WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

FUNCTIONS AND ARRAY IN C

Dr frank guan

INF1002 – Programming Fundamentals

Week 9

RECAP OF LAST WEEK

- C programs consist of modules called functions
- The main() function is the "starting point" fora C program
- The pre-processor (begins with #) executes before a program is compiled
  - definition of symbolic constants
  - the inclusion of other files in the file being compiled
- A variable is a location in memory where a value can be stored for use by a program
- printf(): the first argument, the format control string, describe the output format
- scanf(): The first argument, the format control string, indicates the type of data that should be input by the user
- control structures
  - if – else
  - switch
  - for
  - while
  - do - while

EXPERIENCE SHARING

- “Practice makes perfect”
- Truly common if you spend one week time on:
  - Debugging
  - Troubleshooting
  - Reference reading
  - Discussion
  - Self-learning
  - …

Agenda

1. Functions
2. Arrays
3. Characters and Strings

WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

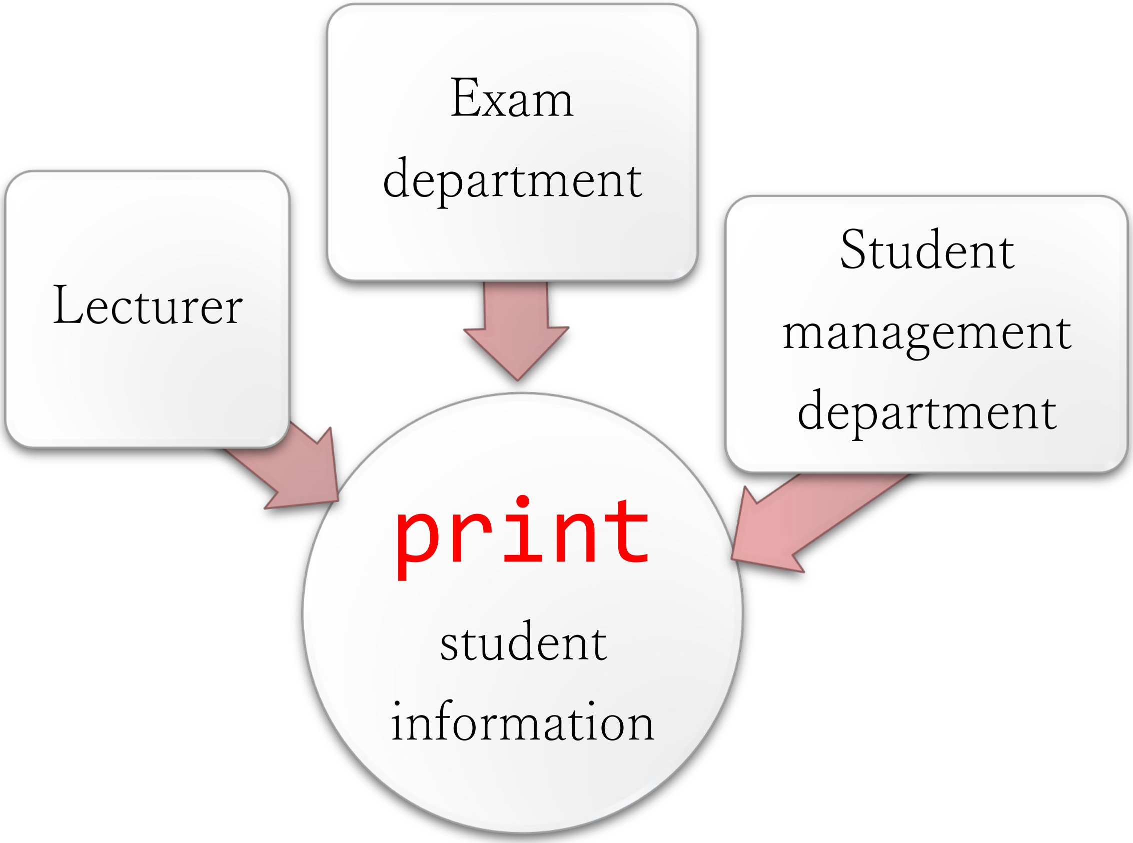
FUNCTIONS IN C

SINGAPORE INSTITUTE OF TECHNOLOGY

FUNCTIONS – C'S BUILDING BLOCK

Functions promote reusability.

Each function is written once.



Exam department

Lecturer

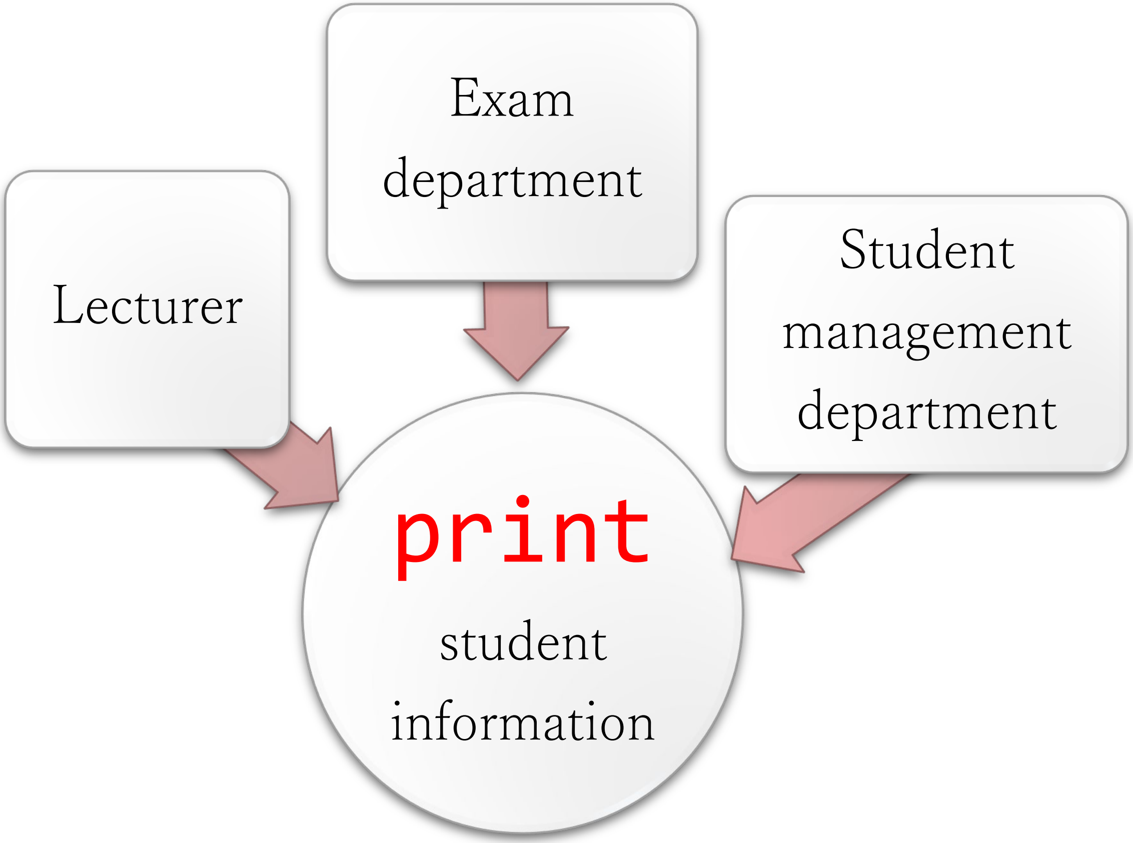
Student management department

print

student information

FUNCTIONS – C'S BUILDING BLOCK

The function can be used repeatedly by many other functions



Exam department

Lecturer

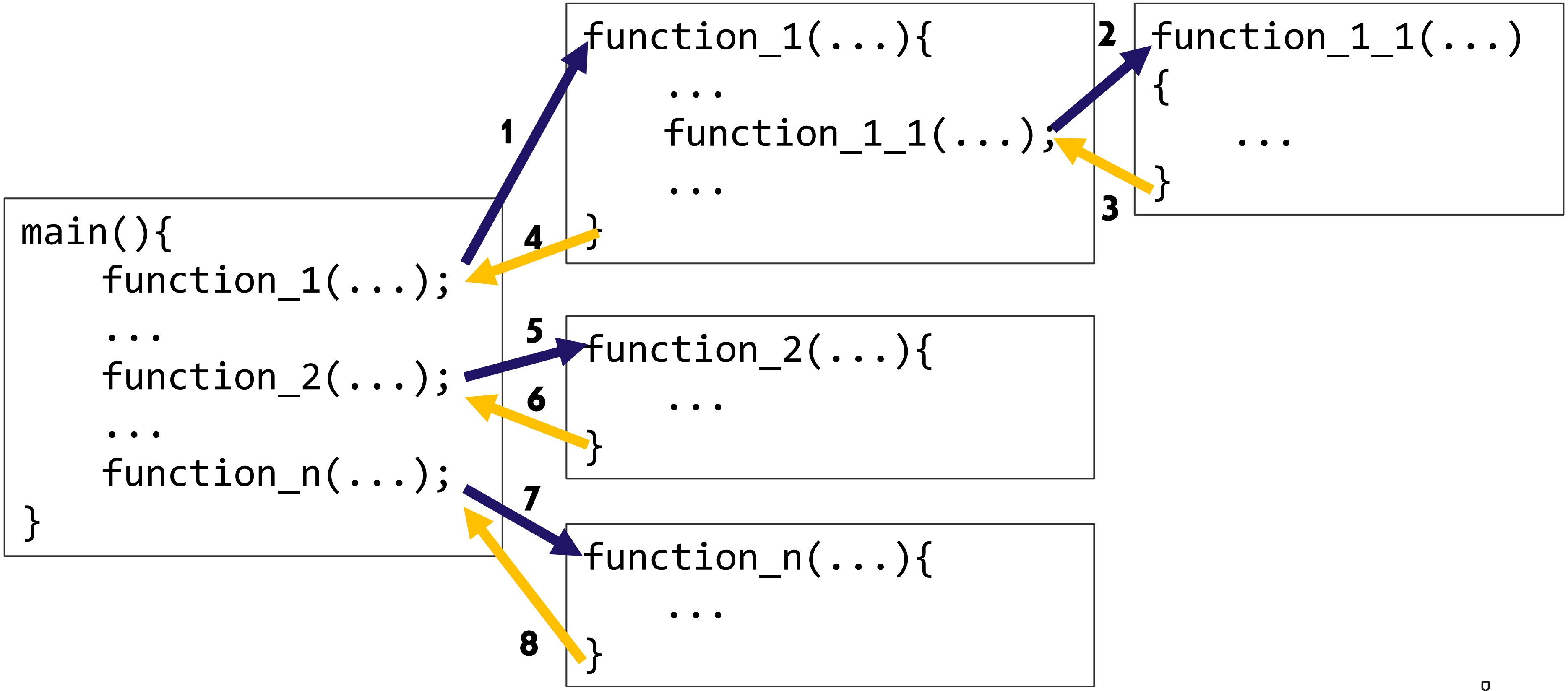
Student management department

print

student information

C FUNCTION CALLS

Functions are invoked by specifying their name and parameters (or arguments) without knowing the actual implementation of the function.



```
main(){
    function_1(...);
    ...
    function_2(...);
    ...
    function_n(...);
}
```

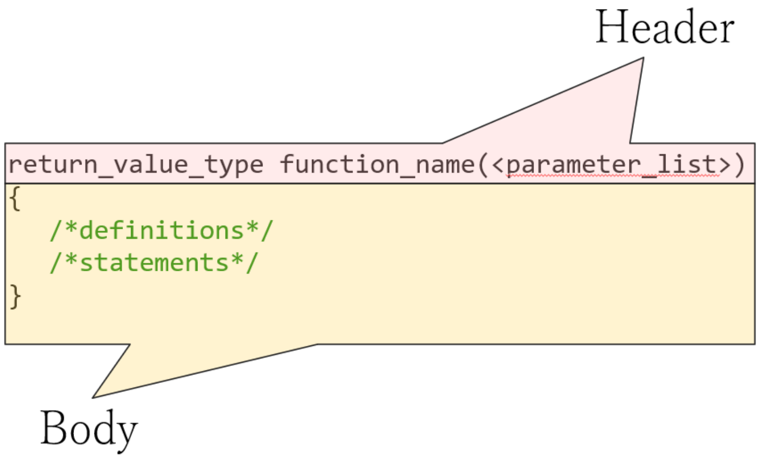
```
function_1(...){
    ...
    function_1_1(...);
    ...
}
```

```
function_1_1(...)
{
    ...
}
```

```
function_2(...){
    ...
}
```

```
function_n(...){
    ...
}
```

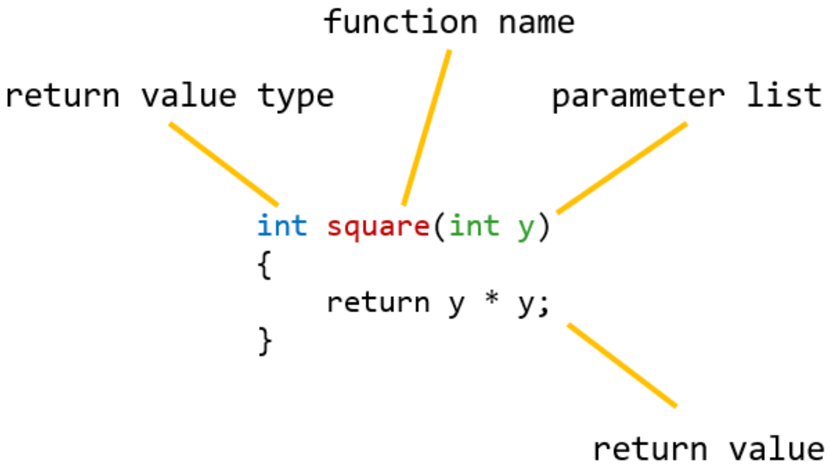
FUNCTION DEFINITIONS



Header

```
return_value_type function_name(<parameter_list>)
{
   /*definitions*/
   /*statements*/
}
```

Body



function name

return value type

parameter list

```
int square(int y)
{
    return y * y;
}
```

return value

The above function calculates the square of an integer **y**

FUNCTION DEFINITIONS

```
return_value_type
function_name(<parameter_list>)
{
   /*definitions*/
   /*statements*/
}
```

- A comma-separated list of parameters received by the function when it is called.
- A type must be listed explicitly for each parameter.
- E.g `int number, char grade`

FUNCTION PROTOTYPES

- A `function prototype` is a function definition without a body, e.g.

```
int square(int);
```

- Function prototypes are optional, but are used by the compiler to validate function calls
  - This prevents errors

- Prototypes are usually declared at the top of a source file, or in a header file

FUNCTION DEFINITION - EXAMPLE

```
/* http://www.programmingsimplified.com */
#include <stdio.h>

/* function prototype */
int addition(int, int);

int main() {

          int first, second, sum;

          /* read input */
          scanf("%d%d", &first, &second);

          /* invoke the addition function */
          sum = addition(first, second);
          printf("%d\n", sum);

          return 0;
}


/* function implementation */
int addition(int a, int b) {

          int result;
          result = a + b;

          return result;
}
```

The function prototype informs the compiler that addition() expects to receive two `integer` values and returns an `integer` result.

FUNCTION DEFINITION - EXAMPLE

```c
/* http://www.programmingsimplified.com */
#include <stdio.h>

/* function prototype */
int addition(int, int);

int main() {

         int first, second, sum;

         /* read input */
         scanf("%d%d", &first, &second);

         /* invoke the addition function */
         sum = addition(first, second);
         printf("%d\n", sum);

         return 0;
}

/* function implementation */
int addition(int a, int b) {

         int result;
         result = a + b;

         return result;
}
```

Whenever there's a function call, the compiler will check that call against the function prototype.

MULTIPLE RETURN STATEMENTS

```
#include <stdio.h>

#define POSITIVE 1
#define NEGATIVE -1
#define ZERO     0

int get_sign(int n);

int main() {

    int n;
    int sign;

    printf("Type an integer: ");
    scanf("%d", &n);

    sign = get_sign(n);

    if (sign == POSITIVE)
        printf("That is a positive number.\n");
    else if (sign == NEGATIVE)
        printf("That is a negative number.\n");
    else
        printf("That is zero.\n");

    return 0;
}
```

```
int get_sign(int n) {

    if (n < 0)
        return NEGATIVE;
    if (n > 0)
        return POSITIVE;

    return ZERO;
}
```

The function stops executing as soon as one `return` statement is executed.

CALLING FUNCTIONS BY VALUE

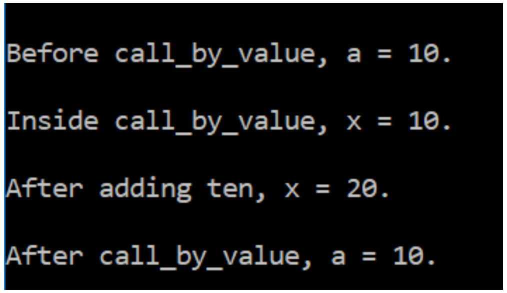
- Call-by-value
  - A copy of the argument's value is made and passed to the called function
  - Changes to the copy do not affect the original variable's value in the caller
  - By default, all calls in C are by value

www.mathwarehouse.com

CALL-BY-VALUE – EXAMPLE

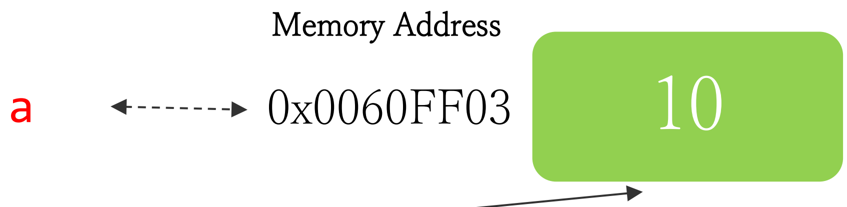
```
#include <stdio.h>

void call_by_value(int);

int main() {

         int a = 10;

         printf(“\nBefore call_by_value, a = %d.\n\n", a);

         call_by_value(a);

         printf("After call_by_value, a = %d.\n \n ", a);

         return 0;
}

/* this function will make a copy of a */
void call_by_value(int x) {

         printf("Inside call_by_value, x = %d.\n \n ", x);
         x += 10;
         printf("After adding ten, x = %d.\n \n ", x);
}
```

Output



```
Before call_by_value, a = 10.

Inside call_by_value, x = 10.

After adding ten, x = 20.

After call_by_value, a = 10.
```

CALL-BY-VALUE – EXAMPLE



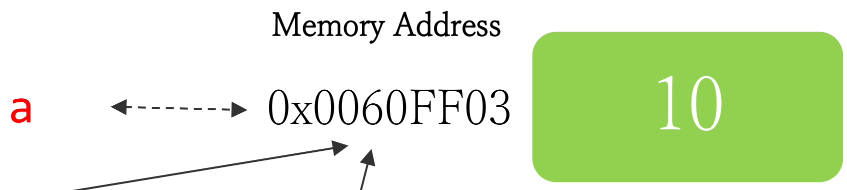
Memory Address

a ←-------→ 0x0060FF03 10

```
#include <stdio.h>

void call_by_value(int);

int main() {

         int a = 10;

         printf(“\nBefore call_by_value, a = %d.\n\n", a);

         call_by_value(a);

         printf("After call_by_value, a = %d.\n \n ", a);

         return 0;
}

/* this function will make a copy of a */
void call_by_value(int x) {

         printf("Inside call_by_value, x = %d.\n \n ", x);
         x += 10;
         printf("After adding ten, x = %d.\n \n ", x);
}
```

CALL-BY-VALUE – EXAMPLE

```
#include <stdio.h>

void call_by_value(int);

int main() {

        int a = 10;

        printf(“\nBefore call_by_value, a = %d.\n\n", a);

        call_by_value(a);

        printf("After call_by_value, a = %d.\n \n ", a);

        return 0;
}

/* this function will make a copy of a */
void call_by_value(int x) {

        printf("Inside call_by_value, x = %d.\n \n ", x);
        x += 10;
        printf("After adding ten, x = %d.\n \n ", x);
}
```



Memory Address

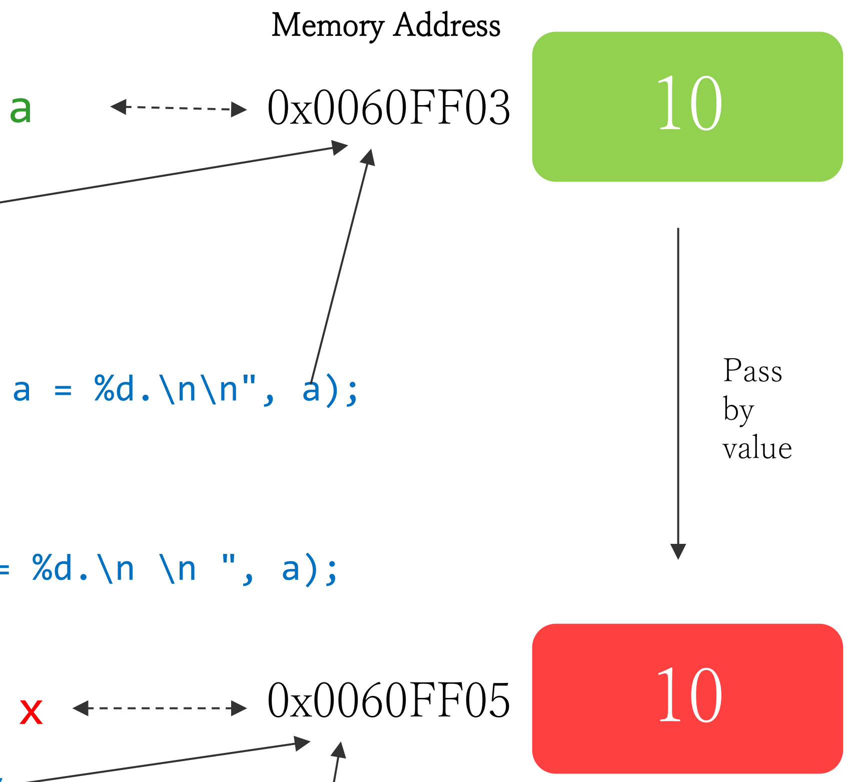
a

0x0060FF03

10

CALL-BY-VALUE – EXAMPLE

```c
#include <stdio.h>

void call_by_value(int);

int main() {

        int a = 10;

        printf(“\nBefore call_by_value, a = %d.\n\n", a);

        call_by_value(a);

        printf("After call_by_value, a = %d.\n \n ", a);

        return 0;
}

/* this function will make a copy of a */
void call_by_value(int x) {

        printf("Inside call_by_value, x = %d.\n \n ", x);
        x += 10;
        printf("After adding ten, x = %d.\n \n ", x);
}
```



Memory Address

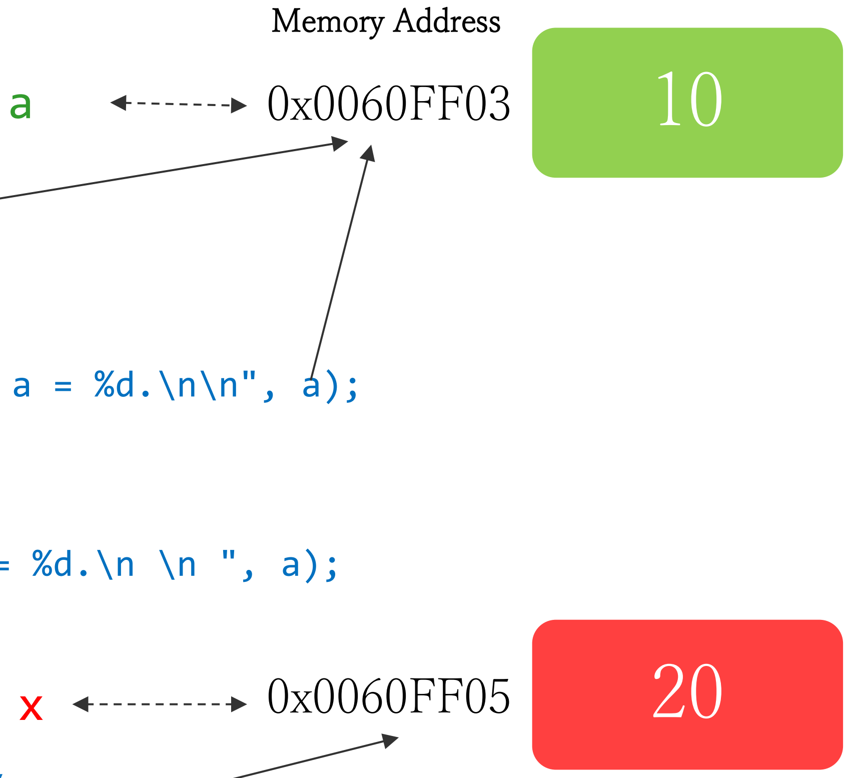
a 0x0060FF03 10

Pass by value

x 0x0060FF05 10

CALL-BY-VALUE – EXAMPLE

```
#include <stdio.h>

void call_by_value(int);

int main() {

        int a = 10;

        printf(“\nBefore call_by_value, a = %d.\n\n", a);

        call_by_value(a);

        printf("After call_by_value, a = %d.\n \n ", a);

        return 0;
}

/* this function will make a copy of a */
void call_by_value(int x) {

        printf("Inside call_by_value, x = %d.\n \n ", x);
        x += 10;
        printf("After adding ten, x = %d.\n \n ", x);
}
```



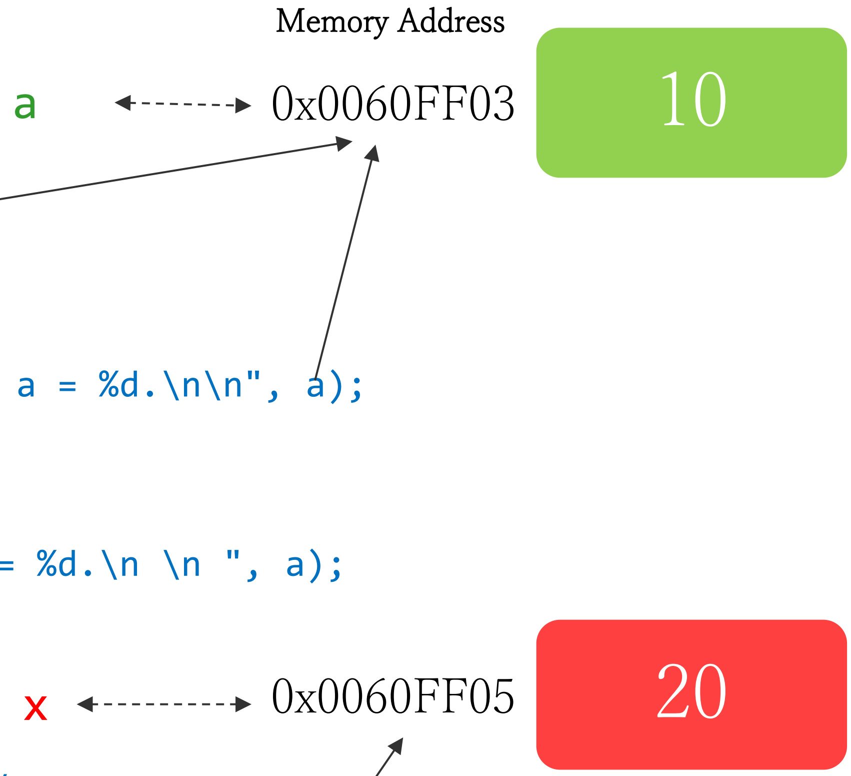
Memory Address

a ↔ 0x0060FF03 10

x ↔ 0x0060FF05 20

CALL-BY-VALUE – EXAMPLE

```
#include <stdio.h>

void call_by_value(int);

int main() {

        int a = 10;

        printf(“\nBefore call_by_value, a = %d.\n\n", a);

        call_by_value(a);

        printf("After call_by_value, a = %d.\n \n ", a);

        return 0;
}

/* this function will make a copy of a */
void call_by_value(int x) {

        printf("Inside call_by_value, x = %d.\n \n ", x);
        x += 10;
        printf("After adding ten, x = %d.\n \n ", x);
}
```



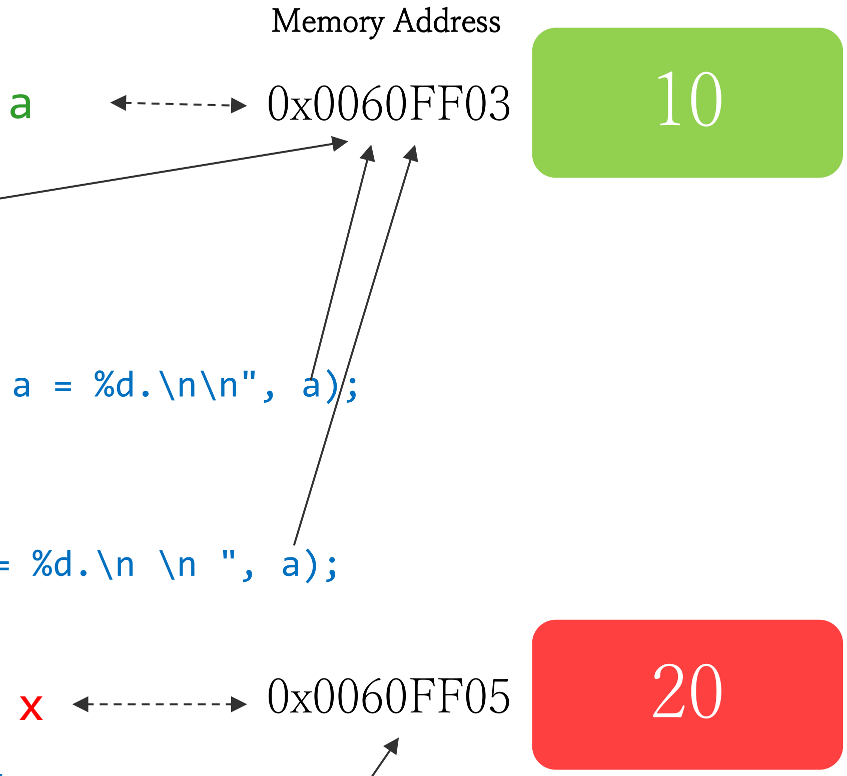
Memory Address

a ↔ 0x0060FF03 | 10

x ↔ 0x0060FF05 | 20

CALL-BY-VALUE – EXAMPLE

```
#include <stdio.h>

void call_by_value(int);

int main() {

        int a = 10;

        printf(“\nBefore call_by_value, a = %d.\n\n", a);

        call_by_value(a);

        printf("After call_by_value, a = %d.\n \n ", a);

        return 0;
}

/* this function will make a copy of a */
void call_by_value(int x) {

        printf("Inside call_by_value, x = %d.\n \n ", x);
        x += 10;
        printf("After adding ten, x = %d.\n \n ", x);
}
```

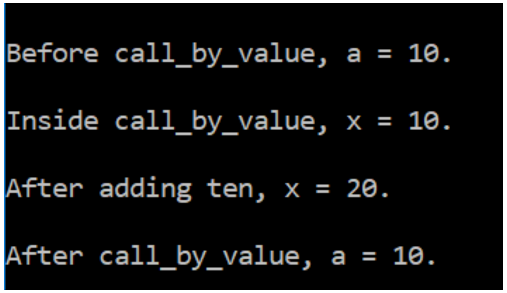


Value of “a” is not updated!

CALL-BY-VALUE – EXAMPLE

```
#include <stdio.h>

void call_by_value(int);

int main() {

         int a = 10;

         printf(“\nBefore call_by_value, a = %d.\n\n", a);

         call_by_value(a);

         printf("After call_by_value, a = %d.\n \n ", a);

         return 0;
}

/* this function will make a copy of a */
void call_by_value(int x) {

         printf("Inside call_by_value, x = %d.\n \n ", x);
         x += 10;
         printf("After adding ten, x = %d.\n \n ", x);
}
```

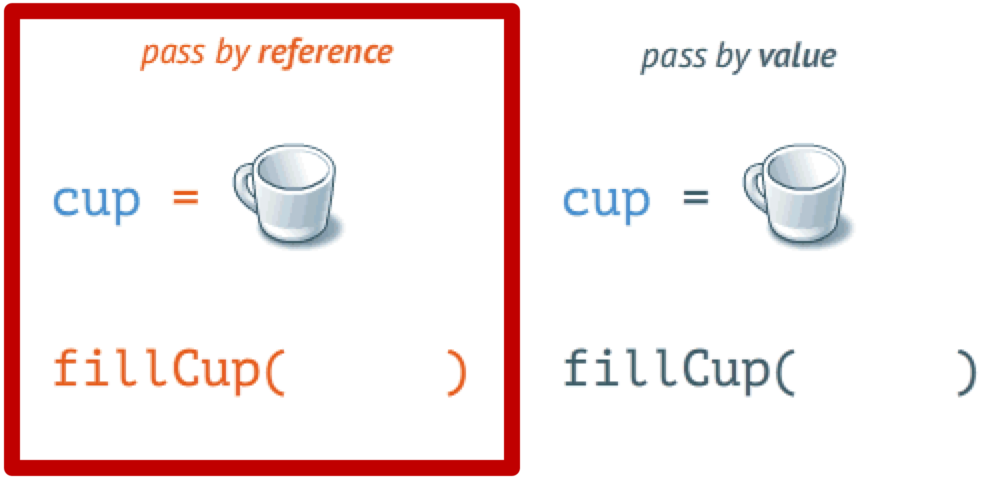
Output



```
Before call_by_value, a = 10.

Inside call_by_value, x = 10.

After adding ten, x = 20.

After call_by_value, a = 10.
```

CALLING FUNCTIONS BY REFERENCE

- Call-by-reference
  - The caller allows the called function to modify the original value
  - Call-by-reference can be simulated by using the address operator (&)
  - More will be covered later…



www.mathwarehouse.com

FUNCTIONS FROM THE STANDARD C LIBRARY

The Standard C Library provides many commonly-used functions, e.g.

Header	Functions
<ctype.h>	Character functions: isalpha(), isdigit(), etc.
<math.h>	Mathematical functions: sqrt(), exp(), log(), sin(), cos(), tan(), etc.
<stdlib.h>	Miscellaneous functions: malloc(), free(), rand(), atoi(), etc.
<stdio.h>	Input/output: printf(), scanf(), fopen(), fread(), fwrite(), etc.
<string.h>	String functions: strcpy(), strcmp(), etc.

FUNCTIONS FROM THE STANDARD C LIBRARY

Visit `cplusplus.com` for a complete list:

- http://www.cplusplus.com/reference/clibrary/

WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

SCOPE RULES

SINGAPORE INSTITUTE OF TECHNOLOGY

SCOPE RULES

The scope of an identifier is the portion of the program in which the identifier can be referenced.

- The same identifier can be re-used in different scopes.

- Tip: the scope of an identifier should be as small as possible.
  - Why?

SCOPE RULES – FILE SCOPE

- An identifier declared outside any function has `file scope`.

- A file-scope identifier is accessible from its declaration until the end of the file.

- Variables with file scope are often called `global variables`.

FILE SCOPE - EXAMPLE

```
#include <stdio.h>

int i = 1;

int main() {

    int x = 4;
    printf("add_i outputs %d\n", add_i(x));
    printf("i is %d\n", i);
    printf("x is %d\n", x);
    return 0;
}

int add_i(int n) {

    int x = n + i;
    i++;

    return x;
}
```

`i` is declared outside any function so `i` has file scope

This statement updates the global variable `i`

SCOPE RULES – BLOCK SCOPE

- Identifiers defined inside a block delimited by braces `{…}` have `block scope`.

- Any block may contain variable definitions.

- Variables with block scope are often called `local variables`.

BLOCK SCOPE - EXAMPLE

```c
#include <stdio.h>

int i = 1;

int main() {

    int x = 4;
    printf("add_i outputs %d\n", add_i(x));
    printf("i is %d\n", i);
    printf("x is %d\n", x);
    return 0;
}

int add_i(int n) {

    int x = n + i;
    i++;

    return x;
}
```

This x has block scope within main()

This x has block scope within add_i()

Output:

```
add_i outputs 5
i is 2
x is 4
```

NEED COFFEE!!

WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

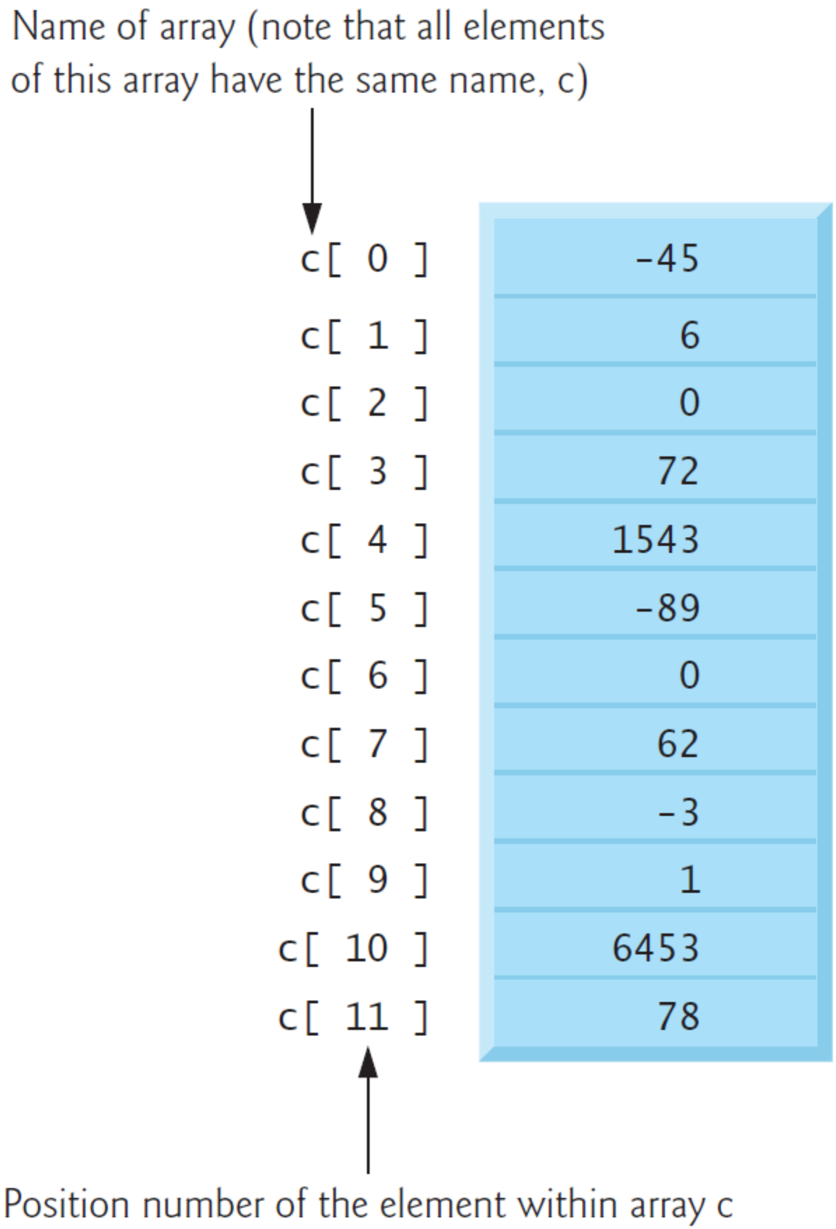
ARRAYS

SINGAPORE
INSTITUTE OF
TECHNOLOGY

ARRAYS

An array is a group of memory locations that all have

- the same **name**
- the same **type**

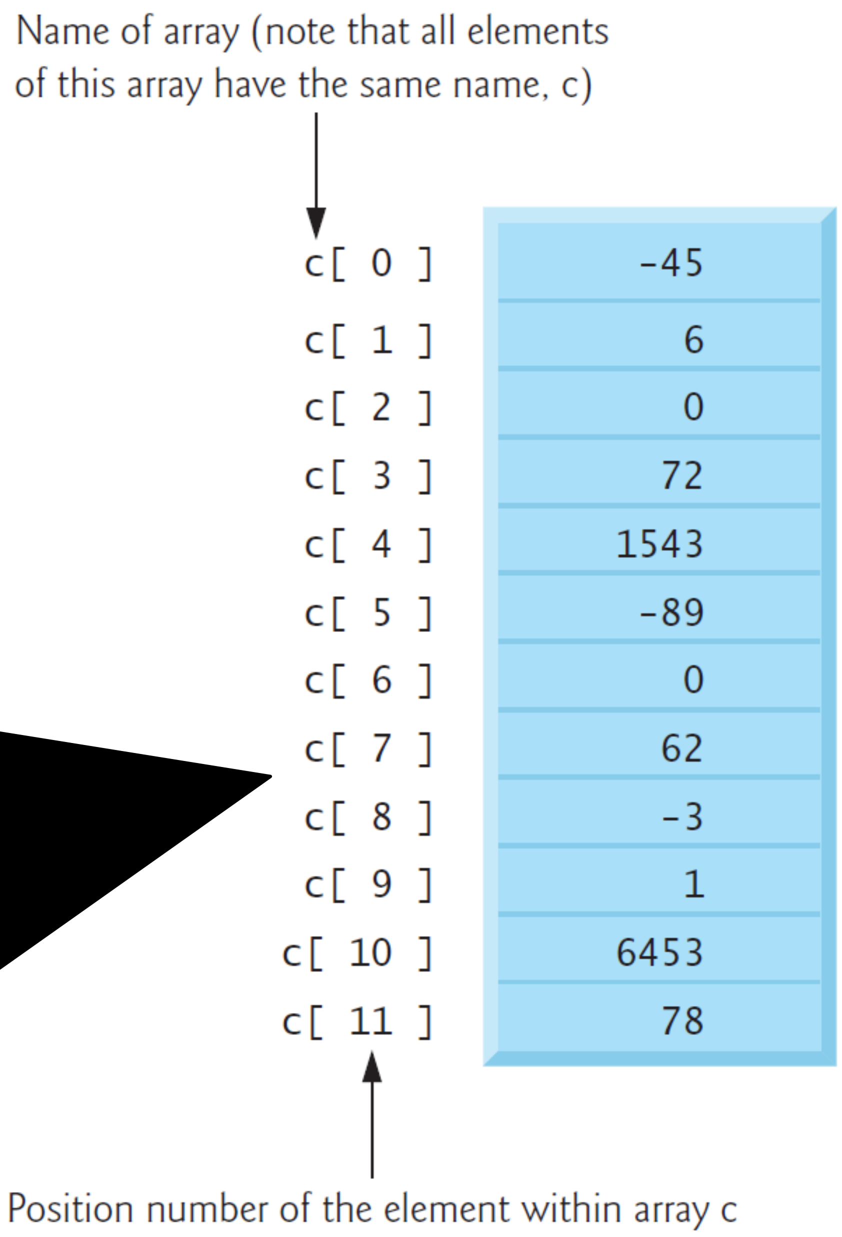


Name of array (note that all elements of this array have the same name, c)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Position number of the element within array c

The position number contained within square brackets is called a **subscript** or **index**.



Name of array (note that all elements of this array have the same name, c)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

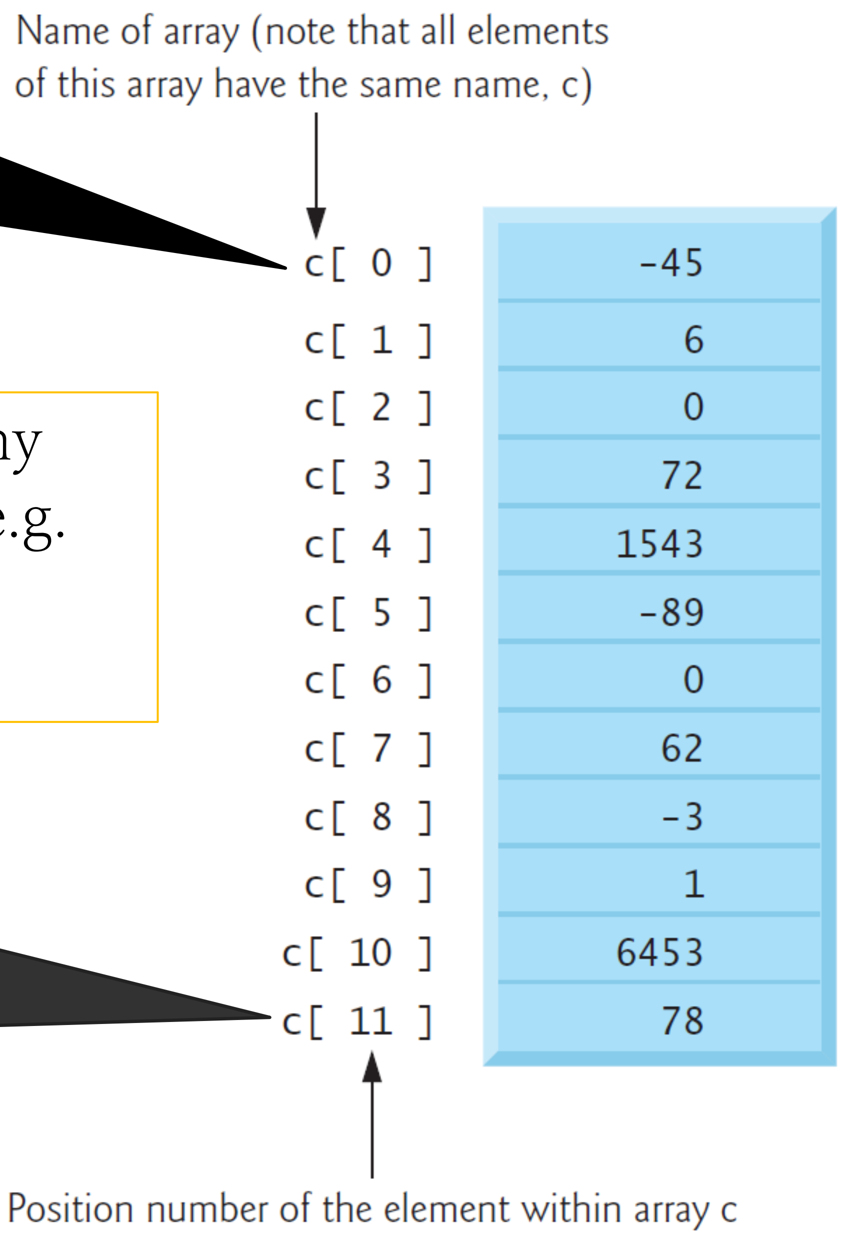
Position number of the element within array c

The index starts at **zero**.

The index can be any integer expression, e.g.

```
int a = 1;
c[a+2] += 2;
```

The index ends at the number of array elements, **minus 1**



Name of array (note that all elements of this array have the same name, c)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Position number of the element within array c

HOW TO USE

```
#include <stdio.h>

#define MAX_STUDENTS 10

int main() {
```

Define

```
    int studentId[MAX_STUDENTS];
```

Initialize

```
    for (int i = 0; i < MAX_STUDENTS; i++)
            studentId[i] = i + 1;

    printf("%7s%13s\n", "Element", "Value");
```

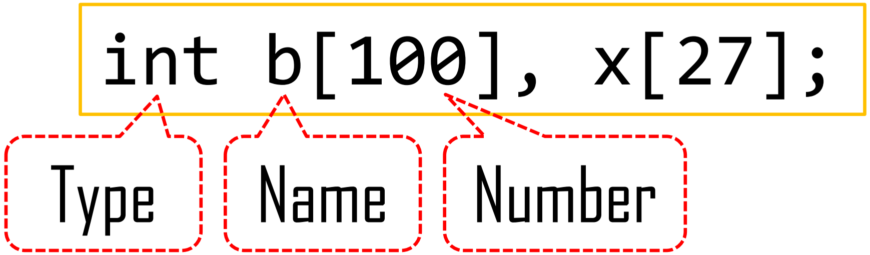
Use

```
    for (int i = 0; i < MAX_STUDENTS; i++)
            printf("%7d%13d\n", i, studentId[i]);

    return 0;
}
```

DEFINING ARRAYS

To define an array we need to specify:

- the type of the elements
- the name of the array
- the number of elements



```
int b[100], x[27];
```

Type

Name

Number

VS

```
int a,       y;
```

ARRAY INITIALISATION

We can use a loop to initialise array elements:

Define an array

Initialize the array

```
#include <stdio.h>

#define MAX_STUDENTS 10

int main() {

        int studentId[MAX_STUDENTS];
        for (int i = 0; i < MAX_STUDENTS; i++)
                studentId[i] = i + 1;

        printf("%7s%13s\n", "Element", "Value");
        for (int i = 0; i < MAX_STUDENTS; i++)
                printf("%7d%13d\n", i, studentId[i]);

        return 0;
}
```

Use the array

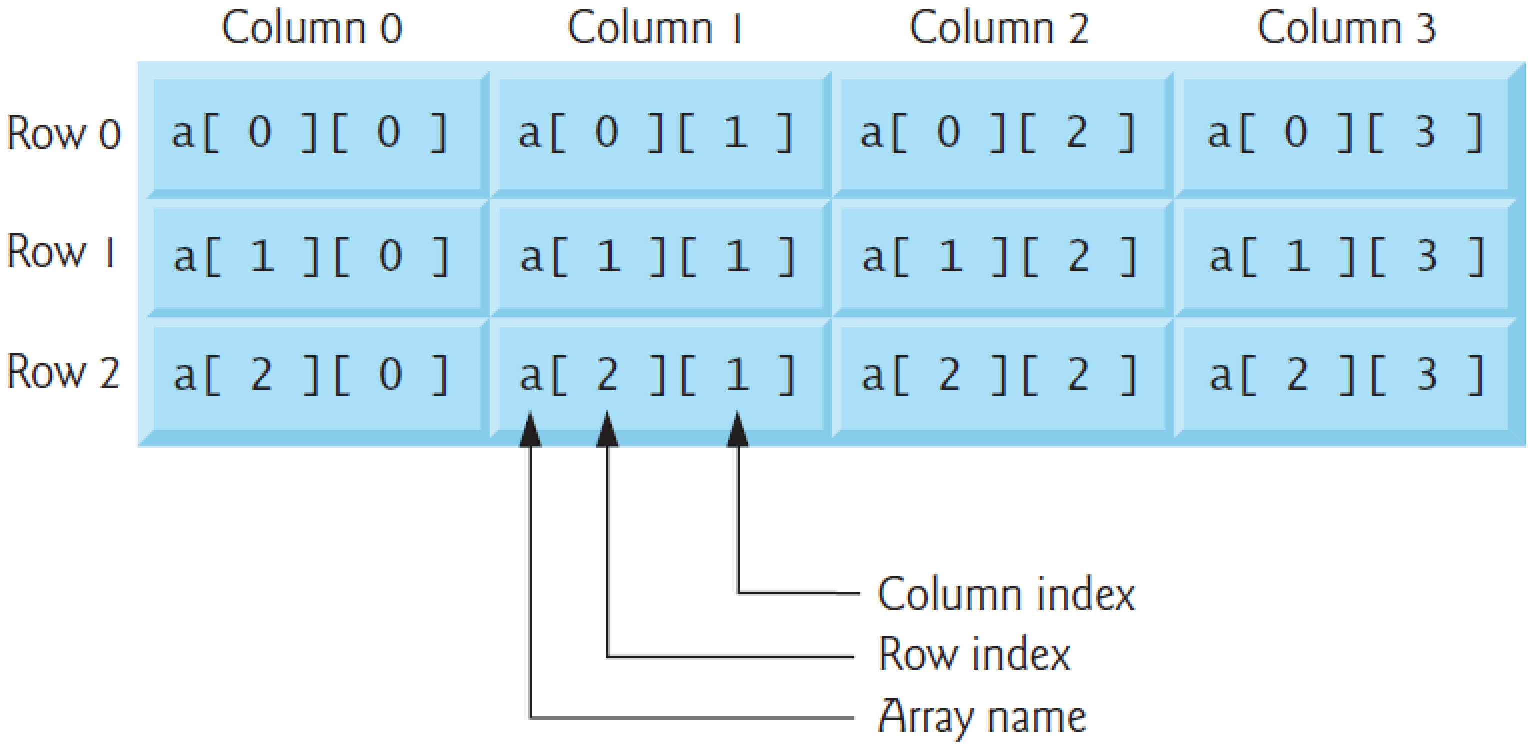
ARRAY INITIALISATION

We can also use an initialiser list:

```
#include <stdio.h>

#define MAX_STUDENTS 10

int main() {

        int studentId[MAX_STUDENTS] =
                { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        printf("%7s%13s\n", "Element", "Value");
        for (int i = 0; i < MAX_STUDENTS; i++)
                printf("%7d%13d\n", i, studentId[i]);

        return 0;
}
```

MULTI-DIMENSIONAL ARRAYS

- The most commonly used are two dimensional arrays or double-subscripted arrays.
- In general, an array with m rows and n columns is called an `m-by-n array`.



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

DECLARING MULTIDIMENSIONAL ARRAYS

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

Declare and initialise a two-dimensional array of integers.

The values in the initialiser list are grouped by row.

b	1	2
	3	4

WE ARE
THINKING TINKERERS
ABLE TO LEARN, UNLEARN AND RELEARN
CATALYSTS FOR TRANSFORMATION
GROUNDED IN THE COMMUNITY
IT'S IN OUR DNA.

STRINGS

SiT SINGAPORE INSTITUTE OF TECHNOLOGY

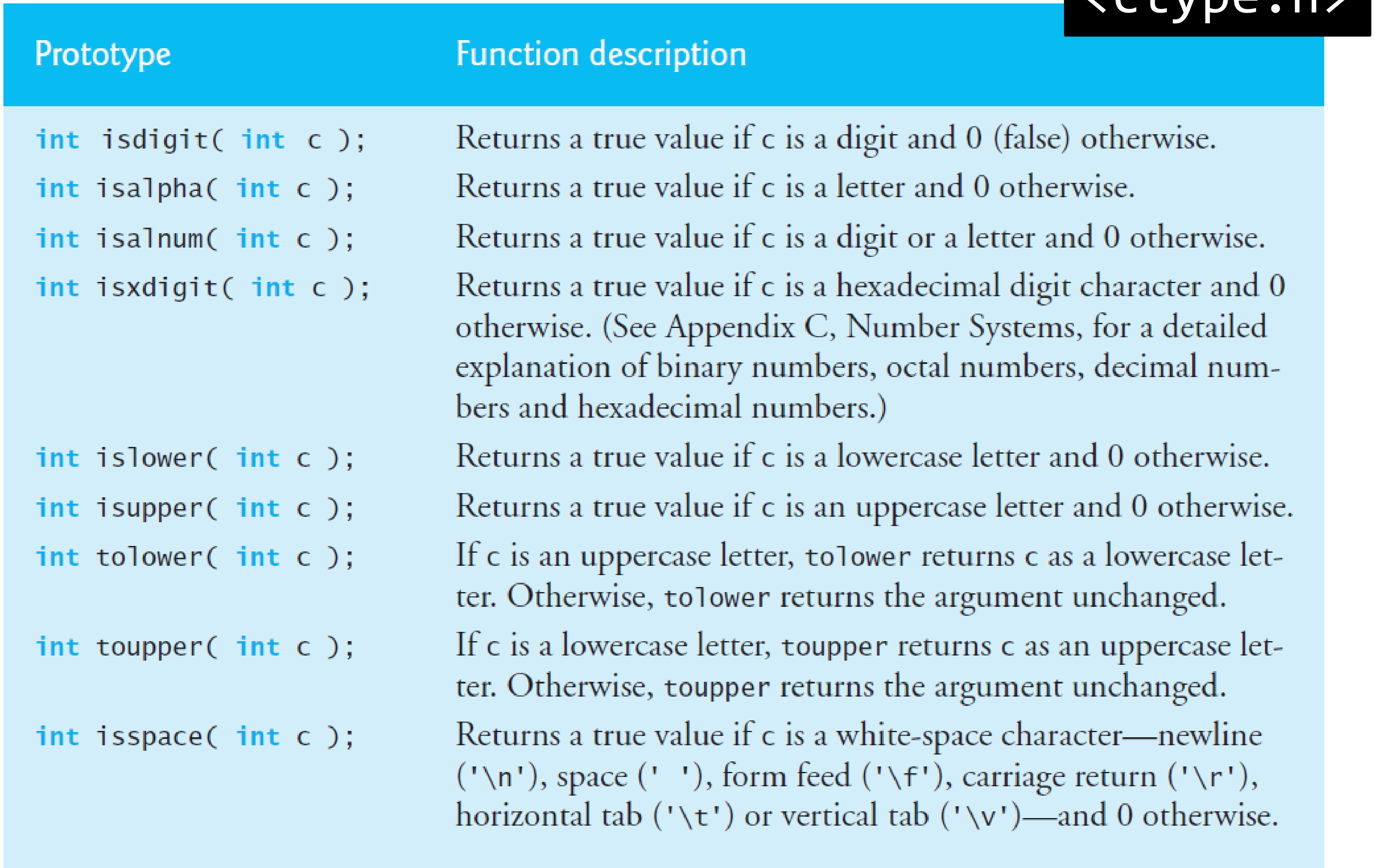
FUNDAMENTALS OF CHARACTERS

A program may contain character constants

- A character constant is denoted by single quotes and has an integer value according to the character set
- E.g. in ASCII:
  - `'z'` has an integer value of 122
  - `'\n'` has an integer value of 10
- Some mathematical operators can be applied to characters
  - + and – move up and down the character set
  - <, >, ==, != compare according to the character set

CHARACTER HANDLING LIBRARY

<ctype.h>



Prototype	Function description
`int isdigit( int c );`	Returns a true value if c is a digit and 0 (false) otherwise.
`int isalpha( int c );`	Returns a true value if c is a letter and 0 otherwise.
`int isalnum( int c );`	Returns a true value if c is a digit or a letter and 0 otherwise.
`int isxdigit( int c );`	Returns a true value if c is a hexadecimal digit character and 0 otherwise. (See Appendix C, Number Systems, for a detailed explanation of binary numbers, octal numbers, decimal numbers and hexadecimal numbers.)
`int islower( int c );`	Returns a true value if c is a lowercase letter and 0 otherwise.
`int isupper( int c );`	Returns a true value if c is an uppercase letter and 0 otherwise.
`int tolower( int c );`	If c is an uppercase letter, `tolower` returns c as a lowercase letter. Otherwise, `tolower` returns the argument unchanged.
`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 a true value if c is a white-space character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`) or vertical tab (`'\v'`)—and 0 otherwise.

FUNDAMENTALS OF STRINGS

A string in C is an **array** of characters **ending in the null character (`'\0'`)**.

String literals (constants) are written in double quotation marks:

- `"SIT-DNA"`
- `"Thinking Tinkerers"`
- `"Able to Learn, Unlearn and Relearn"`
- `"Catalyst for Transformation"`
- `"Grounded in Community"`

STRING INITIALISATION

This is how you can initialise a string:

```
char colour[] = "blue";
```

This creates an array of 5 elements as follows:

b	l	u	e	\0

If no size is specified, the size will be determined based on the number of initialisers in the list. E.g.: 5 in this case

When defining a character array to contain a string, the array must be large enough to store the string and its terminating null character.

STRING INITIALISATION

A string can be stored in an array using **`scanf`**:

```
char word[10];
scanf("%9s\n", word);
```

Variable **`word`** is an array, which is a memory address, so the address operator **`&`** is not needed with argument **`word`**.

STRING INITIALISATION

A string can be stored in an array using `scanf`:

```
char word[10];
scanf("%9s\n", word);
```

Note that `scanf` will read characters until a `space`, `tab`, `newline` or `end-of-file` indicator is encountered. So it is possible that the user could exceed 9 characters and your program might crash.

STRING INITIALISATION

A string can be stored in an array using **`scanf`**:

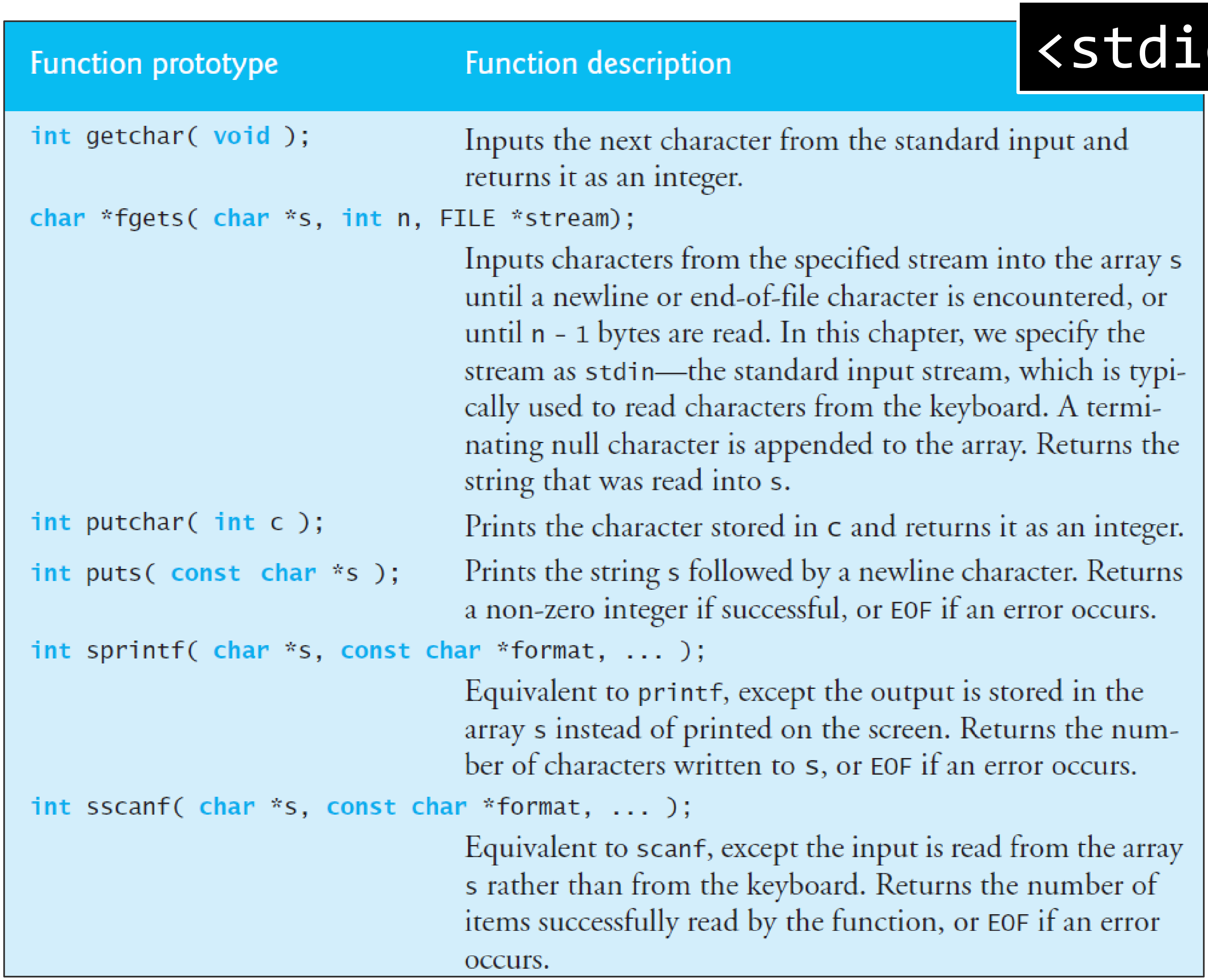
```
char word[10];
scanf("%9s\n", word);
```

It is good practice to use the conversion specifier `%9s` so that **`scanf`** reads up to 9 characters and saves the last one for the null character.

STANDARD I/O FUNCTIONS

FOR STRINGS & CHARACTERS

<stdio.h>



Function prototype	Function description
`int getchar( void );`	Inputs the next character from the standard input and returns it as an integer.
`char *fgets( char *s, int n, FILE *stream);`	Inputs characters from the specified stream into the array s until a newline or end-of-file character is encountered, or until n - 1 bytes are read. In this chapter, we specify the stream as stdin—the standard input stream, which is typically used to read characters from the keyboard. A terminating null character is appended to the array. Returns the string that was read into s.
`int putchar( int c );`	Prints the character stored in c and returns it as an integer.
`int puts( const char *s );`	Prints the string s followed by a newline character. Returns a non-zero integer if successful, or EOF if an error occurs.
`int sprintf( char *s, const char *format, ... );`	Equivalent to printf, except the output is stored in the array s instead of printed on the screen. Returns the number of characters written to s, or EOF if an error occurs.
`int sscanf( char *s, const char *format, ... );`	Equivalent to scanf, except the input is read from the array s rather than from the keyboard. Returns the number of items successfully read by the function, or EOF if an error occurs.

STANDARD I/O FUNCTIONS

FOR STRINGS & CHARACTERS

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

#define MAX_LENGTH 80

/* print the characters of a string in reverse order */
void print_reverse(const char[]);

int main() {

        char sentence[MAX_LENGTH];
        printf("Enter a line of text:\n");
        fgets(sentence, MAX_LENGTH, stdin);
        printf("The input line written backwards:\n");
        print_reverse(sentence);

        return 0;

}
```

fgets() reads characters into an array of chars until a newline or the end-of-file indicator is encountered, or until the maximum number of characters is read.

STANDARD I/O FUNCTIONS

PUTS() AND GETCHAR()

```
#include <stdio.h>

#define MAX_LENGTH 80

int main() {

    char sentence[MAX_LENGTH];
    char c;
    int index = 0;
    puts("Enter a line of text: ");
    while ((c = getchar()) != '\n' && index < MAX_LENGTH - 1)
        sentence[index++] = c;
    sentence[index] = '\0';
    puts("The input line was: ");
    puts(sentence);

    return 0;

}
```

`puts()` takes a string as an argument and prints the string followed by a newline character.

`getchar()` reads a character from the standard input and returns the character as an integer.

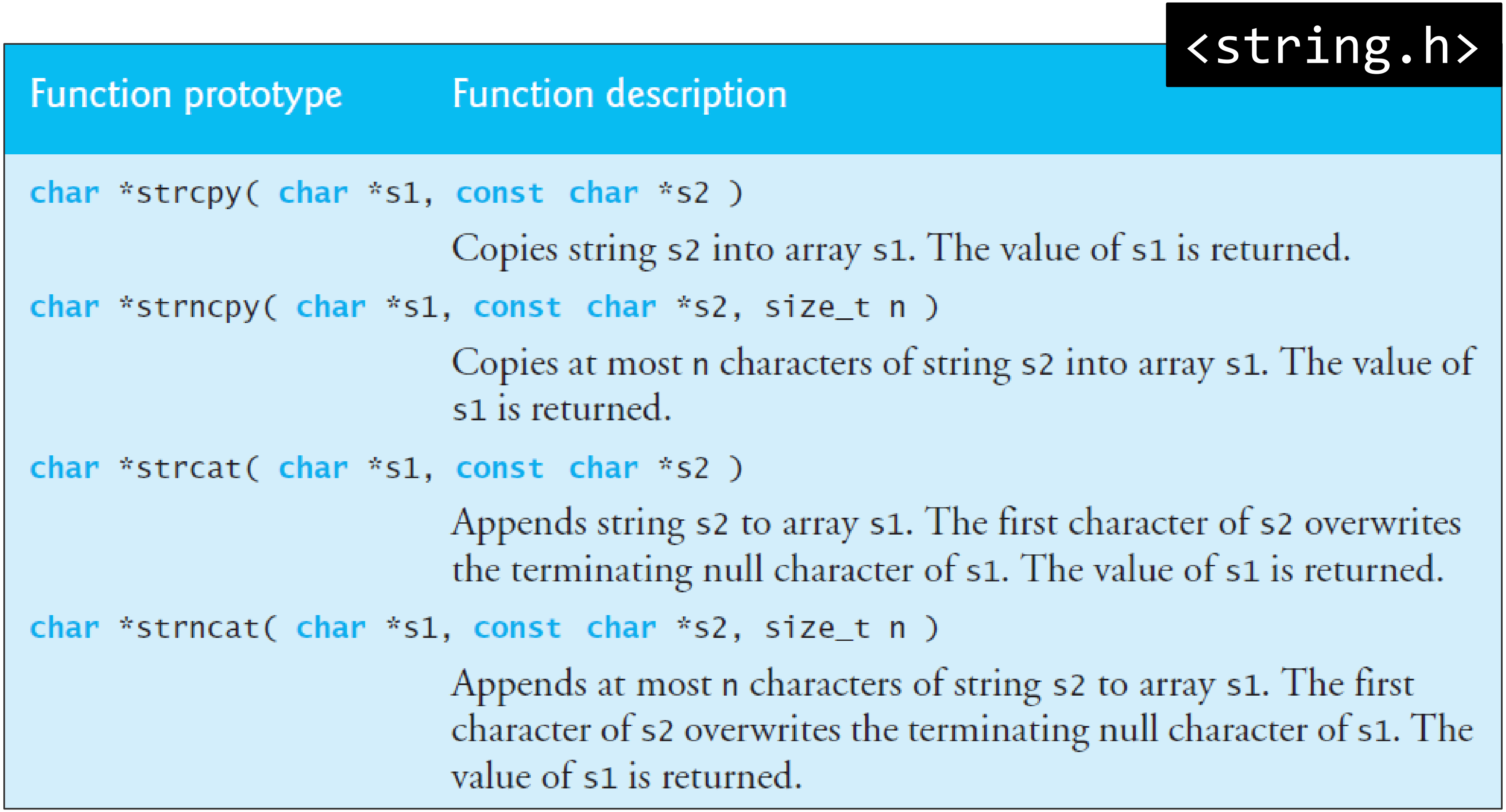
STANDARD I/O FUNCTIONS

FORMATTED DATA TO A STRING WITH SPRINTF()

sprintf() prints formatted data into an array of characters. The function uses the same conversion specifiers as `printf`

```
#include <stdio.h>

#define MAX_LENGTH 80

int main() {

    char s[MAX_LENGTH];
    int x;
    double y;

    printf("Enter an integer and a double: ");
    scanf("%d%lf", &x, &y);

    sprintf(s, "integer: %d, double: %f\n", x, y);
    printf("The formatted string stored in the array is: %s", s);

    return 0;

}
```

STRING MANIPULATION FUNCTIONS



<string.h>

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.

STRING MANIPULATION LIBRARY

EXAMPLE – WHAT DOES THIS PROGRAM DO?

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define MAX_FULLNAME  80
#define MAX_USERNAME  9
#define RANDOM_DIGITS 3

int main() {

    char name[MAX_FULLNAME];
    char userID[MAX_USERNAME];
    int  n, i;

    printf("Enter your name: ");
    scanf("%79s", name);
    strncpy(userID, name, MAX_USERNAME - RANDOM_DIGITS - 1);
    userID[MAX_USERNAME - RANDOM_DIGITS - 1] = '\0';

    n = strlen(userID);
    for (i = 0; i < RANDOM_DIGITS; i++)
        userID[n + i] = '0' + rand() % 10;
    userID[n + RANDOM_DIGITS] = '\0';
    printf("Your username is: %s\n", userID);

    return 0;

}
```

Ensure the string is terminated by ‘\0’.

srand() and rand() generate random numbers.

STRING MANIPULATION LIBRARY

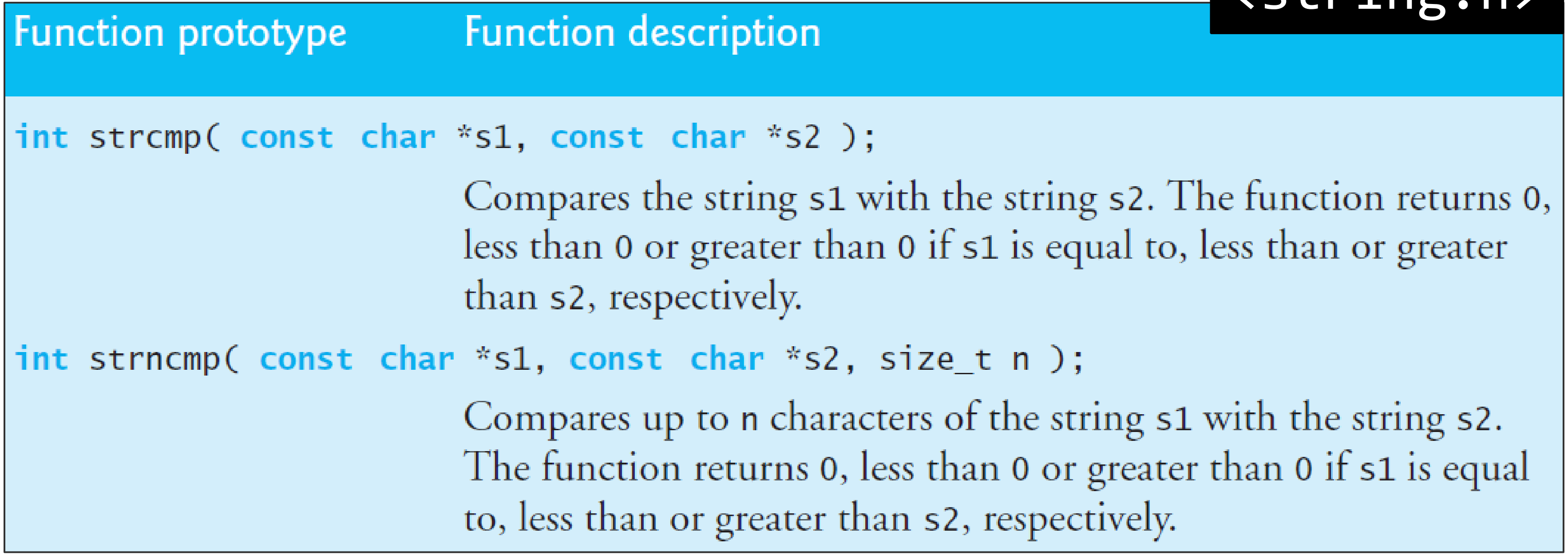
EXAMPLE – OUTPUT

```
Enter your full name: Cristal Ngo Minh Ngoc
Your username is: Crist610
```

```
Enter your full name: Rachel Green
Your username is: Rache822
```

STRING COMPARISON FUNCTIONS

<string.h>



Function prototype	Function description
`int strcmp( const char *s1, const char *s2 );`	Compares the string s1 with the string s2. The function returns 0, less than 0 or greater than 0 if s1 is equal to, less than or greater than s2, respectively.
`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to n characters of the string s1 with the string s2. The function returns 0, less than 0 or greater than 0 if s1 is equal to, less than or greater than s2, respectively.

Note:

`strcmp()` and `strncmp()` are both case-sensitive.

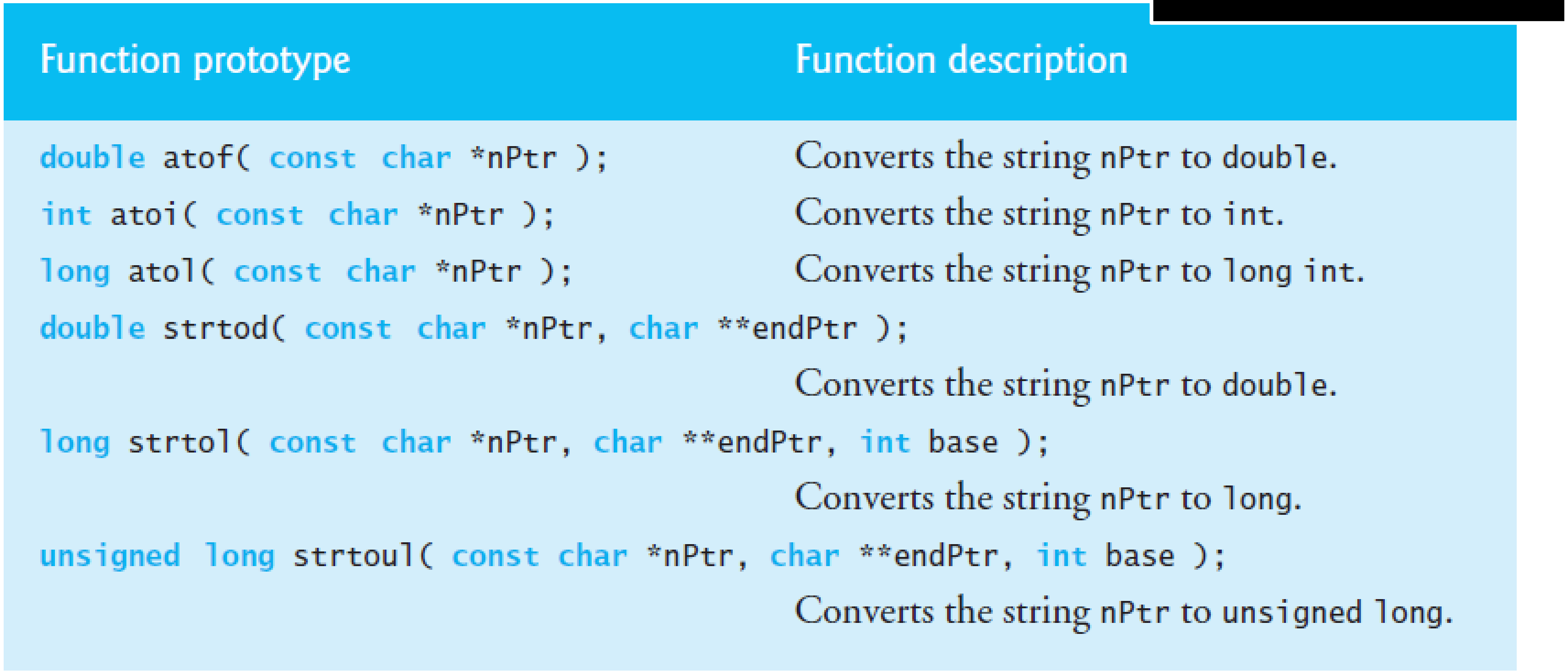
STRING COMPARISON FUNCTIONS

EXAMPLE

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

int main() {

           char word1[20], word2[20];

           printf("Enter two words, separated by a space: ");
           scanf("%19s%19s", word1, word2);

           int c = strcmp(word1, word2);
           if (c < 0)
                      printf("\"%s\" comes first.\n", word1);
           else if (c > 0)
                      printf("\"%s\" comes first.\n", word2);
           else
                      printf("Those two words are the same.\n");

           return 0;
}
```

STRING CONVERSION FUNCTIONS

<stdlib.h>



Function prototype	Function 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 CONVERSION FUNCTIONS

EXAMPLES

```
long l;
l = atol("123456789");
```

convert the string "123456789" to the long integer 123456789

END-OF-WEEK CHECKLIST

- □ Function prototypes
- □ Function definitions
- □ Call by value
- □ Scope rules
- □ Array declarations
- □ Array initialiser lists
- □ Multi-dimensional arrays
- □ Characters
- □ Strings
- □ String manipulation functions
- □ String comparison functions
- □ String I/O functions

GROUP PROJECT

- Group Project
  - Grouping
    - Each team is formed with members from the same lab session
    - Each team will have 5 members (adjustments will be made wherever needed)
    - The final grouping will be announced on LMS
  - Project specs will be uploaded to LMS soonest.
  - Plagiarism is strictly NOT allowed
    - The group project is designed to help you to learn better.
    - Do not copy from peers or from seniors or from other places
    - Do not share your code with others or onto other platforms (e.g. GitHub)

ABOUT USAGE OF AI TOOLS

- For Group Project
  - AI tools are NOT allowed
  - A declaration is needed from each team

- For Test
  - AI tools are NOT allowed