University of Nottingham
UK | CHINA | MALAYSIA

EEEE1042 - Lecture 3
Strings, Data variables, input/output
Autumn Semester 2022.

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.

Week	Dates	Lecture	EEEE1042 Practical	EEEE1032 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%)			

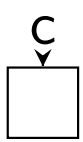
Outline EEEE1042 C Lecture 3:

Strings Chars and Pointers

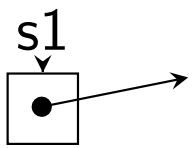
- In C, strings are just arrays of chars.

```
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.
```

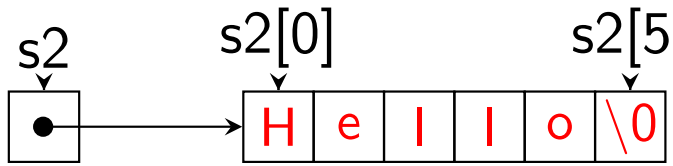
- A pointer is a **memory address pointing** to a location in memory. `s1` is a pointer (because of the `*`), it holds an **address in memory**. `*s1` refers to the contents of the memory address pointed to by `s1`.



`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.

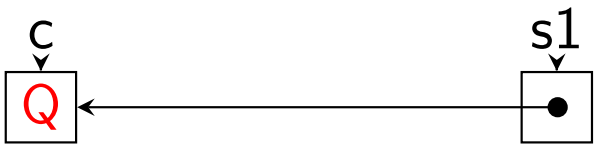


`s2` is a pointer (because of the []) that is declared and initialized. 6 bytes of memory are allocated and filled with the string "Hello\0". `printf("%s",s2);` is valid. `printf("%s",s1);` has undefined behaviour

Strings Chars and Pointers

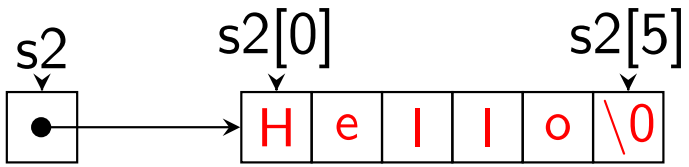
- In C, strings are just arrays of chars.

```
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


```c
#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

```c
#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


```c
#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

```c
#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
```

Outline EEEE1042 C Lecture 3:

Converting Farenheit to Celcius exercise

```c
#include <stdio.h>
/* Code to convert Farenheit to Celcius*/
int main (int argc, char **argv) {
    float F;
    float C;
    printf("\tF\tC\n");
    F=0; C=5/9*(F-32);printf("%9.0f %8.2f\n",F,C);
    F=20; C=5/9*(F-32);printf("%9.0f %8.2f\n",F,C);
    F=40; C=5/9*(F-32);printf("%9.0f %8.2f\n",F,C);
    F=60; C=5/9*(F-32);printf("%9.0f %8.2f\n",F,C);
    F=80; C=5/9*(F-32);printf("%9.0f %8.2f\n",F,C);
    F=100; C=5/9*(F-32);printf("%9.0f %8.2f\n",F,C);
    return(0);
}
```

C=5/9*(F-32)

```
F         C
0     -0.00
20    -0.00
40     0.00
60     0.00
80     0.00
100    0.00
```

C=5.0/9.0*(F-32)

```
F         C
0    -17.78
20    -6.67
40     4.44
60    15.56
80    26.67
100   37.78
```

Type Casting

Type casting is changing variables of one type (eg: int) to be **temporarily** interpreted as of another type (eg: float).
There are two categories of type-casting:

1. Explicit type-casting
   The new variable type is cast explicitly:

```
int x=3;
float y;
y=(float)x; // Typecast x as a float, store in y
            // x itself remains unchanged as an int
```

2. Implicit type-casting

```
printf("%d\n",sizeof(char));
```

   The `printf()` conversion character "%d" expects an integer.
   However `sizeof` returns a long unsigned int.
   The compiler can type-cast the `long unsigned int` into an `int` implicitly before passing as input to the `printf()` function.

Type Casting example 1

```c
#include <stdio.h>
int main () {
   //Examples of typecasting:
   char a,c=81; // 'Q'
   printf("c (as char)=%c\n",c);
   printf("c (as dec )=%d\n",(int)c);//Explicit typecast
   a=c+200;
   printf("a = c+200 =%d\n",a);
   printf("c+200 =%d\n",c+200);//Implicit typecast
   return(0);
}
```

Output:

```
   c (as char)=Q
   c (as dec )=81
   a = c+200 =25
   c+200 =281
```

Type Casting example 2

```c
#include <stdio.h>
int main () {
   int sum = 17, count = 5;
   float mean;
   mean = (float) sum / count; // Explicit typecast
   printf("mean with typecast : %f\n", mean );
   mean = sum / count; // No typecast
   printf("mean without typecast : %f\n", mean );
   return(0);
}
```

Output:

```
   mean with typecast : 3.400000
   mean without typecast : 3.000000
```

Outline EEEE1042 C Lecture 3:

Taking inputs from the command line

The two inputs to the main function are:

1. `int argc;` Number of parameters at the command line
2. `char **argv;` Array of input strings from command line

```
#include <stdio.h>
int main (int argc, char **argv) {
   printf("argc=%d\n",argc);
   printf("Zero'th input argv[0]=%s\n",argv[0]);
   printf("First input argv[1]=%s\n",argv[1]);
   printf("Second input argv[2]=%s\n",argv[2]);
   return(0);
}
```

Output:

```
$ ./helloWorld This is a test
   argc=5
   Zero'th input argv[0]=./helloWorld
   First input argv[1]=This
   Second input argv[2]=is
```

Manoevering the command line

- C was written at a time before **graphical user interfaces** (GUI's) were common-place.
- C is programmed for the **command-line interface** (CLI).
- Code::Blocks manages the files/directories for the CLI for you.
- But you should understand basics of how to manoever in the CLI.

Linux	Windows	Description	Example
cd	cd	change directory	`cd bin; cd ..;`
ls	dir	list directory contents	`dir`
pwd	pwd	print working directory	`pwd`
.	.	refers to the current directory	./make
..	..	refers to the parent directory	`cd ..`

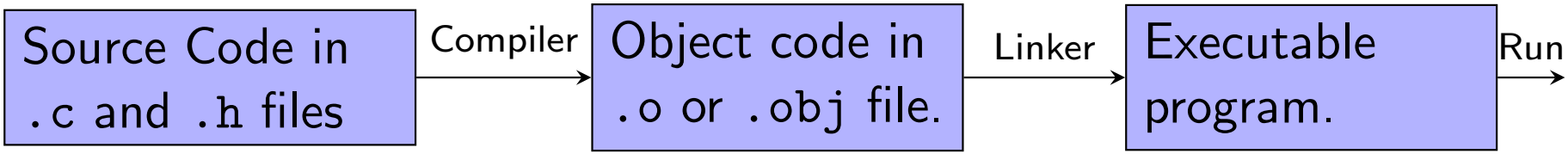
Important Note: Delimiters between directories use the backwards slash "\" in Windows and the forwards slash "/" in Linux.

Example: If you are in `bin` subdir and want to go to `obj` subdir type:

Windows: `$ cd ..\obj` , Linux: `$ cd ../obj`

Outline EEEE1042 C Lecture 3:

Preprocessor and macro expansion



- The C-preprocessor stage happens before the compilation (not shown in the figure)
- C-preprocessor directives all begin with #
- The `#define` directive cuts-and-pastes replacement of the defined macros

```c
#include <stdio.h>
#define PI 3.1415
int main (int argc, char **argv) {
    double r=4.0;
    double A=PI*r*r;
    printf("A=%lf\n",A);
    return(0);
}
```

After C-preproc

```c
#include <stdio.h>
#define PI 3.1415
int main (int argc, char **argv) {
    double r=4.0;
    double A=3.1415*r*r;
    printf("A=%lf\n",A);
    return(0);
}
```

- The `#include<stdio.h>` directive also just cuts-n-pastes the declarations from `stdio.h` into the program.

Preprocessor and Function-like macros

Preprocessor macros can also be defined with input parameters like a function.

- The preprocessor replaces the macros in the source code with the corresponding code from the function, replacing the arguments with the corresponding text.
- As a general rule preprocessor directives are always UPPER CASE to differentiate them from other C code.

```
#define MULT_TWO(x) 2*x
```

- The above preproc directive will replace the text `MULT_TWO(x)` with `2*x` throughout the source code before passing it to the compiler for any input `x`.
- So

```
int price=3;
printf("price=%d\n",MULT_TWO(price));
```

gets replaced with:

```
int price=3;
printf("price=%d\n",2*price);
```

in the source code.

Preprocessor and Function-like macros

Some care is needed when using preprocessor directives:

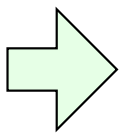
- If the preprocessor directive is #define'd as:

```
#define SQUARE(x) x*x
```

×

- The preprocessor would replace:

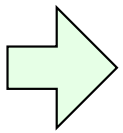
```
y=SQUARE(x);
```



```
y=x*x;
```

However the preprocessor would replace

```
z=SQUARE(x+y);
```



```
z=x+y*x+y;
```

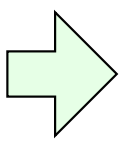
which is not what we want. Therefore you should define your directives with parentheses () around your arguments as follows:

```
#define SQUARE(x) (x)*(x)
```

✓

then

```
z=SQUARE(x+y);
```



```
z=(x+y)*(x+y);
```

Preprocessor and Function-like macros, examples:

```
#include <stdio.h>
#define SQUARE(r) (r)*(r)
int main (int argc, char **argv) {
   double x=3.0,y=4.0;
   printf("SQUARE(x)=%.1lf\n",SQUARE(x));
   printf("SQUARE(y)=%.1lf\n",SQUARE(y));
   printf("SQUARE(x+y)=%.1lf\n",SQUARE(x+y));
   return(0);
}
```

Output with:

`#define SQUARE(r) (r)*(r)`

```
SQUARE(x)=9.0
SQUARE(y)=16.0
SQUARE(x+y)=49.0
```

Output with:

`#define SQUARE(r) r*r`

```
SQUARE(x)=9.0
SQUARE(y)=16.0
SQUARE(x+y)=19.0
```

Outline EEEE1042 C Lecture 3:

Basic Input/Output: scanf

Basic output to the screen is done using `printf()` already discussed.
Basic input from the keyboard is done using `scanf()` which is the counter part to `printf()`:
`scanf(`*`formatString`*`,&`*`variables`*`,...)`

- The *formatString* is very similar to that for `printf()` with `%s` for strings, `%d` for `ints` `%f` for `floats` etc...
- The elements in the input stream must match the elements in the *formatString*.
- The **address of the variable** must be given instead of the variable itself.
  - Variable be a **pointer** to the required type.
- `scanf` returns the number of inputs read.

scanf example

```c
#include<stdio.h>
int main(int argc, char **argv) {
    char name[50];
    int age;
    float height;
    printf("Enter name: ");
    scanf("%s",name);//Name is a pointer to an array of chars
    printf("Enter age: ");
    scanf("%d", &age);//&age is a pointer to an int
    printf("Enter height: ");
    scanf("%f", &height);//&height is a pointer to a float
    printf("%s is %d years old and %.2fm tall\n",name,age,height);
    return(0);
}
```

```
$ ./helloWorld
Enter name: Davy
Enter your age: 15
Enter your height: 1.3
Davy is 15 years old and 1.30m tall
```

FILE Pointers

There are 3 data streams associated with every C program: stdin, stdout and stderr. We can open other data streams using FILE pointers.

- Files are opened/closed with the `fopen`/`fclose` command:

```c
#include<stdio.h>
int main(int argc, char **argv) {
   FILE *f; // File pointer
   char filename[]="a.txt"; // name of the file to open/close
   /* Open the file for reading */
   if ((f=fopen(filename,"r"))!=NULL) {
      fclose(f); // close the file pinter
   } else {printf("Unable to open %s for reading\n",filename);}
   /* Open the file for writing */
   if ((f=fopen(filename,"w"))!=NULL) {
      fclose(f); // close the file pointer
   } else {printf("Unable to open %s for reading\n",filename);}
}
```

`fopen` returns a pointer to the opened stream, or `NULL` on failure.

fscanf

After the file has been opened, we can read/write to the file using `fprintf` and `fscanf`, which are the **file equivalents** of `printf` and `scanf`.

- Open file for reading and `fscanf` input from there.

```c
#include<stdio.h>
int main(int argc, char **argv) {
    FILE *f; // File pointer
    char filename[]="inFile.txt"; // name of the file to open/close
    char name[50];
    int age;
    float height;
    /* Open the file for reading and read from it */
    if ((f=fopen(filename,"r"))!=NULL) {
        fscanf(f,"%s",name);//Name is a pointer to an array of chars
        fscanf(f,"%d", &age);//&age is a pointer to an int
        fscanf(f,"%f", &height);//&height is a pointer to a float
        printf("%s is %d years old and %.2fm tall\n",name,age,height);
        fclose(f); // close the file pointer
    } else {printf("Unable to open %s for reading\n",filename);}
}
```

`fscanf` returns the number of arguments scanned or the constant EOF on reaching end of the file.

fscanf

After the file has been opened, we can read/write to the file using `fprintf` and `fscanf`, which are the **file equivalents** of `printf` and `scanf`.

- Open file for reading and `fscanf` input from there.

```c
#include<stdio.h>
int main(int argc, char **argv) {
    FILE *f; // File pointer
    char filename[]="inFile.txt"; // name of the file to ope
    char name[50];
    int age;
    float height;
    /* Open the file for reading and read from it */
    if ((f=fopen(filename,"r"))!=NULL) {
        fscanf(f,"%s",name);//Name is a pointer to an array of chars
        fscanf(f,"%d", &age);//&age is a pointer to an int
        fscanf(f,"%f", &height);//&height is a pointer to a float
        printf("%s is %d years old and %.2fm tall\n",name,age,height);
        fclose(f); // close the file pointer
    } else {printf("Unable to open %s for reading\n",filename);}
}
```

inFile.txt:

```
Davy
15
1.5
```

`fscanf` returns the number of arguments scanned or the constant EOF on reaching end of the file.

fscanf

After the file has been opened, we can read/write to the file using `fprintf` and `fscanf`, which are the **file equivalents** of `printf` and `scanf`.

- Open file for reading and `fscanf` input from there.

```c
#include<stdio.h>
int main(int argc, char **argv) {
    FILE *f; // File pointer
    char filename[]="inFile.txt"; // name of the file to ope
    char name[50];
    int age;
    float height;
    /* Open the file for reading and read
    if ((f=fopen(filename,"r"))!=NULL) {
        fscanf(f,"%s",name);//Name is a pointer to an array of chars
        fscanf(f,"%d", &age);//&age is a pointer to an int
        fscanf(f,"%f", &height);//&height is a pointer to a float
        printf("%s is %d years old and %.2fm tall\n",name,age,height);
        fclose(f); // close the file pointer
    } else {printf("Unable to open %s for reading\n",filename);}
}
```

inFile.txt:

```
Davy
15
1.5
```

Output:

```
Davy is 15 years old and 1.50m tall
```

`fscanf` returns the number of arguments scanned or the constant EOF on reaching end of the file.

fprintf

After the file has been opened, we can read/write to the file using `fprintf` and `fscanf`, which are the file equivalents of `printf` and `scanf`.

- Additional code to open file for writing and `fprintf` info to.

```
/* Open the file for writing and write to it */
if ((f=fopen("outFile.txt","w"))!=NULL) {
    fprintf(f,"Name : %s\n",name);
    fprintf(f,"Age : %d\n",age);
    fprintf(f,"Height: %.2f\n",height);
    fclose(f); // close the file pointer
} else {printf("Unable to open %s for reading\n",filename);}
```

`fprintf` returns the number of characters printed on success or a negative number on failure.

fprintf

After the file has been opened, we can read/write to the file using `fprintf` and `fscanf`, which are the file equivalents of `printf` and `scanf`.

- Additional code to open file for writing and `fprintf` info to.

```
/* Open the file for writing and write to it */
if ((f=fopen("outFile.txt","w"))!=NULL) {
    fprintf(f,"Name : %s\n",name);
    fprintf(f,"Age : %d\n",age);
    fprintf(f,"Height: %.2f\n",height);
    fclose(f); // close the file pointer
} else {printf("Unable to open %s for reading\n",filename);}
```

inFile.txt:

```
Davy
15
1.5
```

Output:

```
Davy is 15 years old and 1.50m tall
```

`fprintf` returns the number of characters printed on success or a negative number on failure.

fprintf

After the file has been opened, we can read/write to the file using `fprintf` and `fscanf`, which are the file equivalents of `printf` and `scanf`.

- Additional code to open file for writing and `fprintf` info to.

```
/* Open the file for writing and write to it */
if ((f=fopen("outFile.txt","w"))!=NULL) {
    fprintf(f,"Name : %s\n",name);
    fprintf(f,"Age : %d\n",age);
    fprintf(f,"Height: %.2f\n",height);
    fclose(f); // close the file pointer
} else {printf("Unable to open %s for reading\n",filename);}
```

inFile.txt:

```
Davy
15
1.5
```

outFile.txt:

```
Name : Davy
Age : 15
Height: 1.50
```

Output:

```
Davy is 15 years old and 1.50m tall
```

`fprintf` returns the number of characters printed on success or a negative number on failure.