

C Programming Essentials

Unit 1: Sequential Programming

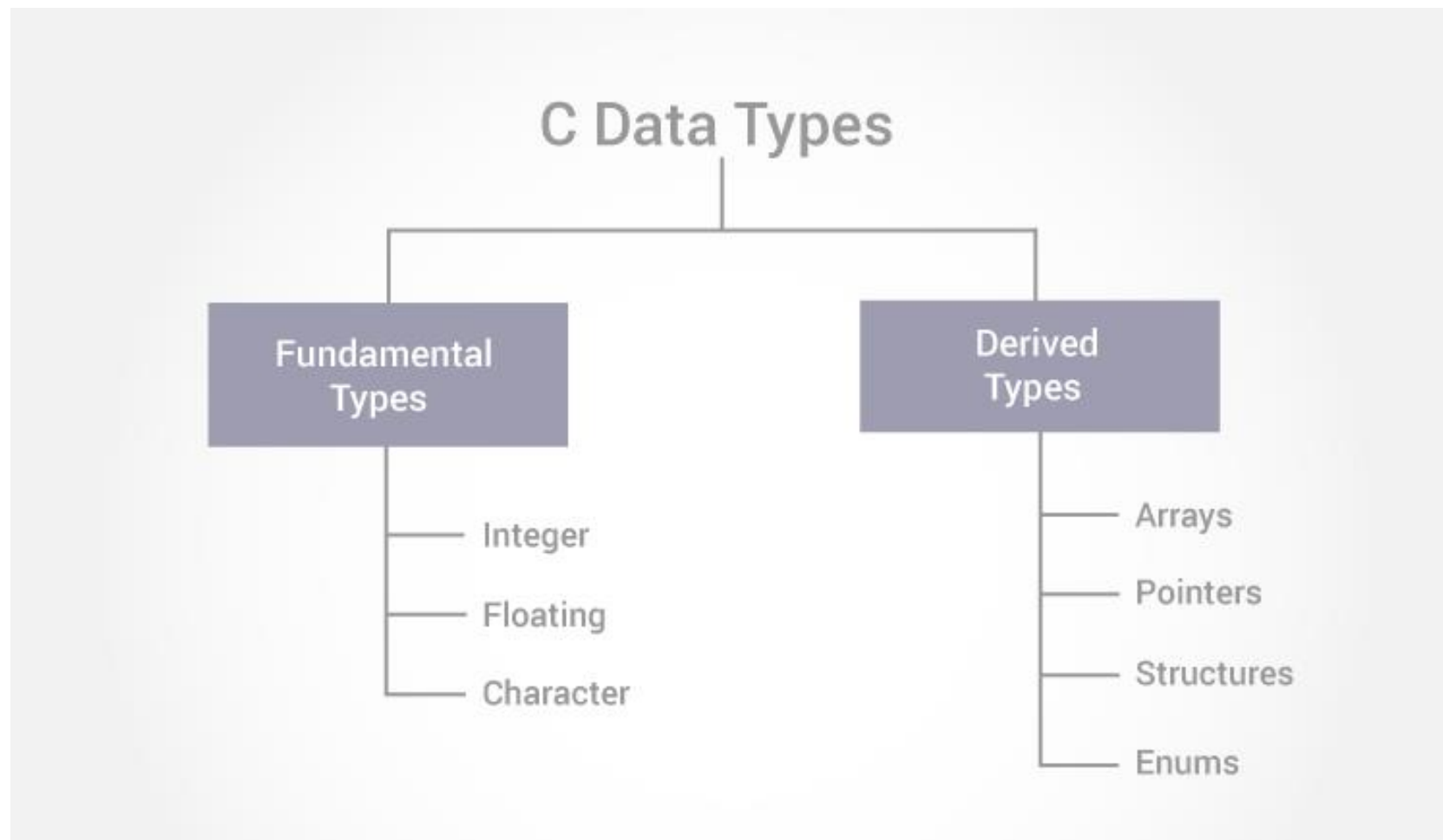
CHAPTER 3: BOOLEAN, CHARACTERS, STRINGS, BASIC DATA CONVERSION
AND BASIC I/O

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

C Programming Data Types



C Programming Data Types

Note:

In C programming, variables or memory locations should be declared before it can be used. Similarly, a function also needs to be declared before use. Data types simply refers to the type and size of data associated with [variables](#) and [functions](#).



Data types in C

1. Fundamental Data Types

1. Integer types
2. Floating type
3. Character type

2. Derived Data Types

1. Arrays
2. Pointers
3. Structures
4. Enumeration



int - Integer data types

Integers are whole numbers that can have both positive and negative values but no decimal values.
Example: 0, -5, 10

In C programming, keyword `int` is used for declaring integer variable. For example:

```
int id;
```

Here, `id` is a variable of type integer.

You can declare multiple variable at once in C programming. For example:

```
int id, age;
```

The size of `int` is either 2 bytes(In older PC's) or 4 bytes. If you consider an integer having size of 4 byte(equal to 32 bits), it can take 2^{32} distinct states as: $-2^{31}, -2^{31}+1, \dots, -2, -1, 0, 1, 2, \dots, 2^{31}-2, 2^{31}-1$

Similarly, `int` of 2 bytes, it can take 2^{16} distinct states from -2^{15} to $2^{15}-1$. If you try to store larger number than $2^{15}-1$, i.e., $+2^{14}7483647$ and smaller number than -2^{15} , i.e., $-2^{14}7483648$, program will not run correctly.

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295



float - Floating types

Floating type variables can hold real numbers such as: 2.34, -9.382, 5.0 etc. You can declare a floating point variable in C by using either float or double keyword. For example:

```
float accountBalance;
```

```
double bookPrice;
```

Here, both accountBalance and bookPrice are floating type variables.

In C, floating values can be represented in exponential form as well. For example:

```
float normalizationFactor = 22.442e2;
```

DATA TYPE	SIZE	RANGE
Float	4 bytes	$3.4e - 38$ to $3.4e + 38$
Double	8 bytes	$1.7e - 308$ to $1.7e + 308$
Long double	10 bytes	$3.4e - 4932$ to $1.1e + 4932$



Difference between float and double

The size of float (single precision float data type) is 4 bytes. And the size of double (double precision float data type) is 8 bytes. Floating point variables has a precision of 6 digits whereas the precision of double is 14 digits.



char - Character types

Keyword char is used for declaring character type variables. For example:

```
char test = 'h';
```

Here, test is a character variable. The value of test is 'h'.

The size of character variable is 1 byte.

LECTURE 2

C Qualifiers (Modifiers)



C Qualifiers

Qualifiers alters the meaning of base data types to yield a new data type.

- Size qualifiers
- Sign qualifiers
- Constant qualifiers
- Volatile qualifiers



Size qualifiers

- Size qualifiers alters the size of a basic type. There are two size qualifiers, long and short.
- For example:
`long double i;`
- The size of double is 8 bytes. However, when long keyword is used, that variable becomes 10 bytes.
- Learn more about long keyword in C programming.
- There is another keyword short which can be used if you previously know the value of a variable will always be a small number.



Sign qualifiers

- Integers and floating point variables can hold both negative and positive values. However, if a variable needs to hold positive value only, unsigned data types are used. For example:
 // unsigned variables cannot hold negative value
 unsigned int positiveInteger;
- There is another qualifier signed which can hold both negative and positive only. However, it is not necessary to define variable signed since a variable is signed by default.
- An integer variable of 4 bytes can hold data from -231 to 231-1. However, if the variable is defined as unsigned, it can hold data from 0 to 232-1.
- It is important to note that, sign qualifiers can be applied to **int** and **char** types (integer types) only.



Constant qualifiers

- An identifier can be declared as a constant. To do so **const** keyword is used.

```
const int cost = 20;
```

- The value of cost cannot be changed in the program.



Demo Program:

`constant.c`

Go gcc!!!

Note: error will occur in compilation stage.



Volatile qualifiers

- A variable should be declared volatile whenever its value can be changed by some external sources outside the program.
- Keyword volatile is used for creating volatile variables.
- To declare a variable volatile, include the keyword volatile before or after the data type in the variable definition. For instance both of these declarations will declare foo to be a volatile integer:

`volatile int foo;`

`int volatile foo;`



Proper Use of C's volatile Keyword

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

1. Memory-mapped peripheral registers
2. Global variables modified by an interrupt service routine
3. Global variables accessed by multiple tasks within a multi-threaded application

The volatile-keyword related topics are beyond this course.

LECTURE 3

Boolean Data Type

Integer as Boolean

C's integer Type is Boolean data type. There is no difference between integer and boolean value.

Non-zero integers means true.

Zero means false.

Option 1

```
typedef int bool;  
#define true 1  
#define false 0
```

Option 2

```
typedef int bool;  
enum { false, true };
```

Option 3

```
typedef enum { false, true } bool;
```

Option 4 (C99)

```
#include <stdbool.h>
```

Booleans in C

```
#include <stdbool.h>

bool  bool1  = true;
bool  bool2  = false;
```

Important!
Compiler needs this or it
won't know about "bool"!

bool added to C in 1999

Many programmers had already defined their own Boolean type

- **To avoid conflict `bool` is disabled by default**

bool and _Bool data types

- These data types were added in C99. Since bool wasn't reserved prior to C99, they use the **_Bool** keyword (which was reserved).
- **bool** is an alias for **_Bool** if you include stdbool.h.
- Basically, including the stdbool.h header is an indication that your code is OK with the identifier bool being 'reserved', i.e. that your code won't use it for its own purposes (similarly for the identifiers true and false).
- **_Bool** does not need inclusion. **bool** needs to include <stdbool.h>



Logical Expressions

Formed using

- 4 relational operators: `<`, `<=`, `>`, `>=`
- 2 equality operators: `==`, `!=`
- 3 logical connectives: `&&`, `||`, `!`

`int` type: 1(true) or 0 (false)

Some examples are

- If `x = 8`, `y = 3`, `z = 2` what is the value of
`x >= 10 && y < 5 || z == 2`

Logical Expressions

Formed using

- 4 relational operators: `<`, `<=`, `>`, `>=`
- 2 equality operators: `==`, `!=`
- 3 logical connectives: `&&`, `||`, `!`

`int` type: 1(true) or 0 (false)

Some examples are

- If `x = 8`, `y = 3`, `z = 2` the value of

`x >= 10 && y < 5 || z == 2` is 1.

- **Precedence comes into picture. Remember last lecture?**



Demo Program:

[debug.c/debug2.c](#)

Go gcc!!!



Standard Boolean Library in C99

`#include <stdbool.h>`

The header **stdbool.h** in the C Standard Library for the C programming language contains four macros for a Boolean data type. This header was introduced in C99.

The macros as defined in the ISO C standard are :

- **bool** which expands to `_Bool`
- **true** which expands to `1`
- **false** which expands to `0`
- **__bool_true_false_are_defined** which expands to `1`

Defined in header <stdlib.h>

```
#define EXIT_SUCCESS /*implementation defined*/  
#define EXIT_FAILURE /*implementation defined*/
```

The EXIT_SUCCESS and EXIT_FAILURE macros expand into integral expressions that can be used as arguments to the `exit` function (and, therefore, as the values to return from the `main function`), and indicate program execution status.

Constant	Explanation
EXIT_SUCCESS	successful execution of a program
EXIT_FAILURE	unsuccessful execution of a program

Notes

Both EXIT_SUCCESS and the value zero indicate successful program execution status (see `exit`), although it is not required that EXIT_SUCCESS equals zero.

EXIT_SUCCESS, EXIT_FAILURE in stdlib.h



Demo Program:

keepgoing.c

Go gcc!!!

LECTURE 4

Character Data type

Character Representation In C Language

C data types for characters:

`char`

`signed char`

`unsigned char`

`uint8_t` (`stdint.h`)

ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters		
32	space	
33	!	
34	"	
35	#	
36	\$	
37	%	
38	&	
39	'	
40	(
41)	
42	*	
43	+	
44	,	
45	-	
46	.	
47	/	
48	0	
49	1	
50	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	
59	;	
60	<	
61	=	
62	>	
63	?	
64	@	
65	A	
66	B	
67	C	
68	D	
69	E	
70	F	
71	G	
72	H	
73	I	
74	J	
75	K	
76	L	
77	M	
78	N	
79	O	
80	P	
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
90	Z	
91	[
92	\	
93]	
94	^	
95	_	
96	`	
97	a	
98	b	
99	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	
124		
125	}	
126	~	

Extended ASCII characters							
128	Ç	160	á	192	Ł	224	Ó
129	ü	161	í	193	ł	225	ô
130	é	162	ó	194	Ť	226	Ô
131	â	163	ú	195	ŧ	227	Õ
132	ä	164	ñ	196	—	228	ö
133	à	165	Ñ	197	†	229	Õ
134	á	166	ª	198	ã	230	μ
135	ç	167	º	199	Ä	231	þ
136	ê	168	¿	200	Ł	232	Ɔ
137	ë	169	®	201	Œ	233	Ù
138	è	170	¬	202	ℒ	234	Ú
139	ï	171	½	203	Ť	235	Û
140	î	172	¼	204	Ŧ	236	ý
141	ì	173	í	205	=	237	Ý
142	Ä	174	«	206	≠	238	—
143	Å	175	»	207	□	239	‘
144	É	176	⋮	208	ð	240	≡
145	æ	177	⋮	209	Ð	241	±
146	Æ	178	⋮	210	Ê	242	≡
147	ô	179	⋮	211	Ë	243	¼
148	ö	180	⋮	212	È	244	¶
149	ò	181	À	213	Ì	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	ˆ
152	ÿ	184	©	216	Ï	248	˚
153	Ö	185	¶	217	Ɔ	249	ˆ
154	Ü	186	¶	218	Ɔ	250	ˆ
155	ø	187	¶	219	Ɔ	251	ˆ
156	£	188	¶	220	Ɔ	252	ˆ
157	Ø	189	¢	221	Ɔ	253	ˆ
158	×	190	¥	222	Ɔ	254	■
159	f	191	Ɔ	223	Ɔ	255	nbsp



Character Data Type in C

- To declare a variable of type character we use the keyword char. - A single character stored in one byte.

- For example:

```
char c;
```

- To assign, or store, a character value in a char data type is easy - a character variable is just a symbol enclosed by single quotes. For example, if `c` is a char variable you can store the letter `A` in it using the following `C` statement:

```
c = 'A';
```

- `c` is an char-size integer of 65 (0x41) and a character 'A' at the same time. char-size 0 value will be a empty character. It is also used as the ending mark of an character array.

Data Types

char x = 12;

This means that x is:

- signed
- an integer
- 8 bits wide

In other words:

$$-2^7 \leq x \leq 2^7 - 1$$

uint8_t x = 12;

This means that x is:

- unsigned
- an integer
- 8 bits wide

In other words:

$$0 \leq x \leq 2^8 - 1$$



Char, unsigned char, and uint8_t

- `char` and `unsigned char` are primitive data type in C language.
- `char` is 8-bit signed integer. `unsigned char` is 8-bit unsigned integer.
- `uint8_t` is defined in `stdint.h`
- `uint8_t` is defined as `unsigned char` type in `stdint.h`
- `uint8_t` documents your intent - you will be storing small numbers, rather than a character. Or, in other words, `uint8_t`, `uint16_t`, `uint32_t` are a group of data types to support UTF8, UTF16, UTF32 for Unicode.

Specifier	Common Equivalent	Signing	Bits	Bytes	Minimum Value	Maximum Value
int8_t	signed char	signed	8	1	-128	127
uint8_t	unsigned char	unsigned	8	1	0	255
int16_t	short	signed	16	2	-32,768	32,767
uint16_t	unsigned short	unsigned	16	2	0	65,535
int32_t	long	signed	32	4	-2,147,483,648	2,147,483,647
uint32_t	unsigned long	unsigned	32	4	0	4,294,967,295
int64_t	long long	signed	64	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	unsigned long long	unsigned	64	8	0	18,446,744,073,709,554,615

Data Types in stdint.h



Array and for-loop (Simple Introduction)

```
char s[6] = "Hello";
```

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]
h	e	l	l	o	\0

```
int i=0;
```

```
for (i=0; i<6; i++) printf("%c", s[i]);
```

```
printf("\n");
```

LECTURE 5

Strings in C



Character and String

C only has a concept of numbers and characters. It very often comes as a surprise to some programmers who learnt a beginner's language such as BASIC that C has no understanding of strings but a string is only an array of characters and C does have a concept of arrays which we shall be meeting later in this course.

Notice that you can only store a single character in a char variable. Later we will be discussing using character strings, which has a very real potential for confusion because a string constant is written between double quotes. But for the moment remember that a char variable is 'A' and not "A".



Array of Characters

To declare an array of characters:

```
char a[] = {'a', 'b', 'c'};
```

Or,

```
char a[] = {'d', 'e', 'f'};
```

```
char *b = a;
```

These statements declared and initialized arrays of characters. They represent a collection of characters in the format of arrays. But, they are not strings. String operations don't apply to them.



Array of Characters

`char *b = calloc(1, sizeof(char)); // this is legal`

`char *b = {'a', 'b', 'c', 'd'}; // this is not legal. rvalue is of char[] type`

`char a[]; // a is address of array of character type`

`char *a; // a's body is of char type.`

These two both hold the address value of the array of characters. They can be used interchangeably in many places. But, not always. Especially, when the literals of array of character are used.



Demo Program:

[charAry.c](#)

Go gcc!!!

Strings are Special Array of Characters

In C, there is no built-in String type.

- Strings are actually one-dimensional array of characters terminated by a null character '\0'.
- Thus a null-terminated string contains the characters that comprise the string followed by a null.

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

These are all strings:

```
char a[] = "Hello";
```

```
char *b = "Hello";
```

```
char c[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

A string needs to be terminated by NULL.

These are not strings by the definition of <string.h>:

```
Char d[] = {'3', '3', '3'};
```



Initialization of String

Character Array Initialized Using String Literals:

```
char a[] = {'h', 'o', 'm', 'e', '\0'}; // by character array literal
```

```
char *a = calloc(strlen(SIZE_STRING)+1, sizeof(char)); // "home", SIZE_STRING=4
```

```
// cleaned up with 0.
```

```
// initialized by string copy, string element updates.
```

```
char *a = "home";
```

Note:

`char a[];` // declaration of an array name. `a` is a reference to an array.

`Char *b;` // declaration of a pointer to char. `b` is a pointer to an array

These two are compatible types but not of the same type. `b`, `a`, `&a[0]` are of the same value.



Initialization of String

Character Array Initialized Using Console Input:

```
int i;  
char mystring[10];  
for (i=0; i<10; i++) scanf("%c", &mystring[i]);  
c[9] = '\0';  
printf("%s", mystring);
```

Using Input Functions:

scanf(), gets(), getchar()



Common Error about Strings

The following results in an error.

1. `char str1[] = "Hello";` // "Hello" is a rvalue of pointer type.
2. `char str1[6];`
`str1 = "Hello";`
3. `char str1[6] = "Hello";`
`char str2[6];`
`str2 = str1;`

LECTURE 6

Console Input

1

`getchar()`: get a character

2

`scanf()`: get a token (formatted)

3

`gets()`: get a line of string with spaces

Input Functions

Input Function

- The `scanf()` Function

- header file `stdio.h`

- Syntax:

```
char mystring[100];  
scanf("%s", mystring);
```

- The name of a string is a pointer constant to the first character in the character array.

- Problem:

terminates its input on the first white space it finds.

white space includes blanks, tabs, carriage returns(CR), form feeds & new line.



Demo Program:

Iscanf.c

Go gcc!!!

```
1  #include <stdio.h>
2
3  char mystring[100];
4
5  main(){
6      printf("Enter String:");
7      scanf("%s", mystring);
8      printf("\n\n%s", mystring);
9  }
```

```
C:\Eric_Chou\C Course\CDev\Ch3\input>gcc Iscanf.c -o Iscanf
```

```
C:\Eric_Chou\C Course\CDev\Ch3\input>Iscanf
Enter String:I am going to school.
```

```
I
```


Input Function

- The `gets()` Function
 - Header file `stdio.h`
 - takes a string from standard input and assigns it to a character array.
 - It replaces the `\n` with `\0`.
 - Syntax:

```
char mystring[100];  
gets(myString);
```

`fgets()` it keeps the `\n` and includes it as part of the string.



Demo Program:

`lgets.c`

Go gcc!!!

```
1  #include <stdio.h>
2
3  char mystring[100];
4
5  main(){
6      printf("Enter String:");
7      gets(mystring);
8      printf("\n\n%s", mystring);
9  }
```

```
C:\Eric_Chou\C Course\CDev\Ch3\input>gcc Igets.c -o Igets
```

```
C:\Eric_Chou\C Course\CDev\Ch3\input>Igets
Enter String:I am going to school.
```

```
I am going to school.
```

```
C:\Eric Chou\C Course\CDev\Ch3\input>
```

Input Function

- The `getchar()` Function
 - Takes single character at a time.
 - Syntax:

```
char mychar;  
mychar=getchar();
```

It can use to read each character of an string.

Example
:

```
int i;  
char mystring[100];  
printf("Enter String:\n");  
for(i=0;i<10;i++)  
    mystring[i]=getchar();  
mystring[9]='\0';  
printf("\n\n%s",mystring);
```



Demo Program:

lgetchar.c

Go gcc!!!

```
1  #include <stdio.h>
2  char mystring[100];
3
4  main(){
5      int i;
6      printf("Enter String:");
7      for (i=0; i<100; i++) mystring[i] = getchar();
8      mystring[9] = '\0';
9      printf("\n\n%s", mystring);
10 }
```

```
C:\Eric_Chou\C Course\CDev\Ch3\input>gcc Igetchar.c -o Igetchar
```

```
C:\Eric_Chou\C Course\CDev\Ch3\input>Igetchar
```

```
Enter String:Hello Boy
```

```
Hello Boy
```

```
C:\Eric_Chou\C Course\CDev\Ch3\input>_
```

LECTURE 7

Console Output



Output Functions

`putchar()`: print a character

`printf()`: formatted print out

`puts()`: get a line of string with spaces

Output function

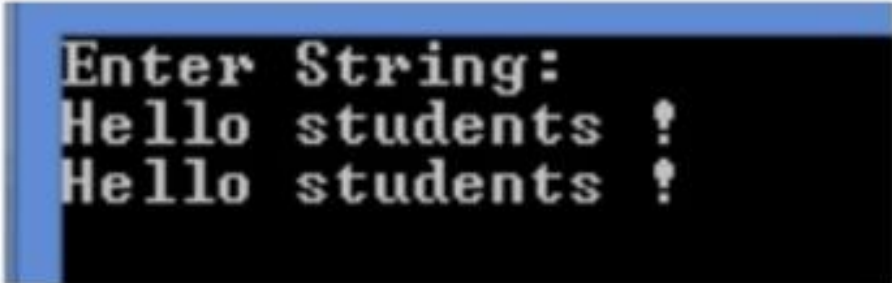
- The `printf ()` function

- header file `stdio.h`
(self study)

- The `puts ()` function

- header file `stdio.h`

```
char mystring[100];  
printf("Enter String:\n");  
gets(mystring);  
puts(mystring);
```



```
Enter String:  
Hello students !  
Hello students !
```



Demo Program:

lputs.c

Go gcc!!!

```
1  #include <stdio.h>
2  char mystring[100];
3
4  main(){
5      printf("Enter String: ");
6      gets(mystring);
7      puts(mystring);
8  }
```

```
C:\Eric_Chou\C Course\CDev\Ch3\output>gcc Iputs.c -o Iputs
C:\Eric_Chou\C Course\CDev\Ch3\output>Iputs
Enter String:
Hello Boy
Hello Boy
C:\Eric_Chou\C Course\CDev\Ch3\output>_
```


Output function

- The `putchar()` Function

- output single character at a time.
- Syntax:

```
char mychar;  
mychar=getchar();
```

It can use to read each character of an string.

Example

```
int i;  
char mystring[100];  
printf("Enter String:\n");  
gets(mystring);  
for(i=0;i<20;i++)  
    putchar(mystring[i]);
```



Demo Program:

lputchar.c

Go gcc!!!

```
1  #include <stdio.h>
2  char mystring[100];
3
4  main(){
5      int i;
6      int done = 0;
7      printf("Enter String: ");
8      gets(mystring);
9      for (i=0; i<100 && !done; i++){
10         if (mystring[i] == '\0'){
11             done = 1;
12         }
13         else putchar(mystring[i]);
14     }
15 }
```

```
C:\Eric_Chou\C Course\CDev\Ch3\output>gcc lputchar.c -o lputchar
C:\Eric_Chou\C Course\CDev\Ch3\output>lputchar
Enter String: I am going to home.
I am going to home.
C:\Eric_Chou\C Course\CDev\Ch3\output>_
```

LECTURE 8

Math Library



Math Library

`#include <math.h>`

- The C and C++ programming languages have a large collection of utility functions built into the language which you can access by including header files in your code using the `#include` preprocessor directive. One of the most useful of these for performing math calculations is the math library.
- To access all of these functions you must include the following line at the top of the source file where you want to use the math functions.

`#include<math.h>`

- The above line is standard **C** syntax. You may use this or the standard **C++** syntax form below:

`#include<cmath>`

`using namespace std;`

Polynomial Function and its Derivative

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/file.h>
4  #include <math.h>

```

Console I/O

FILE I/O

Mathematical Library

```

6  #define LEN 3 // length of the coefficient array
7  double c[] = {1, -2, 1}; // polynomial 1 - 2x + x^2
8  /**
9   * x: double variable
10  * coeff: coefficient array a[0] + a[1]*x^1 + a[2]*x^2 + ... + a[n-1]*x^(n-1)
11  */
12 double poly(double x, double coeff[]){
13     double sum=0.0;
14     for (int i=0; i<LEN; i++){
15         sum += coeff[i] * pow(x,i);
16     }
17     return sum;
18 }

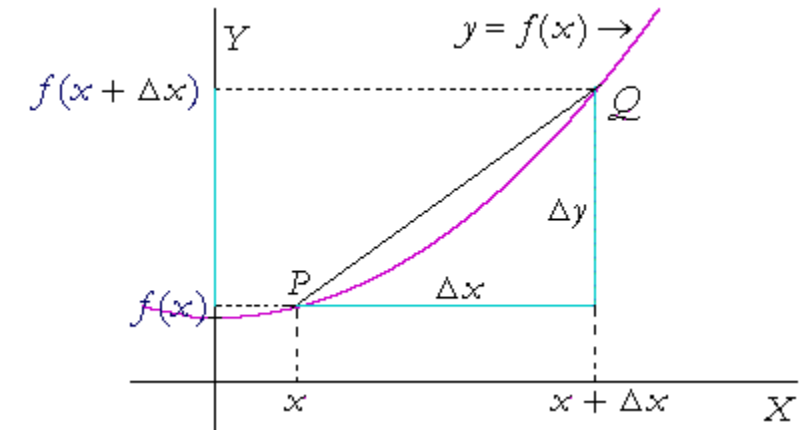
```

Mathematical power function

```

20  /*
21   * diff: calculation of derivatives
22   */
23  double diff(double yh, double y, int steps){
24     double dt = (double) 1/steps;
25     return (yh - y) * steps; // take derivatives
26 }

```



$$f'(x) = \frac{f(x+h) - f(x)}{h}$$

$$= (yh - y) * steps$$

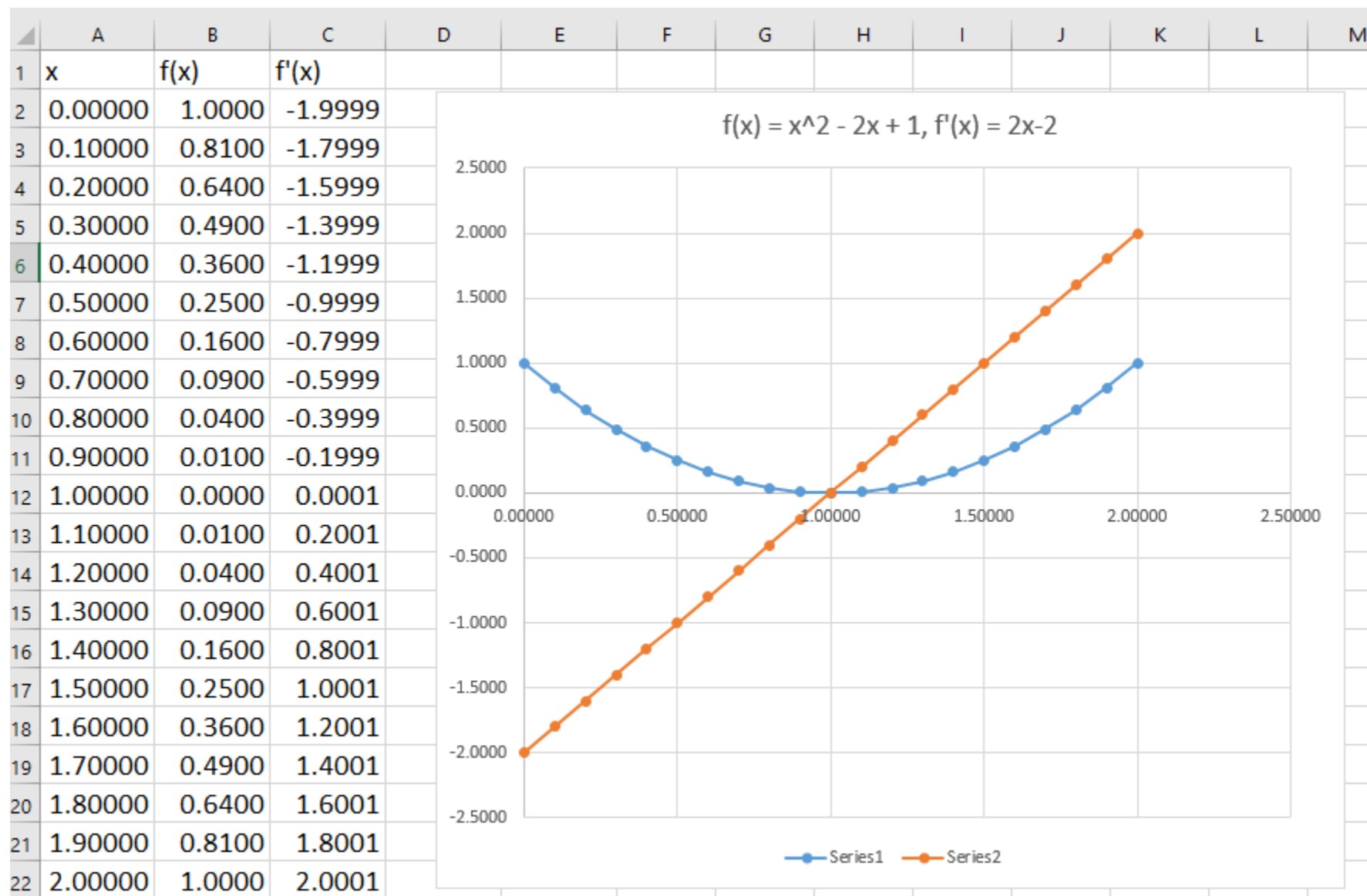
Note: $h = 1/steps = dt$

$$yh = f(x+h)$$

$$y = f(x)$$

Generation of Functional Value and its Derivative

```
28 int main(void){
29     FILE *fp = fopen("poly.txt", "w");
30     fprintf(fp, "Test Derivative of x^2 + 2 x + 1\n");
31     double x;
32     for (x = 0.0; x <= 2.01; x+= 0.1){
33         fprintf(fp, "%10.4f \n", x);
34     }
35     fprintf(fp, "\n");
36     for (x = 0.0; x <= 2.01; x+= 0.1){
37         fprintf(fp, "%10.4f \n", poly(x, c));
38     }
39     fprintf(fp, "\n");
40     for (x = 0.0; x <= 2.01; x+= 0.1){
41         double y = poly(x, c);
42         double yh = poly((x + (1.0/10000.0)), c);
43         fprintf(fp, "%10.4f \n", diff(yh, y, 10000));
44     }
45     fprintf(fp, "\n");
46     fclose(fp);
47     return 0;
48 }
```



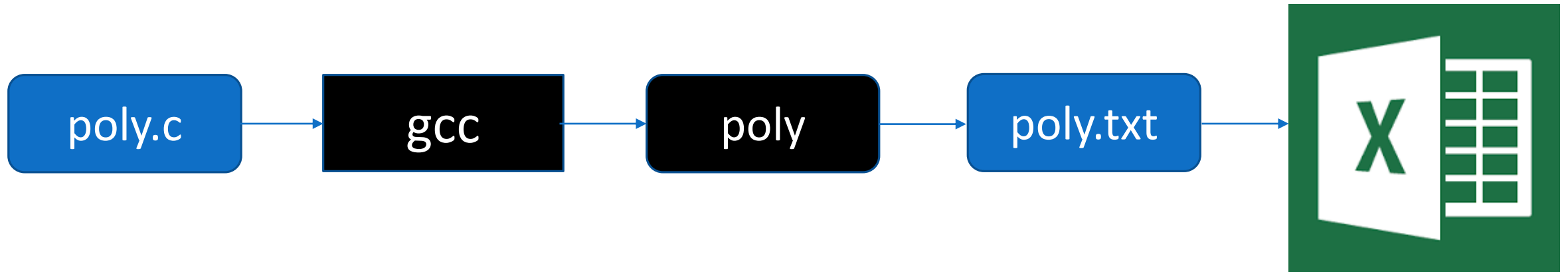
Output Data to a text file “poly.txt”. Then, copy the data into Excel to plot.



Batch file for Windows Command Line

Using C++11 standard for gcc Compiler

```
1 REM Build Poly
2 gcc -std=c11 poly.c -o poly
```





Demo Program:

poly.c + build (First example to use window script and Excel plot)

Go gcc!!!

Purpose: Calculate the functional value of $f(x) = x^2 + 2x + 1$ and its derivative

Note: Just to demonstrate how C language can process mathematical problem very easily. The mathematical and programming problem difficulty may not be suitable for some audience.

fx Functions

Trigonometric functions

✓	cos	Compute cosine (function)
✓	sin	Compute sine (function)
	tan	Compute tangent (function)
✓	acos	Compute arc cosine (function)
✓	asin	Compute arc sine (function)
	atan	Compute arc tangent (function)
	atan2	Compute arc tangent with two parameters (function)

Hyperbolic functions

	cosh	Compute hyperbolic cosine (function)
	sinh	Compute hyperbolic sine (function)
	tanh	Compute hyperbolic tangent (function)
	acosh <small>C++11</small>	Compute area hyperbolic cosine (function)
	asinh <small>C++11</small>	Compute area hyperbolic sine (function)
	atanh <small>C++11</small>	Compute area hyperbolic tangent (function)

Exponential and logarithmic functions

✓	exp	Compute exponential function (function)
	frexp	Get significand and exponent (function)
	ldexp	Generate value from significand and exponent (function)
✓	log	Compute natural logarithm (function)
✓	log10	Compute common logarithm (function)
	modf	Break into fractional and integral parts (function)
✓	exp2 <small>C++11</small>	Compute binary exponential function (function)
	expm1 <small>C++11</small>	Compute exponential minus one (function)
	ilogb <small>C++11</small>	Integer binary logarithm (function)
	log1p <small>C++11</small>	Compute logarithm plus one (function)
✓	log2 <small>C++11</small>	Compute binary logarithm (function)
	logb <small>C++11</small>	Compute floating-point base logarithm (function)
	scalbn <small>C++11</small>	Scale significand using floating-point base exponent (function)
	scalbln <small>C++11</small>	Scale significand using floating-point base exponent (long) (function)


Power functions

✓	pow	Raise to power (function)
✓	sqrt	Compute square root (function)
	cbrt <small>C++11</small>	Compute cubic root (function)
	hypot <small>C++11</small>	Compute hypotenuse (function)

Error and gamma functions

erf <small>C++11</small>	Compute error function (function)
erfc <small>C++11</small>	Compute complementary error function (function)
tgamma <small>C++11</small>	Compute gamma function (function)
lgamma <small>C++11</small>	Compute log-gamma function (function)




Rounding and remainder functions

ceil	Round up value (function)
floor	Round down value (function)
fmod	Compute remainder of division (function)
trunc <small>C++11</small>	Truncate value (function)
 round <small>C++11</small>	Round to nearest (function)
lround <small>C++11</small>	Round to nearest and cast to long integer (function)
llround <small>C++11</small>	Round to nearest and cast to long long integer (function)
rint <small>C++11</small>	Round to integral value (function)
lrint <small>C++11</small>	Round and cast to long integer (function)
llrint <small>C++11</small>	Round and cast to long long integer (function)
nearbyint <small>C++11</small>	Round to nearby integral value (function)
remainder <small>C++11</small>	Compute remainder (IEC 60559) (function)
remquo <small>C++11</small>	Compute remainder and quotient (function)



Floating-point manipulation functions

copysign <small>C++11</small>	Copy sign (function)
nan <small>C++11</small>	Generate quiet NaN (function)
nextafter <small>C++11</small>	Next representable value (function)
nexttoward <small>C++11</small>	Next representable value toward precise value (function)

Minimum, maximum, difference functions

	fdim <small>C++11</small>	Positive difference (function)
	fmax <small>C++11</small>	Maximum value (function)
	fmin <small>C++11</small>	Minimum value (function)

Other functions

	fabs	Compute absolute value (function)
	abs	Compute absolute value (function)
	fma <small>C++11</small>	Multiply-add (function)

Macros / Functions

These are implemented as macros in C and as functions in C++:

Classification macro / functions

fpclassify <small>C++11</small>	Classify floating-point value (macro/function)
isfinite <small>C++11</small>	Is finite value (macro)
isinf <small>C++11</small>	Is infinity (macro/function)
isnan <small>C++11</small>	Is Not-A-Number (macro/function)
isnormal <small>C++11</small>	Is normal (macro/function)
signbit <small>C++11</small>	Sign bit (macro/function)

Comparison macro / functions

isgreater <small>C++11</small>	Is greater (macro)
isgreaterequal <small>C++11</small>	Is greater or equal (macro)
isless <small>C++11</small>	Is less (macro)
islessequal <small>C++11</small>	Is less or equal (macro)
islessgreater <small>C++11</small>	Is less or greater (macro)
isunordered <small>C++11</small>	Is unordered (macro)

Macro constants

math_errhandling <small>C++11</small>	Error handling (macro)
INFINITY <small>C++11</small>	Infinity (constant)
NAN	Not-A-Number (constant)
HUGE_VAL	Huge value (constant)
HUGE_VALF <small>C++11</small>	Huge float value
HUGE_VALL <small>C++11</small>	Huge long double value (constant)

This header also defines the following macro constants (since C99/C++11):

macro	type	description
MATH_ERRNO MATH_ERREXCEPT	int	Bitmask value with the possible values <code>math_errhandling</code> can take.
FP_FAST_FMA FP_FAST_FMAF FP_FAST_FMAL	int	Each, if defined, identifies for which type <code>fma</code> is at least as efficient as <code>x*y+z</code> .
FP_INFINITE FP_NAN FP_NORMAL FP_SUBNORMAL FP_ZERO	int	The possible values returned by <code>fpclassify</code> .
FP_ILOGB0 FP_ILOGBNAN	int	Special values the <code>ilogb</code> function may return.

Types

double_t <small>C++11</small>	Floating-point type (type)
float_t	Floating-point type (type)

LECTURE 9

Random number Generation



Basic Library and Functions for Random Number Generation

Libraries: time.h, stdlib.h, stdint.h

Function: void srand(long seed) // set random number generation with seed

int rand(void) // integer random number generator from 0 to RAND_MAX

Constant: RAND_MAX

These functions and constants can be used directly. However, we can also build some functions to make the use of this functions more efficiently and systematically.



Custom-Defined Function in random.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include <time.h>
5 void reset_random(void);
6 double random(void);
7 uint32_t randInt(uint32_t);
8 int randomInteger(int, int, int);
```

Functions with no argument, you may leave it empty or put keyword `void` there.

Functions with arguments, you may simple leave the argument type to notify the compiler how much argument memory space needed.

random.h: header file to export functions and variables for other modules to use.

reset_random(): reset the random number generator with a time-controlled seed.

random(): generate a random fractional number [0, 1)

randInt(): generate a random integer from 0 to max-1.

randomInteger(): generate a random integer starting from baseline, for every “steps” numbers and totally, “count” number of possible samples.



srand(long seed)

For each seed number, the random number generator rand() will generate a pseudo-random number generator. For a same seed, the random sequence generated will be the same every time.

In order to be un-predictable in generating random numbers, we may use the time() function to generate a random integer of long type. This will make the random sequence be different every time the srand(seed) is called.



void reset_random() function

```
4 void reset_random(){  
5     srand(time(NULL));  
6 }
```

Generate different random sequence every time the random number generator is called.



double random()

double random() generates a fractional number in [0, 1) from 0 to 1 (exclusive). This function works similar to Math.random() in Java language.

See Also:

randomOne(): generates a fractional number in [0, 1] from 0 to 1 (inclusive)

```
12 double random(){
13     return (double) rand()/ (double) (RAND_MAX+1);
14 }
```

```
20 double randomOne(){
21     return (double) rand()/ (double) (RAND_MAX);
22 }
```



uint32_t randInt(uint32_t max)

- randInt(max) generates an integer from 0 to max-1. The same function can be achieved by

(uint32_t) rand() % max

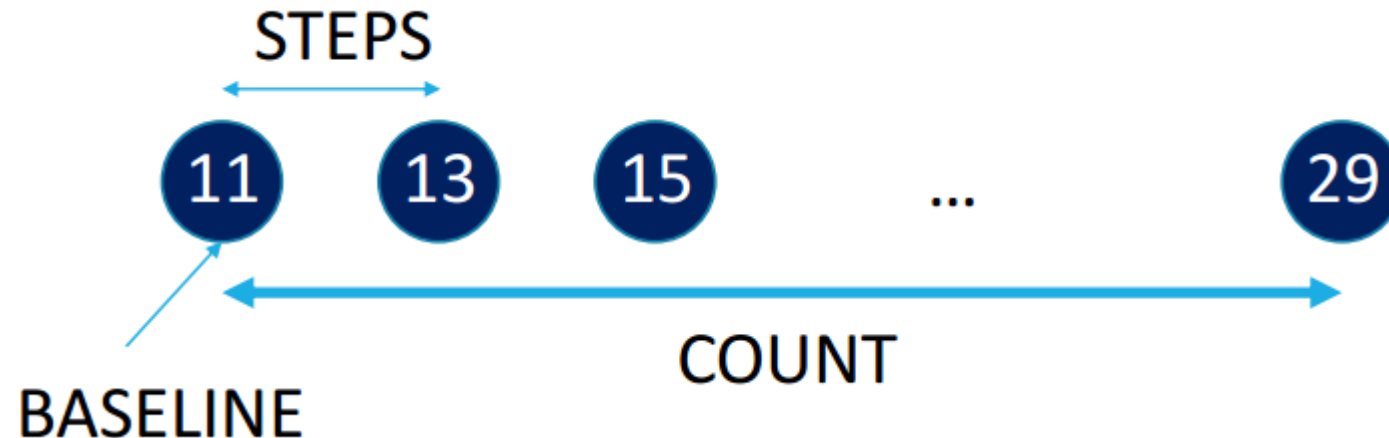
```
24  /*
25  *   random unsigned int generator from 0 to max -1
26  */
27  uint32_t randInt(uint32_t max){
28      return (uint32_t) (random()*max);
29  }
```

```
int randomInteger(int baseline, int steps, int count)
```

Generate random samples with specific number of candidates, starting number and a stepping number.

The samples generated by `randomInteger(11, 2, 10)` is shown in the figure below.

This function is very robust. It can be used in many applications.





Demo Program:

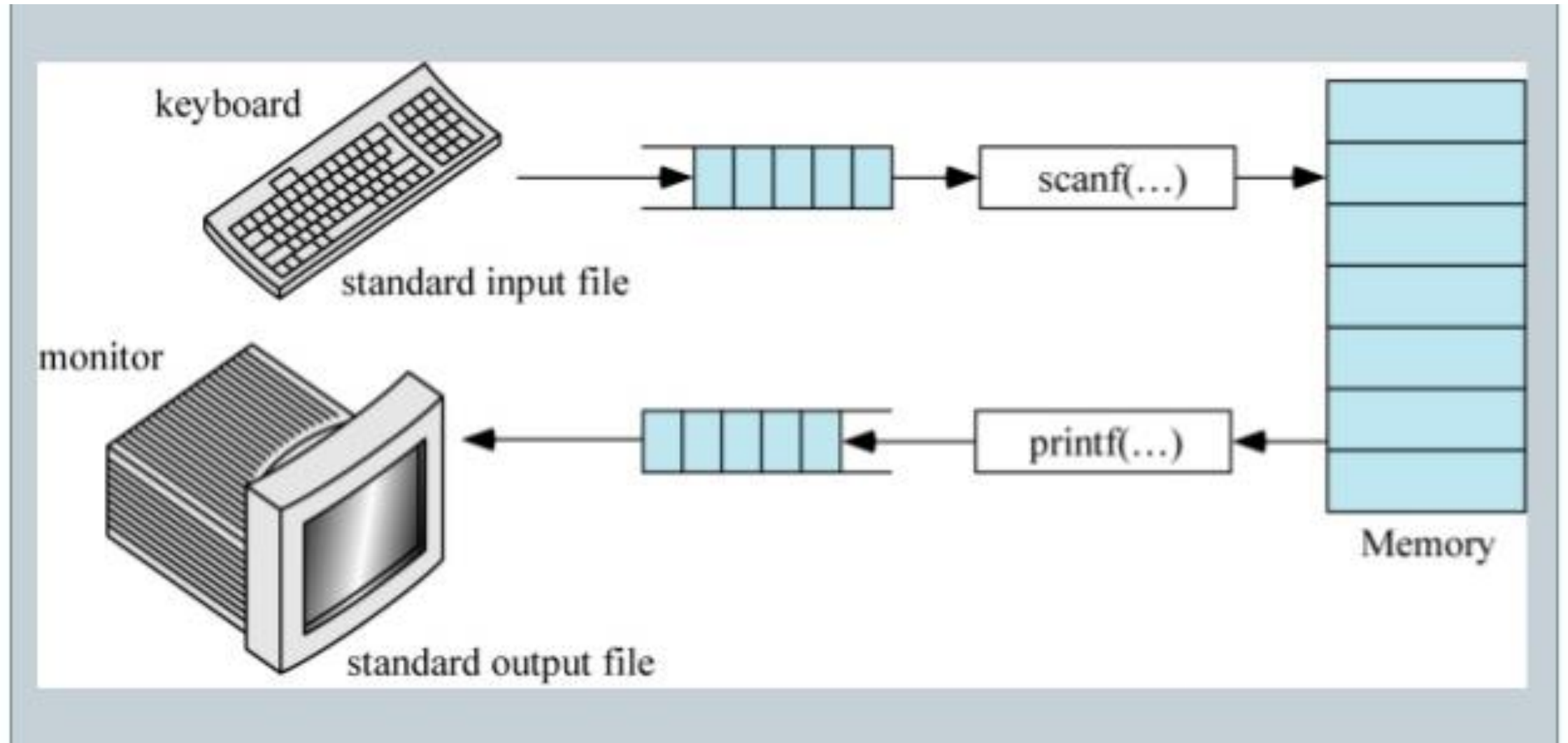
random.c

Go gcc!!!

```
31  /*
32  *   Parametric random number generator that start from the "baseline" number,
33  *   for every "steps"th number and for total of "count" elements.
34  */
35  int randomInteger(int baseline, int steps, int count){
36      return (int) (random()*count)* steps + baseline;
37  }
```


LECTURE 10

Formatted Print (Printf)



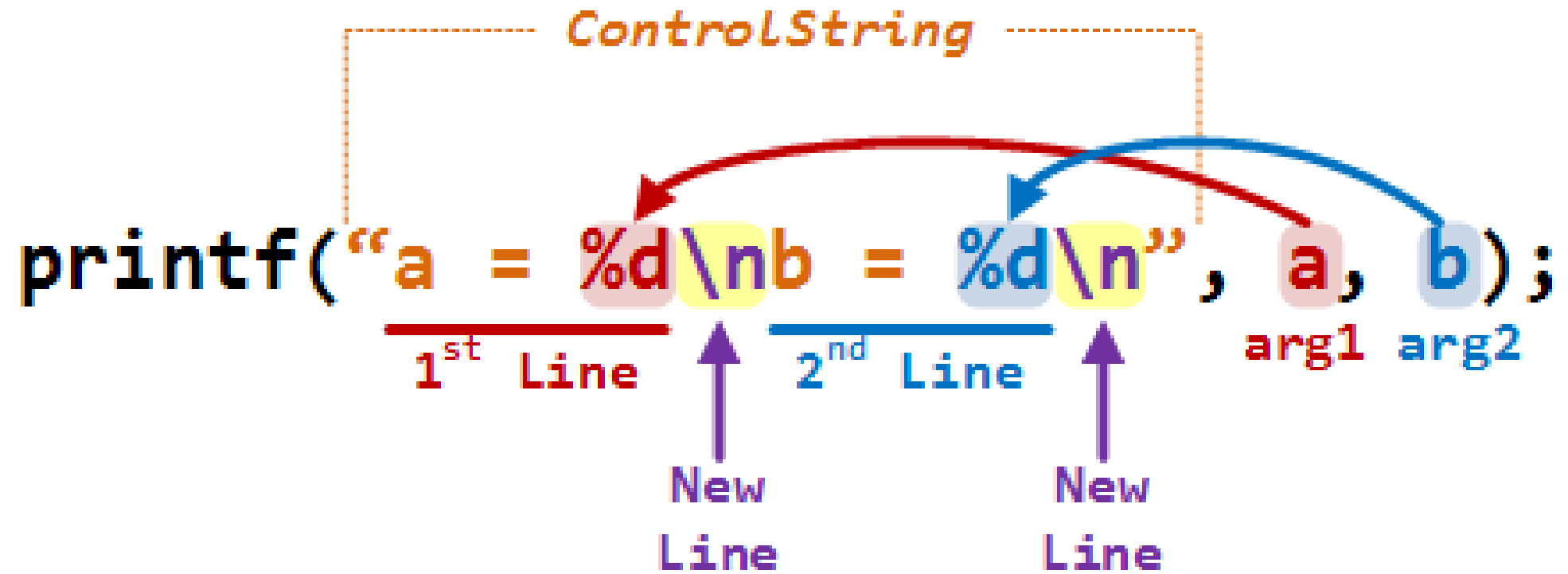
ControlString

```
printf("a = %d\nb = %d\n", a, b);
```

1st Line 2nd Line

New Line New Line

arg1 arg2

A diagram illustrating the printf function's control string. The code 'printf("a = %d\nb = %d\n", a, b);' is shown. The string 'a = %d\nb = %d\n' is enclosed in a dashed orange box labeled 'ControlString'. The first format specifier '%d' is highlighted in a red box, and the second '%d' is in a blue box. The arguments 'a' and 'b' are also highlighted in red and blue boxes respectively, with labels 'arg1' and 'arg2' below them. Red and blue curved arrows connect the format specifiers to their corresponding arguments. Below the string, two underlined sections are labeled '1st Line' and '2nd Line'. Purple arrows point from the text 'New Line' to the '\n' characters in the string.

Formatted Print (printf)



Input / Output

- `printf ();` //used to print to console(screen)
- `scanf ();` //used to take an input from console(user).
 - example: `printf("%c", 'a');` `scanf("%d", &a);`
 - More format specifiers
 - `%c` The character format specifier.
 - `%d` The integer format specifier.
 - `%i` The integer format specifier (same as `%d`).
 - `%f` The floating-point format specifier.
 - `%o` The unsigned octal format specifier.
 - `%s` The string format specifier.
 - `%u` The unsigned integer format specifier.
 - `%x` The unsigned hexadecimal format specifier.
 - `%%` Outputs a percent sign.
 - `%n` The newline mark like `\n`



Some more geek stuff

- & in scanf (call by reference)
 - It is used to access the address of the variable used.
 - example:
 - `scanf("%d", &a);`
 - we are reading into the address of a. If no &, a will not be updated.
- Data Hierarchy.
 - example:
 - int value can be assigned to float not vice-versa.
 - Type casting.

A *format specifier* follows this prototype: [see compatibility note below]
%[flags][width][.precision][length]specifier

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

<i>specifier</i>	Output	Example
<i>d or i</i>	Signed decimal integer	392
<i>u</i>	Unsigned decimal integer	7235
<i>o</i>	Unsigned octal	610
<i>x</i>	Unsigned hexadecimal integer	7fa
<i>X</i>	Unsigned hexadecimal integer (uppercase)	7FA
<i>f</i>	Decimal floating point, lowercase	392.65
<i>F</i>	Decimal floating point, uppercase	392.65
<i>e</i>	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
<i>E</i>	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
<i>g</i>	Use the shortest representation: %e or %f	392.65
<i>G</i>	Use the shortest representation: %E or %F	392.65
<i>a</i>	Hexadecimal floating point, lowercase	-0xc.90fep-2
<i>A</i>	Hexadecimal floating point, uppercase	-0XC.90FEP-2
<i>c</i>	Character	a
<i>s</i>	String of characters	sample
<i>p</i>	Pointer address	b8000000
<i>n</i>	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
<i>%</i>	A % followed by another % character will write a single % to the stream.	%

The *format specifier* can also contain sub-specifiers: *flags*, *width*, *.precision* and *modifiers* (in that order), which are optional and follow these specifications:

<i>flags</i>	description
-	Left-justify within the given field width; Right justification is the default (see <i>width</i> sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see <i>width</i> sub-specifier).

<i>width</i>	description
(<i>number</i>)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The <i>width</i> is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

<i>.precision</i>	description
<i>.number</i>	<p>For integer specifiers (d, i, o, u, x, X): <i>precision</i> specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A <i>precision</i> of 0 means that no character is written for the value 0.</p> <p>For a, A, e, E, f and F specifiers: this is the number of digits to be printed after the decimal point (by default, this is 6).</p> <p>For g and G specifiers: This is the maximum number of significant digits to be printed.</p> <p>For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered.</p> <p>If the period is specified without an explicit value for <i>precision</i>, 0 is assumed.</p>
.*	The <i>precision</i> is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

The *length* sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without *length* specifier (if a different type is used, the proper type promotion or conversion is performed, if allowed):

	specifiers						
<i>length</i>	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Note regarding the *c* specifier: it takes an *int* (or *wint_t*) as argument, but performs the proper conversion to a *char* value (or a *wchar_t*) before formatting it for output.

Note: Yellow rows indicate specifiers and sub-specifiers introduced by C99. See [<inttypes>](#) for the specifiers for extended types.



Demo Program:

formattedprint.c

Go gcc!!!

```
1  #include <stdio.h>
2  int main() {
3      char ch = 'A';
4      char str[20] = "fresh2refresh.com";
5      float flt = 10.234;
6      int no = 150;
7      double dbl = 20.123456;
8      printf("Character is %c \n", ch);
9      printf("String is %s \n", str);
10     printf("Float value is %f \n", flt);
11     printf("Integer value is %d\n", no);
12     printf("Double value is %lf \n", dbl);
13     printf("Octal value is %o \n", no);
14     printf("Hexadecimal value is %x \n", no);
15     return 0;
16 }
```

```
C:\Eric_Chou\C Course\CDev\Ch3\formattedprint>formattedprint
Character is A
String is fresh2refresh.com
Float value is 10.234000
Integer value is 150
Double value is -0.000000
Octal value is 226
Hexadecimal value is 96
```

Keyboard input: `scanf()`

- `scanf()` will **scan** formatted input from the keyboard.
- It uses the same format specifiers as `printf()`
- To read an integer:

```
int num_students;  
scanf("%d", &num_students);
```

Specifier for
"reading an integer value"

VERY IMPORTANT
special symbol

"Place value into this
variable"

Format specifiers for scanf ()

- Format specifiers:

- **%c** for single characters

- ✧ `scanf (" %c", &some_character);`

always put a space between " and % when reading characters

- **%d** for integers

- ✧ `scanf ("%d", &some_integer);`

always using pass by reference (an address)
to allow the variable to be updated

- **%f** for float

- ✧ `scanf ("%f", &some_float);`

- **%lf** for double

- ✧ `scanf ("%lf", &some_double);`

LECTURE 11

Escape Sequence



Escape Sequence in C

- An escape sequence in C language is a sequence of characters that doesn't represent itself when used inside string literal or character.
- It is composed of two or more characters starting with backslash \. For example: \n represents new line.
- Formatted Print/Scan and escape sequence, in combination, are used to provide formatted input and output.

Escape Sequence	Meaning
\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark
\nnn	octal number
\xhh	hexadecimal number
\0	Null

Part numbers
must have
leading zeros

```
printf ("Part Number\tQty On Hand\tQty On Order\t\tPrice\n");
```

Part Number	Qty On Hand	Qty On Order	Price
031235	22	86	\$ 45.62
000321	55	21	\$ 122.00
028764	0	24	\$ 0.75
003232	12	0	\$ 10.91

End of Report

Leading zeros
suppressed

Decimal points
must be
aligned



Demo Program:

escape.c

Go gcc!!!

```
1  #include <stdio.h>
2  #include <conio.h>
3  void main(){
4      int number=50;
5      //clrscr();    // clear console screen
6      printf("You\nare\nlearning\n'c\' language\n\"Do you know C language\"\n");
7
8      char ch = getch();    // get a key code
9      printf("ASCII Keycode = %d\n", ch);
10
11 }
```