University of Nottingham
UK | CHINA | MALAYSIA

EEEE1042 - Lecture 5
Strings, functions
Autumn Semester 2021.

Dr. Chan, Kheong Sann
kheongsann.chan@nottingham.edu.my

University of Nottingham Malaysia
Department of Electrical and Electronic Engineering

EEEE1042 C++ Programming: Scheduled classes

EEEE1042: for EE students, EEEE1032: for Mecha students.

			EEEE1042	EEEE1032	
Week	Dates	Lecture	Practical	Practical	Assessment
4	Sep 26 – 30	Thu2-4pm			
5	Oct 3 – 09	Thu2-4pm	Mon3-6pm	Wed3-6pm	
6	Oct 10 – 14	Thu2-4pm	P.H.	Wed3-6pm	
7	Oct 17 – 21	Thu2-4pm	Mon3-6pm	Wed3-6pm	PT1 5%
8	Oct 24 – 28	Thu2-4pm	P.H.	Wed3-6pm	
9	Oct 31 –Nov 04	Project Week 1			
10	Nov 07 – 11	Thu2-4pm	Mon3-6pm	Wed3-6pm	PT2 5%
11	Nov 14 – 18	Thu2-4pm	Mon3-6pm	Wed3-6pm	CW1 10%
12	Nov 21 – 25	Project Week 2			
13	Nov 28 –Dec 04	Thu2-4pm	Mon3-6pm	Wed3-6pm	PT3 5%
14	Dec 05 – 09	Project Week 3			
15	Dec 12 – 16	Thu2-4pm	P.H.	Wed3-6pm	PT4 5%
16	Dec 19 – 23	Study Week			CW2 30%
17-18	Dec 26 – Jan 06	Study Weeks			
19-20	Jan 09 – 21	Final Exam (40%)			

Outline EEEE1042 C Lecture 5:

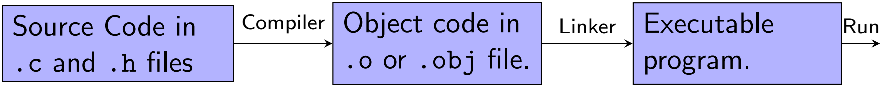
1. Functions
   - Call by value vs call by reference
   - Recursion
2. Generating Random Numbers
3. String functions in C
   - Strings Chars and Pointers
   - C string library

Function declarations and definitions

Three main parts of the program

1. Function declarations ⇒ at beginning or in header files.
2. Main body ⇒ in the main c/cpp file.
3. Function definitions ⇒ at end or own function file.

```c
#include <stdio.h>
int printAndAdd (int , int ); // function declaration
int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}
int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```



Source Code in .c and .h files

Compiler

Object code in .o or .obj file.

Linker

Executable program.

Run

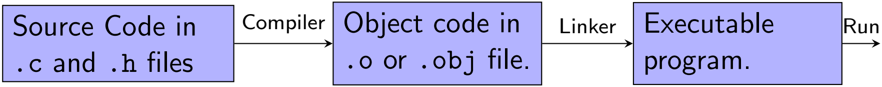
Function declarations and definitions

Three main parts of the program

1. Function declarations ⇒ at beginning or in header files.
2. Main body ⇒ in the main c/cpp file.
3. Function definitions ⇒ at end or own function file.

```
#include <stdio.h>
int printAndAdd (int , int ); // function declaration
int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}
int printAndAdd (int x, int y) { // functio
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```

The **function declaration**, goes at the top or in its own header file. As the function declaration only needs to know the **type** of the input parameters, the name of the variable can be omitted.


Source Code in .c and .h files
Compiler
Object code in .o or .obj file.
Linker
Executable program.
Run

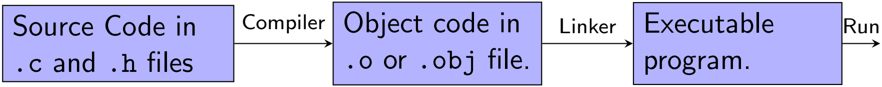
Function declarations and definitions

Three main parts of the program

1. Function declarations ⇒ at beginning or in header files.
2. Main body ⇒ in the main c/cpp file.
3. Function definitions ⇒ at end or own function file.

```
#include <stdio.h>
int printAndAdd (int , int ); // function
int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}
int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```

The **function call** can be made after function has been declared.


Source Code in .c and .h files
Compiler
Object code in .o or .obj file.
Linker
Executable program.
Run

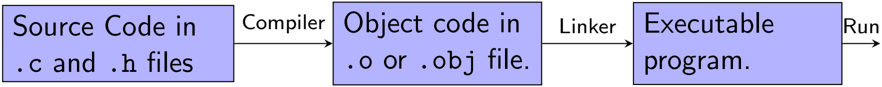
Function declarations and definitions

Three main parts of the program

1. Function declarations ⇒ at beginning or in header files.
2. Main body ⇒ in the main c/cpp file.
3. Function definitions ⇒ at end or own function file.

```c
#include <stdio.h>
int printAndAdd (int , int ); // function declaration
int main () {
    int x;
    x=printAndAdd(3,6); // function call
    return 0;
}
int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
    return(z);
}
```

The **function definition** is not needed until the linking stage. It can go in its own file and be compiled separately.



Source Code in .c and .h files

Compiler

Object code in .o or .obj file.

Linker

Executable program.

Run

Functions in POP and OOP

Functions are the heart of POP (procedural-oriented programming).

- Every chunk of code that performs a self-contained task with potential for re-use should be separated out into its own function.
- In bigger projects, functions are separated into their own file and compiled separately
  - Enables more modular code with easier re-use
  - The function file can be compiled separately and independently from the main.
  - No need to recompile subfunctions, when working on the main.

Functions are still used in OOP (object-oriented programming), but they are bundled together with data of the object.

- This helps to manage the functions better.
- Function management is important as project grows in size

Function example: prime number

Code without functions/comments

```
for (i=1,k=3;i<N;k++) {
   for (j=0;j<i;j++) {
      if (k%x[j]==0)
         goto notPrime;
   }
   printf("%3d %6d\n",i,k);
   x[i]=k;
   i++;
   notPrime:
}
```

Code with functions/comments

```
// Iterate over each test
// number k
for (i=1,k=3;i<N;k++) {
   if (isPrime(k,x,i)) {
      printf("%3d %6d\n",i,k);
// Save prime, and increment
      x[i]=k;
      i++;
   }
}
```

You should aim to produce the code on the right.

Functions: Call by reference vs call by value

1. Call-by-value

```
int printAndAdd (int x, int y); // function declaration

int main () {
    int x=printAndAdd(3,6); // function call
}
```

In call-by-value, when the function is called, the **value** in the calling environment is **copied** into the variables in the local function.
⇒ function **unable** to affect variables in calling environment.

2. Call-by-reference

```
int printAndAdd (int* x, int* y); // function declaration

int main () {
    int x=3,y=6,z=printAndAdd(&x,&y); // function call
}
```

In call-by-reference, the **address** of the variable in the calling environment is passed to the variables in the local function.
⇒ function **is able** to affect variables in calling environment.

Functions: Call by reference vs call by value

1. Call-by-value

```
int printAndAdd (int x, int y); // function declaration

int main () {
    int x=printAndAdd(3,6); // function call
}
```

values copied into local variables

In call-by-value, when the function is called, the **value** in the calling environment is **copied** into the variables in the local function.
⇒ function **unable** to affect variables in calling environment.

2. Call-by-reference

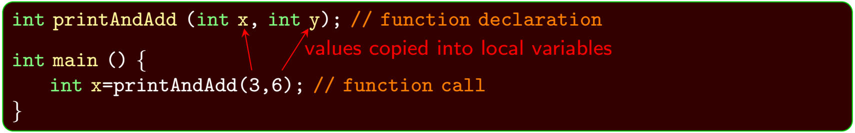
```
int printAndAdd (int* x, int* y); // function declaration

int main () {
    int x=3,y=6,z=printAndAdd(&x,&y); // function call
}
```

In call-by-reference, the **address** of the variable in the calling environment is passed to the variables in the local function.
⇒ function **is able** to affect variables in calling environment.

Functions: Call by reference vs call by value

1. Call-by-value



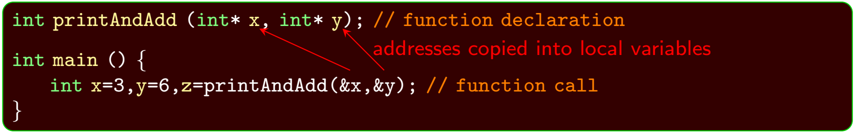
```
int printAndAdd (int x, int y); // function declaration

int main () {
    int x=printAndAdd(3,6); // function call
}
```

values copied into local variables

In call-by-value, when the function is called, the **value** in the calling environment is **copied** into the variables in the local function.
⇒ function **unable** to affect variables in calling environment.

2. Call-by-reference



```
int printAndAdd (int* x, int* y); // function declaration

int main () {
    int x=3,y=6,z=printAndAdd(&x,&y); // function call
}
```

addresses copied into local variables

In call-by-reference, the **address** of the variable in the calling environment is passed to the variables in the local function.
⇒ function **is able** to affect variables in calling environment.

Example: Call-by-value

```c
#include <stdio.h>
int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
// Reassign input variables, to see if affects calling environment
    x=100; y=100;
    printf("In printAndAdd, reassigning x and y: x=%d, y=%d\n",x,y);
    return(z);
}
int main () {
    int x=3, y=6, z;
    z=printAndAdd(x,y); // function call
    printf("Inside main function: x=%d, y=%d, z=%d\n",x,y,z);
    return 0;
}
```

Example: Call-by-value

```c
#include <stdio.h>
int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
// Reassign input variables, to see if affects calling environment
    x=100; y=100;
    printf("In printAndAdd, reassigning x and y: x=%d, y=%d\n",x,y);
    return(z);
}
int main () {
    int x=3, y=6, z;
    z=printAndAdd(x,y); // function call
    printf("Inside main function: x=%d, y=%d, z=%d\n",x,y,z);
    return 0;
}
```

Output:

```
In function printAndAdd: x+y=9
In printAndAdd, reassigning x and y: x=100, y=100
Inside main function: x=3, y=6, z=9
```

Example: Call-by-value

```c
#include <stdio.h>
int printAndAdd (int x, int y) { // function definition
    int z=x+y;
    printf("In function printAndAdd: x+y=%d\n",z);
// Reassign input variables, to see if affects calling environment
    x=100; y=100;
    printf("In printAndAdd, reassigning x and y: x=%d, y=%d\n",x,y);
    return(z);
}
int main () {
    int x=3, y=6, z;
    z=printAndAdd(x,y); // function call
    printf("Inside main function: x=%d, y=%d, z=%d\n",x,y,z);
    return 0;
}
```

Output:

```
In function printAndAdd: x+y=9
In printAndAdd, reassigning x and y: x=100, y=100
Inside main function: x=3, y=6, z=9
```

⇒ Variables in function scope **do not** affect variables in calling scope

Call-by-value

Calling by value has limitations.

1. You can't pass an array when using call-by-value.
    - Arrays are pointers, and passing a pointer ot the array is using call-by-reference
    - Call-by-value only passes single values.
2. C functions can only return a single value
    - If you want your function to return more than one value, you must use call-by-reference:
    - Pass the address of an output variable to the function.
    - Function modifies the variable in the calling environment.

Example: Call-by-reference

```c
#include <stdio.h>
int printAndAdd (int* x, int* y) { // function definition
    int z=*x+*y;
    printf("In function printAndAdd: *x+*y=%d\n",z);
// Reassign input variables, to see if affects calling environment
    *x=100; *y=100;
    printf("In printAndAdd, reassigning *x and *y: x=%d, y=%d\n",*x,*y);
    return(z);
}
int main () {
    int x=3, y=6, z;
    z=printAndAdd(&x,&y); // function call
    printf("Inside main function: x=%d, y=%d, z=%d\n",x,y,z);
    return 0;
}
```

Example: Call-by-reference

```
#include <stdio.h>
int printAndAdd (int* x, int* y) { // function definition
    int z=*x+*y;
    printf("In function printAndAdd: *x+*y=%d\n",z);
// Reassign input variables, to see if affects calling environment
    *x=100; *y=100;
    printf("In printAndAdd, reassigning *x and *y: x=%d, y=%d\n",*x,*y);
    return(z);
}
int main () {
    int x=3, y=6, z;
    z=printAndAdd(&x,&y); // function call
    printf("Inside main function: x=%d, y=%d, z=%d\n",x,y,z);
    return 0;
}
```

Output:

```
In function printAndAdd: *x+*y=9
In printAndAdd, reassigning *x and *y: x=100, y=100
Inside main function: x=100, y=100, z=9
```

Example: Call-by-reference

```
#include <stdio.h>
int printAndAdd (int* x, int* y) { // function definition
    int z=*x+*y;
    printf("In function printAndAdd: *x+*y=%d\n",z);
// Reassign input variables, to see if affects calling environment
    *x=100; *y=100;
    printf("In printAndAdd, reassigning *x and *y: x=%d, y=%d\n",*x,*y);
    return(z);
}
int main () {
    int x=3, y=6, z;
    z=printAndAdd(&x,&y); // function call
    printf("Inside main function: x=%d, y=%d, z=%d\n",x,y,z);
    return 0;
}
```

Output:

```
In function printAndAdd: *x+*y=9
In printAndAdd, reassigning *x and *y: x=100, y=100
Inside main function: x=100, y=100, z=9
```

⇒ Variables in function scope **can** affect variables in calling scope

Call-by-reference example: Incrementing an array

```c
#include <stdio.h>
void printArray(int *a, int N) {
    // print all elements in array a[] of length N
    int i;
    for (i=0;i<N;i++) printf("%3d %3d\n",i,a[i]);
}
void incrementArray(int *a, int N) {
    int i;
    // update all elements in array a[] of length N.
    for (i=0;i<N;i++) a[i]++;
}
int main () {
    int a[]={20, 15, 18, 33, 25, 13};
    printf("Before increment\n");
    printArray(a,6);
    incrementArray(a,6);
    printf("After increment\n");
    printArray(a,6);
    return 0;
}
```

Call-by-reference example: Incrementing an array

```c
#include <stdio.h>
void printArray(int *a, int N) {
    // print all elements in array a[] of length N
    int i;
    for (i=0;i<N;i++) printf("%3d %3d\n",i,a[i]);
}
void incrementArray(int *a, int N) {
    int i;
    // update all elements in array a[] of length N.
    for (i=0;i<N;i++) a[i]++;
}
int main () {
    int a[]={20, 15, 18, 33, 25, 13};
    printf("Before increment\n");
    printArray(a,6);
    incrementArray(a,6);
    printf("After increment\n");
    printArray(a,6);
    return 0;
}
```

Output:

```
Before increment
0 20
1 15
2 18
3 33
4 25
5 13
After increment
0 21
1 16
2 19
3 34
4 26
5 14
```

Recursive functions

- Recursive functions are a special type of function that are allowed to call themselves.
- You might think this could result in an infinite loop, which it can if you do not control the exit condition properly
- Recursion is a useful tool to in certain cases such as in the case of computing factorials:
  Definition: $n! = n \times (n-1)!$ for $n \geq 1$ and
  $0! = 1$
- We can use recursion to evaluate $n!$ by multiplying n by $(n-1)!$, ie: a function calling itself.
- The terminating condition is that when $n = 0$ the function should return 1.

Example Recursion: Computing Factorial

```c
#include <stdio.h>
// Recursive function factorial
long unsigned int factorial (long unsigned int n) {
    if (n==0) return(1); // 0! = 1
    else return(n*factorial(n-1));
}
int main () {
    int i;
    for (i=0;i<10;i++) printf("%d! = %lu\n",i,factorial(i));
    return 0;
}
```

Example Recursion: Computing Factorial

```c
#include <stdio.h>
// Recursive function factorial
long unsigned int factorial (long unsigned int n) {
    if (n==0) return(1); // 0! = 1
    else return(n*factorial(n-1));
}
int main () {
    int i;
    for (i=0;i<10;i++) printf("%d! = %lu\n",i,factorial(i));
    return 0;
}
```

Output:

```
    0! = 1
    1! = 1
    2! = 2
    3! = 6
    4! = 24
    5! = 120
    6! = 720
    7! = 5040
    8! = 40320
    9! = 362880
```

Outline EEEE1042 C Lecture 5:

1. Functions
   - Call by value vs call by reference
   - Recursion
2. Generating Random Numbers
3. String functions in C
   - Strings Chars and Pointers
   - C string library

Generating Random Numbers

- Sometimes you need the computer to generate random numbers for your program
- Computers cannot generate truly random numbers, but they can generate **pseudo-random** numbers.
- **Pseudo-random numbers generators** (PRNG) are an algorithm that produce numbers having the appearance of being random.
  - However pseduo-random numbers are not truly random. They will repeat with some period N.
  - If the period of repetition is large enough, then for all intents and purposes, the numbers will be random.
- The issue with PRNGs is about making the period big enough that the output never repeats within the duration of usage.

The Standard C PRNG

Reference: Press, Teukolsky, Vetterling Flannery "Numerical Recipes in C", Second Edition, chapter 7.

The synopsis for generating random numbers using the standard C library routine is:

```
#include <stdlib.h>
#define RAND_MAX ...
void srand(unsigned seed);
int rand(void);
```

- The function `void srand(unsigned seed);` initializes the random seed to the given value. This value is used to seed the random number generator that allows it to generate the future random number sequences.
- The function `int rand (void);` returns an integer between 0 and `RAND_MAX` **inclusive**.

Example generating random numbers

```c
#include<stdio.h>
#include<stdlib.h>
int main () {
   srand(1); // Initialize the random seed
   printf("Random number: %d\n",rand());
   printf("Random number: %d\n",rand());
   printf("Random number: %d\n",rand());
   printf("Random number: %d\n",rand());
   printf("RAND_MAX=%d\n",RAND_MAX);
   return 0;
}
```

Output:

```
Random number: 1804289383
Random number: 846930886
Random number: 1681692777
Random number: 1714636915
RAND_MAX=2147483647
```

The same seed should generate the same random sequence of numbers.

Generating uniformly distributed floats

To generate a uniformly distributed random floating number, between 0 and 1 divide by (RAND_MAX+1.0):

```
#include<stdio.h>
#include<stdlib.h>
int main () {
   printf("Random number:%f\n",rand()/(RAND_MAX+1.0));
   printf("Random number:%f\n",rand()/(RAND_MAX+1.0));
   printf("Random number:%f\n",rand()/(RAND_MAX+1.0));
   printf("Random number:%f\n",rand()/(RAND_MAX+1.0));
   return 0;
}
```

Output:

```
Random number: 0.840188
Random number: 0.394383
Random number: 0.783099
Random number: 0.798440
```

The same seed should generate the same random sequence of numbers.

Generating uniform integers between 1 and 10

To generate uniformly distributed random integers between 1 and 10 use the most significant bits (MSBs):

`1+(int) (10.0*rand()/(RAND_MAX+1.0));`

```
#include<stdio.h>
#include<stdlib.h>
int main () {
   int x;
   x=1+(10.0*rand())/(RAND_MAX+1.0);
   printf("Random number: %d\n",x);
   x=1+(10.0*rand())/(RAND_MAX+1.0);
   printf("Random number: %d\n",x);
   return 0;
}
```

Output:

```
Random number: 9
Random number: 4
```

Don't use the LSB's, ie: don't do: `(rand()%10)+1.0`.

Generating uniform integers between 1 and 10

To generate uniformly distributed random integers between 1 and 10 use the most significant bits (MSBs):

`1+(int) (10.0*rand()/(RAND_MAX+1.0));`

The MSB's and LSB's of the a 4-byte integer:

1	1	0	0	0	1	0	1	1	0	1	0	1	1	1	0	0	1	0	0	1	1	0	1	1	1	0	1	0	1	0	0

MSB's

LSB's

```
    x=1+(10.0*rand())/(RAND_MAX+1.0);
    printf("Random number: %d\n",x);
    x=1+(10.0*rand())/(RAND_MAX+1.0);
    printf("Random number: %d\n",x);
    return 0;
}
```

Output:

```
Random number: 9
Random number: 4
```

Don't use the LSB's, ie: don't do: `(rand()%10)+1.0`.

Outline EEEE1042 C Lecture 5:

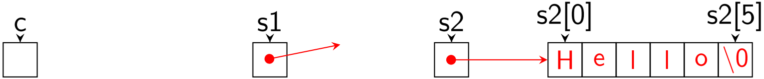
1. Functions
   - Call by value vs call by reference
   - Recursion
2. Generating Random Numbers
3. String functions in C
   - Strings Chars and Pointers
   - C string library

Refresher on Strings Chars and Pointers

- In C, strings are just arrays of chars.

```
char c;             // c is a single char
char *s1;          //s1 is a pointer to a char
char s2[]="Hello"; //s2 is an array of 6 chars.
```

- A pointer is a **memory address pointing** to a location in memory. s1 is a pointer (because of the *), it holds an **address in memory**. *s1 refers to the contents of the memory address pointed to by s1.



c is a char that the compiler uses to refer to 1 byte of memory, interpreted as a char.

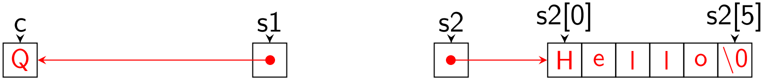
s1 is a pointer to a char. It consumes 4 or 8 bytes of memory. When declared, but uninitialized it is pointing randomly.

s2 is a pointer (because of the []) that is declared and initialized. 6 bytes of memory are allocated and filled with the string "Hello\0". `printf("%s",s2);` is valid. `printf("%s",s1);` has undefined behaviour

Strings Chars and Pointers

- In C, strings are just arrays of chars.

```
char c;              // c is a single char
char *s1;           //s1 is a pointer to a char
char s2[]="Hello"; //s2 is an array of 6 chars.
c='Q'; //Assign a value to the memory of c
s1=&c; // store the address of c into s1
```



On assignment, the compiler puts the value into the memory associated with c

On assignment, the **address** of c (denoted by &c) is stored in s1. Now *s1 is an alias s1[0] is an alias for c

s2 is a pointer (because of the []) that is declared and initialized. 6 bytes of memory are allocated and filled with the string "Hello\0". printf("%s",s2); is valid.

printf("%s",s1) is now valid, but the string pointed to by s1 is unterminated: There is no '\0' character terminating the char c which only has 8 bits allocated.

Strings Chars and Pointers: Example

```
#include <stdio.h>
int main () {
   char c;   // c is a single char
   char *s1;  //s1 is a pointer to a char
   char s2[]="Hello";  //s2 is a pointer to an array of 6 chars.
   c=’Q’; //Assign a value to the memory of c
   s1=&c; // Assign the address of c to memory of s1
   printf("Before: c=%c\n",c);  //Print c as char
   printf("Before: *s1=%c\n",*s1);//Print s1 as char
   printf("Before: s1=%s\n",s1);  //Print s1 as string
   *s1=’W’;// reassign the value of *s1 (alias for c)
   printf("After: c=%c\n",c); //Print c as char
   printf("After: *s1=%c\n",*s1);//Print s1 as char
   return(0);
}
```

Strings Chars and Pointers: Example

```
#include <stdio.h>
int main () {
   char c;   // c is a single char
   char *s1;  //s1 is a pointer to a char
   char s2[]="Hello"; //s2 is a pointer to an array of 6 chars.
   c=’Q’; //Assign a value to the memory of c
   s1=&c; // Assign the address of c to memory of s1
   printf("Before: c=%c\n",c);  //Print c as char
   printf("Before: *s1=%c\n",*s1);//Print s1 as char
   printf("Before: s1=%s\n",s1); //Print s1 as string
   *s1=’W’;// reassign the value of *s1 (alias for c)
   printf("After: c=%c\n",c); //Print c as char
   printf("After: *s1=%c\n",*s1);//Print s1 as char
   return(0);
}
```

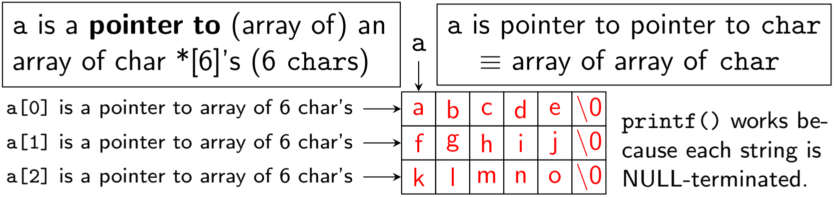
```
Before: c=Q
Before: *s1=Q
Before: s1=Q(rubbish text)
After: c=W
After: *s1=W
```

2D string arrays

```c
#include <stdio.h>
int main () {
   // Declare and initialize 2D char array
   char a[3][6]={"abcde","fghij","klmno"};
   printf("a[0]=%s\n",a[0]);
   printf("a[1]=%s\n",a[1]);
   printf("a[2]=%s\n",a[2]);
   return 0;
}
```

Output:

```
a[0]=abcde
a[1]=fghij
a[2]=klmno
```



a is a **pointer to** (array of) an array of char *[6]'s (6 chars)

a is pointer to pointer to char ≡ array of array of char

a

a[0] is a pointer to array of 6 char's →

a[1] is a pointer to array of 6 char's →

a[2] is a pointer to array of 6 char's →

a	b	c	d	e	\0
f	g	h	i	j	\0
k	l	m	n	o	\0

printf() works because each string is NULL-terminated.

a[0][0] is the first element in the array a[0]='a'

a[1][0] is the first element in the array a[1]='f'

a[2][0] is the first element in the array a[2]='k'

Initializing 2D int arrays

```
#include <stdio.h>
int main () {
    //How to initialize a 2D int array:
    int a[3][4]={
            {0, 1, 2, 3} , /* initializers for a[0] */
            {4, 5, 6, 7} , /* initializers for a[1] */
            {8, 9, 10, 11} /* initializers for a[2] */
    };
    printf("a[0][0]=%d\n",a[0][0]);
    printf("a[1][1]=%d\n",a[1][1]);
    printf("a[2][2]=%d\n",a[2][2]);
    printf("a[2][3]=%d\n",a[2][3]);
    printf("a[3][3]=%d\n",a[3][3]);
    return 0;
}
```

Initializing 2D int arrays

```
#include <stdio.h>
int main () {
    //How to initialize a 2D int array:
    int a[3][4]={
           {0, 1, 2, 3} , /* initializers for a[0] */
           {4, 5, 6, 7} , /* initializers for a[1] */
           {8, 9, 10, 11} /* initializers for a[2] */
    };
    printf("a[0][0]=%d\n",a[0][0]);
    printf("a[1][1]=%d\n",a[1][1]);
    printf("a[2][2]=%d\n",a[2][2]);
    printf("a[2][3]=%d\n",a[2][3]);
    printf("a[3][3]=%d\n",a[3][3]);
    return 0;
}
```

Output:

```
    a[0][0]=0
    a[1][1]=5
    a[2][2]=10
    a[2][3]=11
```

Example: Printing/processing 2D int array

```c
#include <stdio.h>
int main () {
    //How to print a 2D int array:
    int a[3][4]={
              {0, 1, 2, 3} , /* initializers for a[0] */
              {4, 5, 6, 7} , /* initializers for a[1] */
              {8, 9, 10, 11} /* initializers for a[2] */
    };
    int i,j;
    for (i=0;i<3;i++) {
        for (j=0;j<4;j++) {
            printf(" %2d",a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Example: Printing/processing 2D int array

```
#include <stdio.h>
int main () {
    //How to print a 2D int array:
    int a[3][4]={
            {0, 1, 2, 3} , /* initializers for a[0] */
            {4, 5, 6, 7} , /* initializers for a[1] */
            {8, 9, 10, 11} /* initializers for a[2] */
    };
    int i,j;
    for (i=0;i<3;i++) {
        for (j=0;j<4;j++) {
            printf(" %2d",a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
  0 1 2 3
  4 5 6 7
  8 9 10 11
```

Example: Printing/processing 2D char array

```
#include <stdio.h>
int main () {
    //How to print a 2D int array:
    int i,j;
    for (i=0;i<5;i++) {
        for (j=0;j<5;j++) {
            printf(" %c",’a’+i*5+j);
        }
        printf("\n");
    }
    return 0;
}
```

Example: Printing/processing 2D char array

```
#include <stdio.h>
int main () {
    //How to print a 2D int array:
    int i,j;
    for (i=0;i<5;i++) {
        for (j=0;j<5;j++) {
            printf(" %c",’a’+i*5+j);
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
 a b c d e
 f g h i j
 k l m n o
 p q r s t
 u v w x y
```

Example: Printing/processing 2D char array

```
#include <stdio.h>
int main () {
    //How to print a 2D int array:
    int i,j;
    for (i=0;i<5;i++) {
        for (j=0;j<5;j++) {
            printf(" %c",’a’+i*5+j);
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
    a b c d e
    f g h i j
    k l m n o
    p q r s t
    u v w x y
```

Note: that while the previous example used a 2D array of ints, this current example does not use a 2D array of chars, it is calculating each char directly from the indices `i` and `j`.

C string library

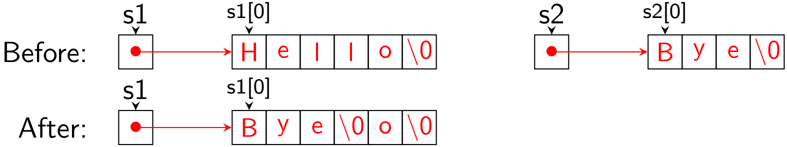
In C, as per the POP programming paradigm, there is a library of functions for handling strings. To access the C-string library in C, you need to `#include<string.h>`

The common C-string functions are listed here:

`strcpy(s1, s2);`	Copies string s2 into string s1.
`strcat(s1, s2);`	Concatenate string s2 onto the end of s1.
`strlen(s1);`	Returns the length of s1.
`strcmp(s1, s2);`	Returns 0 if s1 is same as s2. Returns +1 or -1 if s2 is greater or less than s1
`strchr(s1, ch);`	Returns pointer to first occurence of character ch in string s1.
`strstr(s1, s2);`	Returns pointer to first occurence of string s2 in string s1.

Example: strcpy(char *s1, char *s2);

Copy string s2 into string s1. s1 must hold sufficient memory.



Before: s1 → s1[0]: | H | e | l | l | o | \0 | s2 → s2[0]: | B | y | e | \0 |

After: s1 → s1[0]: | B | y | e | \0 | o | \0 |

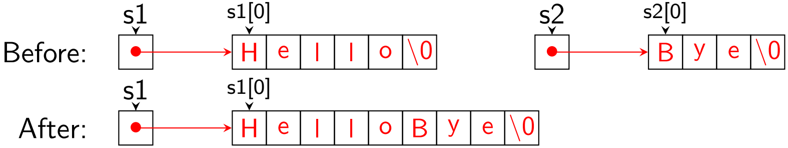
```
#include<stdio.h>
#include<string.h>
int main () {
    char s1[]="Hello";
    char s2[]="Bye";
    strcpy(s1,s2);
    printf("s1=%s\n",s1);
    return 0;
}
```

Output:

```
s1=Bye
```

Example: strcat(char *s1, char *s2);

Concatenate s2 onto end of s1. s1 must hold enough memory.



Before: s1 → s1[0]: H e l l o \0 s2 → s2[0]: B y e \0

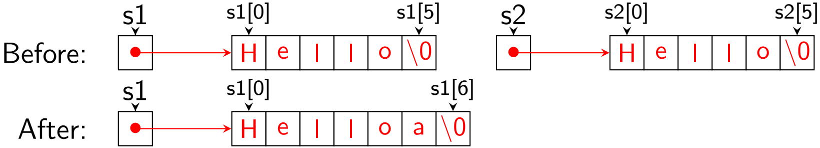
After: s1 → s1[0]: H e l l o B y e \0

```
#include<stdio.h>
#include<string.h>
int main () {
    char s1[10]="Hello";
    char s2[]="Bye";
    strcat(s1,s2);
    printf("s1=%s\n",s1);
    return 0;
}
```

Output:

```
s1=HelloBye
```

Example: strcmp(char *s1, char *s2);



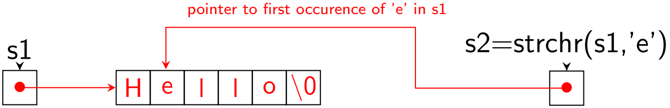
```
#include<stdio.h>
#include<string.h>
int main () {
   char s1[10]="Hello";
   char s2[]="Hello";
   printf("Before strcat=%d\n",strcmp(s1,s2));
   strcat(s1,"a");
   printf("After strcat=%d\n",strcmp(s1,s2));
   return 0;
}
```

Output:

```
Before strcat=0
After strcat=97
```

Example: strchr(char *s1, char c);

Search for first occurence of character ’e’ in s1



pointer to first occurence of ’e’ in s1

s1

H | e | l | l | o | \0

s2=strchr(s1,’e’)

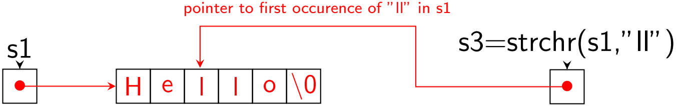
```
#include<stdio.h>
#include<string.h>
int main () {
   char s1[10]="Hello";
   char *s2=strchr(s1,’e’);
   printf("s2=%s\n",s2);
   return 0;
}
```

Output:

```
s2=ello
```

Example: strstr(char *s1, char* s2);

Search for first occurence of string ”ll” in s1



pointer to first occurence of ”ll” in s1

s1

H e l l o \0

s3=strchr(s1,”ll”)

```
#include<stdio.h>
#include<string.h>
int main () {
   char s1[10]="Hello";
   char *s3=strstr(s1,"ll");
   printf("s3=%s\n",s3);
   return 0;
}
```

Output:

```
s3=llo
```

External resources

Sites to learn about C functions:

- www.tutorialspoint.com
- www.programiz.com
- tutorialsclass.com
- www.w3schools.in

Numerical recipes in C, Chapt 7 for random numbers:

- Numerical Recipes in C

Some external sites you can go to learn about C strings:

- www.tutorialspoint.com
- www.programiz.com
- www.programiz.net
- www.cprogramming.com
- www.tutorialkart.com