

# Ateneo de Manila University

## C for Java Programmers



Department of Information Systems and  
Computer Science

S.Y. 2001-2002

<http://sysads.ateneo.net/wyu/>

[wyy@admu.edu.ph](mailto:wyy@admu.edu.ph)

## Section I

# C Programming Language

## C

- ★ high level programming language
- ★ low-level (bitwise)
- ★ small size and light weight
- ★ loose typing
- ★ structured programming language
- ★ pointer implementation

## History of C

- ★ 1969 - UNIX developed with DEC PDP-7 Assembly Language
- ★ 1970 - A new language "B" a second attempt
- ★ 1971 - A totally new language "C" a successor to "B"
- ★ 1973 - UNIX OS almost totally written in "C"

## Obfuscated C Code

★ International Obfuscated C Code Contest <http://reality.sgi.com/csp/iocc>

★ This is how NOT to program in C

```
#include <stdio.h>

main(t,_,a)
char *a;
{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86, 0, a+1 )+a)):1,t<_?main(t+1, _, a ):3,main ( -94, -27+t, a
)&&t == 2 ?_<13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<-72?main(,
t,"@n'+,#'/*{}w+/w#cdnr/+,{r/*de}+,/*{*,/w{%+,/w#q#n+,/#{1,+,/n{n+\\
,/+#n+,/#;#q#n+,/+k#;*,/'r : 'd*'3},{w+K w'K:'+}e#';dq#'l q#'+d'K#!/\
+k#;q#'r}eKK#w'r}eKK{nl}'/#;#q#n')}{#}w')}{nl}'/+#n';d}rw' i;# )}{n\
l}!/n{n#'; r{#w'r nc{nl}'/#{1,+ 'K {rw' iK{;[{nl}'/w#q#\
n'wk nw' iwk{KK{nl}!/w{% 'l##w# ' i; :{nl}'/*{q#'ld;r'}{nlwb!/*de}'c \
; ;{nl}'-{}rw}'/+,}##' * }#nc, ',#nw}'/+kd'+e}+;\
#'rdq#w! nr' / ' ) }+}{r1#'{n' ' )# }'+}##(!!/"
:t<-50?_==*a ?putchar(a[31]):main(-65,_,a+1):main((*a == '/')+t,_,a\
+1 ):0<t?main ( 2, 2 , "%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
i@bK'(q)-[w]*%n+r3#l,{:}\nuwloca-0;m .vpbks,fxntdCeghiry"),a+1);}
```

## Elements of a C Program

- ★ Preprocessor Commands
- ★ Type definitions
- ★ Function prototypes
- ★ Variables
- ★ Functions

## Fundamental Differences

- ★ strict vs loose typing
- ★ use of a preprocessor
- ★ fundamental datatypes
- ★ function definition and evaluation
- ★ pointers and references
- ★ memory management

## Fundamental Datatypes

- ★ Integer, Characters, Floating Point Numbers
- ★ Enumeration, Void, Array
- ★ Boolean (Java only)
- ★ Union and Structure (C only)

## Function Definition and Evaluation

```
type function_name (parameters)
{
    local variables

    C Statements

    return [ return value ]
}
```

- functions can exist without being members of class
- arguments in C are evaluated dependent on the system
- arguments can be passed by value or by reference
- optional return values can be specified
- parameters can be fixed or variable

## Elementary C Program

```
/* Sample program */

main()
{

    printf( "I Like C !\n" );
    exit ( 0 );

}
```

- `main()` does not have to be part of a class
- C requires a semicolon at the end of every statement
- Statements are grouped using `{` and `}`
- C programs are terminated by called the `exit()` or `return`

## Section II

# Memory Management

## Pointers and References

### ★ Pointer

- is a variable that contains the address of a particular memory location

### ★ Uses of pointers:

- a pointer can reference a variety of different values over the course of program execution
- a pointer will only reference a single value, but the type of that value is not known during run time
- the amount of memory to be used during the course of program execution cannot be determined at compile time

## Operations on Pointers

- ★ dereferencing a pointer ( \* )
- ★ dereferencing a member of a structure ( -> )
- ★ pointer addresses can be retrieved ( & )
- ★ pointers can be subscripted like arrays
- ★ pointers can be operated on by arithmetic operators

## Memory Management

- ★ in Java, memory management is left to the systems
  - implicit memory management
- ★ in C, memory management is a task of the programmer
  - explicit memory management
- ★ Possible faults in C that are unusual in Java:
  - value is used before it is initialized
  - failure to deallocate unused memory
  - using a value after it was deallocated
  - accessing memory outside the range of the allocated memory area

## Allocating and Deallocating Memory

★ `void *calloc(size_t nmemb, size_t size);`

- allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero.

★ `void *alloca( size_t size);`

- similar calling sequence to `malloc`
- allocates memory from the stack
- memory is deallocate automatically at the end of the function

★ `void *malloc(size_t size);`

- allocates *size* bytes and returns a pointer to the allocated memory. The memory is not cleared.

★ `void *realloc(void *ptr, size_t size);`

- changes the size of the memory block pointed to by *ptr* to *size* bytes. The

contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If *ptr* is NULL, the call is equivalent to **malloc(size)**; if *size* is equal to zero, the call is equivalent to **free(ptr)**. Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc()**, **calloc()** or **realloc()**.

```
★ void free(void *ptr);
```

- frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc()**, **calloc()** or **realloc()**. Otherwise, or if **free(ptr)** has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

## Section III

# Strings

## String

- ★ an array of characters
- ★ is a sequence of zero or more characters followed by a NULL (`\0`) character
- ★ string manipulation functions are defined in `string.h`

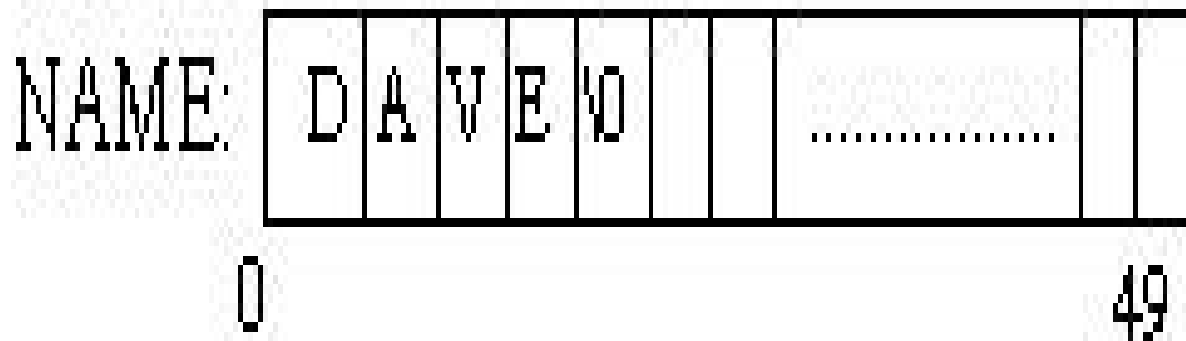


Figure 1: Representation of String

## String Manipulation

- `char *strcat (const char *dest, const char *src)`  
- copy one string into another
- `int strcmp(const char *string1, const char *string2)` - compare string1 and string2 to determine alphabetic order
- `char *strcpy(const char *string1, const char *string2)` - copy string2 to string1
- `char *strerror(int errnum)` - get error message corresponding to specified error number
- `int strlen(const char *string)` - determine the length of a string
- `char *strncat(const char *string1, char *string2, size_t n)` - append n characters from string2 to string1

- `int strncmp(const char *string1, char *string2, size_t n)` - compare first n characters of two strings
- `char *strncpy(const char *string1, const char *string2, size_t n)` - copy first n characters of string2 to string1
- `int strcasecmp(const char *s1, const char *s2)` - case insensitive version of `strcmp()`.
- `int strncasecmp(const char *s1, const char *s2, int n)` - case insensitive version of `strncmp()`.

## String Search

- `char *strchr(const char *string, int c)` - find first occurrence of character `c` in string
- `char *strrchr(const char *string, int c)` - find last occurrence of character `c` in string
- `char *strstr(const char *s1, const char *s2)` - locates the first occurrence of the string `s2` in string `s1`
- `char *strpbrk(const char *s1, const char *s2)` - returns a pointer to the first occurrence in string `s1` of any character from string `s2`, or a null pointer if no character from `s2` exists in `s1`
- `size_t strspn(const char *s1, const char *s2)` - returns the number of characters at the beginning of `s1` that match `s2`
- `size_t strcspn(const char *s1, const char *s2)` - returns the number of characters at the beginning of `s1` that do not match `s2`

- `char *strtok(char *s1, const char *s2)` - break the string pointed to by `s1` into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by `s2`
- `char *strtok_r(char *s1, const char *s2, char **lasts)` - has the same functionality as `strtok()` except that a pointer to a string placeholder `lasts` must be supplied by the caller



Copyright © 2000-2001 by William Emmanuel S. Yu. This material may be distributed only subject to the terms and conditions set forth in the Open Content License, v1.0 or later (the latest version is presently available at <http://opencontent.org/opl.shtml>).