function call		
	[]	Array subscripting		
	.	Structure and union member access		
	->	Structure and union member access through pointer		
	(type){list}	Compound literal(C99)		
2	++ --	Prefix increment and decrement <sup>[note 1]</sup>	Right-to-left	
	+ -	Unary plus and minus		
	! ~	Logical NOT and bitwise NOT		
	(type)	Cast		
	*	Indirection (dereference)		
	&	Address-of		
	sizeof	Size-of <sup>[note 2]</sup>		
	_Alignof	Alignment requirement(C11)		
3	* / %	Multiplication, division, and remainder	Left-to-right	
4	+ -	Addition and subtraction		
5	<< >>	Bitwise left shift and right shift		
6	< <=	For relational operators < and ≤ respectively		
	> >=	For relational operators > and ≥ respectively		
7	== !=	For relational = and ≠ respectively		
8	&	Bitwise AND		
9	^	Bitwise XOR (exclusive or)		
10		Bitwise OR (inclusive or)		
11	&&	Logical AND		
12		Logical OR		
13	?:	Ternary conditional <sup>[note 3]</sup>		Right-to-left
14	=	Simple assignment		Right-to-left
	+= -=	Assignment by sum and difference		
	*= /= %=	Assignment by product, quotient, and remainder		
	<<= >>=	Assignment by bitwise left shift and right shift		
	&= ^=  =	Assignment by bitwise AND, XOR, and OR		
15	,	Comma	Left-to-right	

# More data types: structures

- Structures can be passed as function inputs or outputs by value or by reference.
- Passing by value requires the compiler to copy the entire structure into memory for the function.
- Passing by reference involves passing a pointer to the structure, which is more efficient.

```
struct contact stretchByValue(struct contact c)
{
    c.height += 0.02;
    return c;
}
void stretchByReference(struct contact *cptr)
{
    cptr->height += 0.02;
}

int main(void)
{
    struct contact George;

    George.height = 1.4; // poor fellow has been stooped over
    George = stretchByValue(George); // stretch for the stars
    stretchByReference(&George);    // and stretch some more
}
```

# More data types: typedef

- C also allows you to define your own names for data types using the typedef statement.

```
typedef struct contact {  
    char name[30];  
    int phone;  
    float height; // in meters  
} contact;      // defines contact as shorthand for "struct contact"  
  
contact c1;     // now we can declare the variable as type contact
```

```
typedef unsigned char byte;  
typedef char bool;  
#define TRUE 1  
#define FALSE 0  
  
byte pos = 0x45;  
bool loveC = TRUE;
```

```
typedef double vector[3];  
typedef double matrix[3][3];  
  
vector a = {4.5, 2.3, 7.0};  
matrix b = {{3.3, 4.7, 9.2}, {2.5, 4, 9}, {3.1, 99.2, 88}};
```

# Dynamic memory allocation

- In all the examples thus far, variables have been declared *statically*: their size is known at compile time.
- This can be problematic for arrays and strings of variable size because the array must be declared large enough to accommodate the largest size the program will ever see.
- An alternative is to *dynamically* allocate memory at run time when the actual size is known.
- The **malloc** function from **stdlib.h** allocates a block of memory of a specified size and returns a pointer to it. If not enough memory is available, it returns a NULL pointer instead.

```
// dynamically allocate 20 bytes of memory  
short *data = malloc(10*sizeof(short));
```

- The **free** function de-allocates the memory so that it could later be used for other purposes.
- Failing to de-allocate dynamically allocated data is called a *memory leak* and should be avoided.

# Dynamic memory allocation

```
// Dynamically allocate and de-allocate an array using malloc and free
#include <stdlib.h>

// Insert getMean function from C Code Example eC.24.

int main(void) {
    int len, i;
    int *nums;

    printf("How many numbers would you like to enter?   ");
    scanf("%d", &len);
    nums = malloc(len*sizeof(int));
    if (nums == NULL) printf("ERROR: out of memory.\n");
    else {
        for (i=0; i<len; i++) {
            printf("Enter number:   ");
            scanf("%d", &nums[i]);
        }
        printf("The average is %f\n", getMean(nums, len));
    }
    free(nums);
}
```

# Example Linked lists

- A *linked list* is a common data structure used to store a variable number of elements.
- Each element in the list is a structure containing one or more data fields and a link to the next element.
- The first element in the list is called the *head*.
  
- The code in the following slides describes a linked list for storing computer user accounts to accommodate a variable number of users.
- Each user has a user name, a password, a unique user identification number (UID), and a field indicating whether they have administrator privileges.
- Each element of the list is of type *userL*, containing all of this user information along with a link to the next element in the list.
- A pointer to the head of the list is stored in a global variable called *users*, and is initially set to NULL to indicate that there are no users.
- The program defines functions to insert, delete, and find a user and to count the number of users.



# Example Linked lists

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

typedef struct userL {
    char uname[80];    // user name
    char passwd[80];  // password
    int uid;           // user identification number
    int admin;         // 1 indicates administrator privileges
    struct userL *next;
} userL;

userL *users = NULL;

void insertUser(char *uname, char *passwd, int uid, int admin) {
    userL *newUser;

    newUser = malloc(sizeof(userL)); // create space for new user
    strcpy(newUser->uname, uname);   // copy values into user fields
    strcpy(newUser->passwd, passwd);
    newUser->uid = uid;
    newUser->admin = admin;
    newUser->next = users;           // insert at start of linked list
    users = newUser;
}
```

# Example Linked lists

```
void deleteUser(int uid) { // delete first user with given uid
    userL *cur = users;
    userL *prev = NULL;

    while (cur != NULL) {
        if (cur->uid == uid) { // found the user to delete
            if (prev == NULL) users = cur->next;
            else prev->next = cur->next;
            free(cur);
            return; // done
        }
        prev = cur; // otherwise, keep scanning through list
        cur = cur->next;
    }
}

userL *findUser(int uid) {
    userL *cur = users;

    while (cur != NULL) {
        if (cur->uid == uid) return cur;
        else cur = cur->next;
    }
    return NULL;
}
```

# Example Linked lists

```
int numUsers(void) {
    userL *cur = users;
    int count = 0;

    while (cur != NULL) {
        count++;
        cur = cur->next;
    }
    return count;
}
```

# Standard libraries

- Programmers commonly use a variety of standard functions, such as printing and trigonometric operations.
- To save each programmer from having to write these functions from scratch, C provides **libraries** of frequently used functions.
- Each library has a **header file** and an associated **object file**, which is a partially compiled C file.
- The header file holds variable declarations, defined types, and function prototypes.
- The **object file** contains the functions themselves and is **linked at compile-time** to create the executable.
- Because the library function calls are already compiled into an object file, compile time is reduced.

# Standard libraries

C Library Header File	Description
<code>stdio.h</code>	<b>Standard input/output library.</b> Includes functions for printing or reading to/from the screen or a file ( <code>printf</code> , <code>fprintf</code> and <code>scanf</code> , <code>fscanf</code> ) and to open and close files ( <code>fopen</code> and <code>fclose</code> ).
<code>stdlib.h</code>	<b>Standard library.</b> Includes functions for random number generation ( <code>rand</code> and <code>srand</code> ), for dynamically allocating or freeing memory ( <code>malloc</code> and <code>free</code> ), terminating the program early ( <code>exit</code> ), and for conversion between strings and numbers ( <code>atoi</code> , <code>atol</code> , and <code>atof</code> ).
<code>math.h</code>	<b>Math library.</b> Includes standard math functions such as <code>sin</code> , <code>cos</code> , <code>asin</code> , <code>acos</code> , <code>sqrt</code> , <code>log</code> , <code>log10</code> , <code>exp</code> , <code>floor</code> , and <code>ceil</code> .
<code>string.h</code>	<b>String library.</b> Includes functions to compare, copy, concatenate, and determine the length of strings.

# Printf

- The *print formatted* statement **printf** displays text to the console.
- Its required input argument is a string enclosed in quotes " ".
- The string contains text and optional commands to print variables.
- Variables to be printed are listed after the string and are printed using format codes.

```
// Simple print function
#include <stdio.h>

int num = 42;

int main(void) {
    printf("The answer is %d.\n", num);
}
```

## Console Output:

The answer is 42.

# Printf

**Table eC.6** printf format codes for printing variables

Code	Format
%d	Decimal
%u	Unsigned decimal
%x	Hexadecimal
%o	Octal
%f	Floating point number (float or double)
%e	Floating point number (float or double) in scientific notation (e.g., 1.56e7)
%c	Character (char)
%s	String (null-terminated array of characters)

# Printf

- Floating point formats (floats and doubles) default to printing six digits after the decimal point.
- To change the precision, replace `%f` with `%w.df`, where `w` is the minimum width of the number, and `d` is the number of decimal places to print.
- Note that the decimal point is included in the width count.

```
// Print floating point numbers with different formats
float pi = 3.14159, e = 2.7182, c = 2.998e8;
printf("pi = %4.2f\n e = %8.3f\n c = %5.3f\n", pi, e, c);
```

## Console Output:

```
pi = 3.14
e = 2.718
c = 299800000.000
```



# Printf

- Because % and \ are used in print formatting, to print these characters:

```
// How to print % and \ to the console  
printf("Here are some special characters: %% \\ \n");
```

## **Console Output:**

```
Here are some special characters: % \
```

# scanf

- The **scanf** function reads text typed on the keyboard. It uses format codes in the same way as **printf**.
- When the **scanf** function is encountered, the program waits until the user types a value.
- The arguments to **scanf** are a string (indicating one or more format codes) and pointers to the variables where the results should be stored.

```
// Read variables from the command line
#include <stdio.h>

int main(void)
{
    int a;
    char str[80];
    float f;

    printf("Enter an integer.\n");
    scanf("%d", &a);
    printf("Enter a floating point number.\n");
    scanf("%f", &f);
    printf("Enter a string.\n");
    scanf("%s", str);    // note no & needed: str is a pointer
}
```

# File manipulation

- Many programs need to read and write files, either to manipulate data already stored in a file or to log large amounts of information.
- In C, the file must first be opened with the **fopen** function.
- It can then be read or written with **fscanf** or **fprintf** in a way analogous to reading and writing to the console.
- Finally, it should be closed with the **fclose** command.
- The **fopen** function takes as arguments the file name and a print mode.
  - It returns a file pointer of type **FILE\***.
  - If fopen is unable to open the file, it returns **NULL**.
  - The modes are:
    - "w": Write to a file. If the file exists, it is overwritten.
    - "r": Read from a file.
    - "a": Append to the end of an existing file. If the file doesn't exist, it is created.

# File manipulation

```
// Write "Testing file write." to result.txt
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE *fptr;

    if ((fptr = fopen("result.txt", "w")) == NULL) {
        printf("Unable to open result.txt for writing.\n");
        exit(1); // exit the program indicating unsuccessful execution
    }
    fprintf(fptr, "Testing file write.\n");
    fclose(fptr);
}
```

# File manipulation

```
#include <stdio.h>

int main(void)
{
    FILE *fptr;
    int data;

    // read in data from input file
    if ((fptr = fopen("data.txt", "r")) == NULL) {
        printf("Unable to read data.txt\n");
        exit(1);
    }

    while (!feof(fptr)) { // check that the end of the file hasn't been reached
        fscanf(fptr, "%d", &data);
        printf("Read data: %d\n", data);
    }

    fclose(fptr);
}
```

## data.txt

25 32 14 89

## Console Output:

Read data: 25

Read data: 32

Read data: 14

Read data: 89

## Other Handy stdio Functions

- The `sprintf` function prints characters into a string, and `sscanf` reads variables from a string.
- The `fgetc` function reads a single character from a file, while `fgets` reads a complete line into a string.
- `fscanf` is rather limited in its ability to read and parse complex files, so it is often easier to `fgets` one line at a time and then digest that line using `sscanf` or with a loop that inspects characters one at a time using `fgetc`.

## Other Handy stdio Functions

Reading and writing binary files is pretty much the same as any other file, the only difference is how you open it:

```
unsigned char buffer[10];  
FILE *ptr;  
ptr = fopen("test.bin", "rb"); // r for read, b for binary  
fread(buffer, sizeof(buffer), 1, ptr); // read 10 bytes to our buffer
```

Writing to a file is pretty much the same, with the exception that you're using fwrite() instead of fread():

```
FILE *write_ptr;  
write_ptr = fopen("test.bin", "wb"); // w for write, b for binary  
fwrite(buffer, sizeof(buffer), 1, write_ptr); // write 10 bytes from our buffer
```

# stdlib

- The standard library **stdlib.h** provides general purpose functions including random number generation (**rand** and **srand**), dynamic memory allocation (**malloc** and **free**), exiting the program early (**exit**), and number format conversions.
- To use these functions, add the following line at the top of the C file:

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



# rand and srand

- **rand** returns a pseudo-random integer.
- Pseudo-random numbers have the statistics of random numbers but follow a deterministic pattern starting with an initial value called the **seed**.
- To convert the number to a particular range, use the modulo operator (%)

```
#include <stdlib.h>
int x, y;

x = rand();          // x = a random integer
y = rand() % 10;    // y = a random number from 0 to 9
printf("x = %d, y = %d\n", x, y);
```

## Console Output:

```
x = 1481765933, y = 3
```

# rand and srand

- The values generated by the previous program will be the same each time the program runs.
- We can create a different sequence of random numbers at each run by changing the **seed**.
- This is done by calling the **srand** function, which takes the seed as its input argument.

```
// Produce a different random number each run
#include <stdlib.h>
#include <time.h>    // needed to call time()

int main(void)
{
    int x;

    srand(time(NULL));    // seed the random number generator
    x = rand() % 10;      // random number from 0 to 9
    printf("x = %d\n", x);
}
```

- For historical reasons, the time function usually returns the current time in seconds relative to January 1, 1970 00:00 UTC. UTC stands for Coordinated Universal Time, which is the same as Greenwich Mean Time (GMT).

# exit

- The **exit** function terminates a program early.
- It takes a single argument that is returned to the operating system to indicate the reason for termination.
- **0** indicates **normal completion**, while **nonzero** conveys an **error condition**.

# Format Conversion: atoi, atol, atof

- Functions for converting strings to integers, long integers, or doubles: **atoi**, **atol**, **atof**, respectively.

```
// Convert ASCII strings to ints, longs, and floats
#include <stdlib.h>

int main(void)
{
    int x;
    long int y;
    double z;

    x = atoi("42");
    y = atol("833");
    z = atof("3.822");

    printf("x = %d\t y = %d\t z = %f\n", x, y, z);
}
```

## Console Output:

```
x = 42   y = 833   z = 3.822000
```

# math.h

- The library math.h provides math functions, such as trigonometry functions, square root, and logs.
- To use math functions, use `#include <math.h>`

```
// Example math functions
#include <stdio.h>
#include <math.h>

int main(void) {
    float a, b, c, d, e, f, g, h;

    a = cos(0);           // 1, note: the input argument is in radians
    b = 2 * acos(0);      // pi (acos means arc cosine)
    c = sqrt(144);        // 12
    d = exp(2);           // e^2 = 7.389056,
    e = log(7.389056);    // 2 (natural logarithm, base e)
    f = log10(1000);      // 3 (log base 10)
    g = floor(178.567);   // 178, rounds to next lowest whole number
    h = pow(2, 10);       // computes 2 raised to the 10th power

    printf("a = %.0f, b = %f, c = %.0f, d = %.0f, e = %.2f, f = %.0f, g = %.2f, h = %.2f\n",
           a, b, c, d, e, f, g, h);
}
```

## Console Output:

```
a = 1, b = 3.141593, c = 12, d = 7, e = 2.00, f = 3, g = 178.00, h = 1024.00
```

# string.h

- The string library string.h provides commonly used string manipulation functions.

```
// copy src into dst and return dst
char *strcpy(char *dst, char *src);

// concatenate (append) src to the end of dst and return dst
char *strcat(char *dst, char *src);

// compare two strings. Return 0 if equal, nonzero otherwise
int strcmp(char *s1, char *s2);

// return the length of str, not including the null termination
int strlen(char *str);
```

# Static Variables

- The keyword **static** can be applied to both external (global) and internal (local) variables.
- The static declaration, applied to an *external variable* or function, limits the scope of that object to the rest of the source file being compiled. External static thus provides a way to hide names to other .c files.
- Internal static variables are local to a particular function just as automatic variables are, but unlike automatics, they remain in existence rather than coming and going each time the function is activated. This means that internal static variables provide private, permanent storage within a single function.

# Compiler and command line options

- Multiple C files are compiled into a single executable by listing all file names on the compile line:

```
gcc main.c file2.c file3.c
```

Compiler Option	Description	Example
-o outfile	specifies output file name	gcc -o hello hello.c
-S	create assembly language output file (not executable)	gcc -S hello.c this produces hello.s
-v	verbose mode – prints the compiler results and processes as compilation completes	gcc -v hello.c
-Olevel	specify the optimization level (level is typically 0 through 3), producing faster and/or smaller code at the expense of longer compile time	gcc -O3 hello.c
--version	list the version of the compiler	gcc --version
--help	list all command line options	gcc --help
-Wall	print all warnings	gcc -Wall hello.c



# Command Line Arguments

- Like other functions, main can also take input variables.
- However, unlike other functions, these arguments are specified at the command line.

```
// Print command line arguments
#include <stdio.h>

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

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

```
gcc -o testargs testargs.c
./testargs arg1 25 lastarg!
```

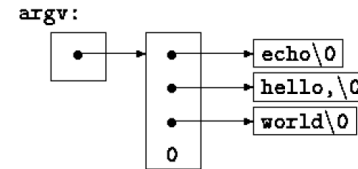
## Console Output:

```
argv[0] = ./testargs
argv[1] = arg1
argv[2] = 25
argv[3] = lastarg!
```

- **argc** stands for *argument count*, and it denotes the number of arguments on the command line.
- **argv** stands for argument vector, and it is an array of the strings found on the command line.
  - Note that the executable name is counted as the 1<sup>st</sup> argument.

# Command-line Arguments

- In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing.
- When `main` is called, it is called with two arguments.
- The first (conventionally called **argc**, for *argument count*) is the number of command-line arguments the program was invoked with; the second (**argv**, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string.
- Multiple levels of pointers are used to manipulate these character strings.
  
- E.g., the program `echo`, which echoes its command-line arguments on a single line, separated by blanks: `echo hello, world` prints the output `hello, world`
- By convention, `argv[0]` is the name by which the program was invoked, so `argc` is at least 1.
- In the example above, `argc` is 3, and `argv[0]`, `argv[1]`, and `argv[2]` are "echo", "hello," and "world" respectively.
- The standard requires that `argv[argc]` be a null pointer.



# Command-line Arguments

- The first version of echo treats `argv` as an array of character pointers:

```
#include <stdio.h>

/* echo command-line arguments; 1st version */
int main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

# Command-line Arguments

- Since `argv` is a pointer to an array of pointers, we can manipulate the pointer rather than index the array. This next variant is based on incrementing `argv`, which is a pointer to pointer to char, while `argc` is counted down:

```
#include <stdio.h>

/* echo command-line arguments; 2nd version */
int main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

# Common mistakes

## C Code Mistake eC.1 MISSING & IN scanf

### Erroneous Code

```
int a;  
printf("Enter an integer:\t");  
scanf("%d", a); // missing & before a
```

### Corrected Code:

```
int a;  
printf("Enter an integer:\t");  
scanf("%d", &a);
```

## C Code Mistake eC.2 USING = INSTEAD OF == FOR COMPARISON

### Erroneous Code

```
if (x = 1) // always evaluates as TRUE  
    printf("Found!\n");
```

### Corrected Code

```
if (x == 1)  
    printf("Found!\n");
```

## C Code Mistake eC.3 INDEXING PAST LAST ELEMENT OF ARRAY

### Erroneous Code

```
int array[10];  
array[10] = 42; // index is 0-9
```

### Corrected Code

```
int array[10];  
array[9] = 42;
```

# Common mistakes

## C Code Mistake eC.4 USING = IN #define STATEMENT

### Erroneous Code

```
// replaces NUM with "= 4" in code
#define NUM = 4
```

### Corrected Code

```
#define NUM 4
```

## C Code Mistake eC.5 USING AN UNINITIALIZED VARIABLE

### Erroneous Code

```
int i;
if (i == 10) // i is uninitialized
    ...
```

### Corrected Code

```
int i = 10;
if (i == 10)
    ...
```

## C Code Mistake eC.6 NOT INCLUDING PATH OF USER-CREATED HEADER FILES

### Erroneous Code

```
#include "myfile.h"
```

### Corrected Code

```
#include "othercode\myfile.h"
```

# Common mistakes

## C Code Mistake eC.7 USING LOGICAL OPERATORS (!, ||, &&) INSTEAD OF BITWISE (~, |, &)

### Erroneous Code

```
char x=!5;    // logical NOT: x = 0
char y=5||2;  // logical OR:  y = 1
char z=5&&2;  // logical AND: z = 1
```

### Corrected Code

```
char x=~5;    // bitwise NOT:  x = 0b11111010
char y=5|2;   // bitwise OR:   y = 0b00000111
char z=5&2;   // logical AND:  z = 0b00000000
```

## C Code Mistake eC.8 FORGETTING break IN A switch/case STATEMENT

### Erroneous Code

```
char x = 'd';
...
switch (x) {
    case 'u': direction = 1;
    case 'd': direction = 2;
    case 'l': direction = 3;
    case 'r': direction = 4;
    default: direction = 0;
}
// direction = 0
```

### Corrected Code

```
char x = 'd';
...
switch (x) {
    case 'u': direction = 1; break;
    case 'd': direction = 2; break;
    case 'l': direction = 3; break;
    case 'r': direction = 4; break;
    default: direction = 0;
}
// direction = 2
```

# Common mistakes

## C Code Mistake eC.9 MISSING CURLY BRACES {}

### Erroneous Code

```
if (ptr == NULL) // missing curly braces
    printf("Unable to open file.\n");
    exit(1);      // always executes
```

### Corrected Code

```
if (ptr == NULL) {
    printf("Unable to open file.\n");
    exit(1);
}
```

## C Code Mistake eC.10 USING A FUNCTION BEFORE IT IS DECLARED

### Erroneous Code

```
int main(void)
{
    test();
}

void test(void)
{...
}
```

### Corrected Code

```
void test(void)
{...
}

int main(void)
{
    test();
}
```



# Common mistakes

## C Code Mistake eC.11 DECLARING A LOCAL AND GLOBAL VARIABLE WITH THE SAME NAME

### Erroneous Code

```
int x = 5;    // global declaration of x
int test(void)
{
    int x = 3; // local declaration of x
    ...
}
```

### Corrected Code

```
int x = 5;    // global declaration of x
int test(void)
{
    int y = 3; // local variable is y
    ...
}
```

## C Code Mistake eC.12 TRYING TO INITIALIZE AN ARRAY WITH {} AFTER DECLARATION

### Erroneous Code

```
int scores[3];
scores = {93, 81, 97}; // won't compile
```

### Corrected Code

```
int scores[3] = {93, 81, 97};
```

# Common mistakes

## C Code Mistake eC.13 ASSIGNING ONE ARRAY TO ANOTHER USING =

### Erroneous Code

```
int scores[3] = {88, 79, 93};
int scores2[3];

scores2 = scores;
```

### Corrected Code

```
int scores[3] = {88, 79, 93};
int scores2[3];

for (i=0; i<3; i++)
    scores2[i] = scores[i];
```

## C Code Mistake eC.14 FORGETTING THE SEMI-COLON AFTER A do/while LOOP

### Erroneous Code

```
int num;
do {
    num = getNum();
} while (num < 100) // missing ;
```

### Corrected Code

```
int num;
do {
    num = getNum();
} while (num < 100);
```

# Common mistakes

## C Code Mistake eC.15 USING COMMAS INSTEAD OF SEMICOLONS IN for LOOP

### Erroneous Code

```
for (i=0, i < 200, i++)  
...
```

### Corrected Code

```
for (i=0; i < 200; i++)  
...
```

## C Code Mistake eC.16 INTEGER DIVISION INSTEAD OF FLOATING POINT DIVISION

### Erroneous Code

```
// integer (truncated) division occurs when  
// both arguments of division are integers  
float x = 9 / 4; // x = 2.0
```

### Corrected Code

```
// at least one of the arguments of  
// division must be a float to  
// perform floating point division  
float x = 9.0 / 4; // x = 2.25
```

## C Code Mistake eC.17 WRITING TO AN UNINITIALIZED POINTER

### Erroneous Code

```
int *y = 77;
```

### Corrected Code

```
int x, *y = &x;  
*y = 77;
```

# References

- Sarah Harris and David Harris “Digital Design and Computer Architecture. ARM Edition”, Morgan Kaufmann, 2015.
  - Appendix C
- Ritchie, Dennis M., Brian W. Kernighan, and Michael E. Lesk. The C programming language. Englewood Cliffs: Prentice Hall, 1988.
  - 5.3 Pointers and Arrays (pages 97-100)