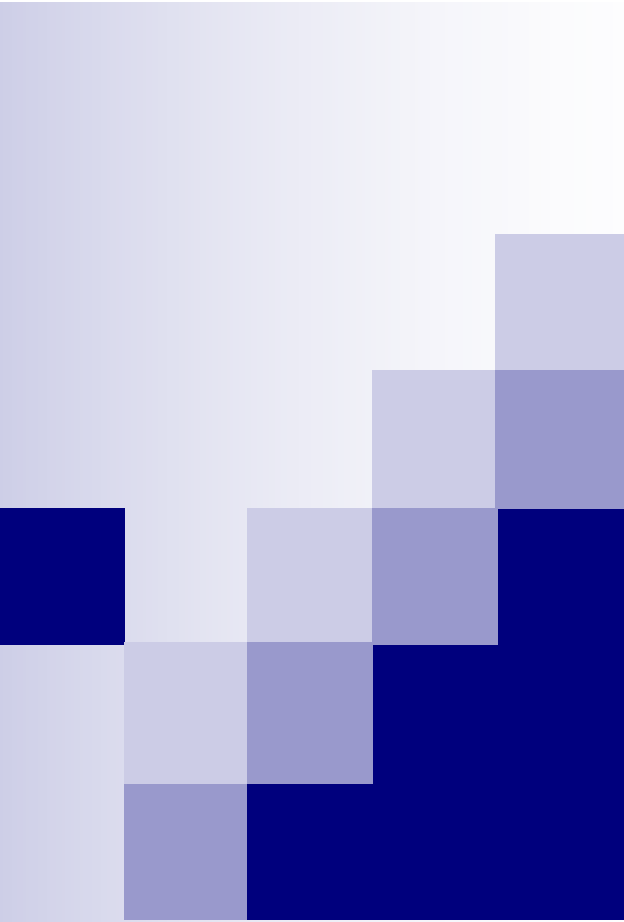


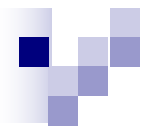
# **CS10003: Programming & Data Structures**

**Dept. of Computer Science & Engineering  
Indian Institute of Technology Kharagpur**

*Autumn 2020*

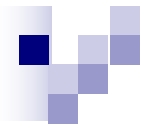


# **Data-types, Variables and I/O**



# Integer Data Type

Integer data type	Bit size	Minimum value	Maximum value
char	8	$-2^7 = -128$	$2^7 - 1 = 127$
short int	16	$-2^{15} = -32768$	$2^{15} - 1 = 32767$
int	32	$-2^{31} = -2147483648$	$2^{31} - 1 = 2147483647$
long int	32	$-2^{31} = -2147483648$	$2^{31} - 1 = 2147483647$
long long int	64	$-2^{63} = -9223372036854775808$	$2^{63} - 1 = 9223372036854775807$
unsigned char	8	0	$2^8 - 1 = 255$
unsigned short int	16	0	$2^{16} - 1 = 65535$
unsigned int	32	0	$2^{32} - 1 = 4294967295$
unsigned long int	32	0	$2^{32} - 1 = 4294967295$
unsigned long long int	64	0	$2^{64} - 1 = 18446744073709551615$

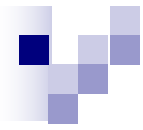


# Float Data Type

Like integers, C provides representations of real numbers and those representations are finite.

Depending on the size of the representation, C's real numbers have got different names.

<b>Real data type</b>	<b>Bit size</b>
float	32
double	64
long double	128



# The **sizeof** function

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

```
void main()
```

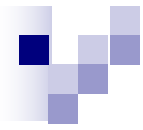
```
{
```

```
    printf("size of short int: %d\n",sizeof(short  
int));
```

```
    printf("size of int is %d\n",sizeof(int));
```

```
    printf("size of long int is %d\n",sizeof(long  
int));
```

```
}
```



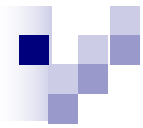
# Char data type

char for representing characters.

We need a way to express our thoughts in writing.

This has been traditionally achieved by using an alphabet of symbols with each symbol representing a sound or a word or some punctuation or special mark.

The computer also needs to communicate its findings to the user in the form of something written.



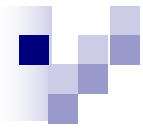
# Char data type

Since the outputs are meant for human readers, it is advisable that the computer somehow translates its bit-wise world to a human-readable script.

The Roman script (mistakenly also called the English script) is a natural candidate for the representation.

The Roman alphabet consists of the lower-case letters (a to z), the upper case letters (A to Z), the numerals (0 through 9) and some punctuation symbols (period, comma, quotes etc.).

In addition, computer developers planned for inclusion of some more control symbols (hash, caret, underscore etc.). Each such symbol is called a **character**.



# ASCII Code

In order to promote interoperability between different computers, some standard encoding scheme is adopted for the computer character set.

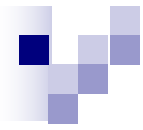
This encoding is known as **ASCII** (abbreviation for **American Standard Code for Information Interchange**).

In this scheme each character is assigned a unique integer value between 32 and 127.

Since eight-bit units (bytes) are very common in a computer's internal data representation, the code of a character is represented by an 8-bit unit.

Since an 8-bit unit can hold a total of  $2^8=256$  values and the computer character set is much smaller than that, some values of this 8-bit unit do not correspond to visible characters.





# Printable Characters

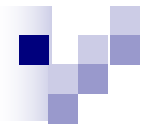
These values are often used for representing invisible control characters (like line feed, alarm, tab etc.) and extended Roman letters (inflected letters like ä, é, ç).

Some values are reserved for possible future use.

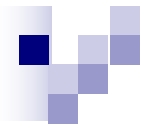
The ASCII encoding of the printable characters is summarized in the following table.



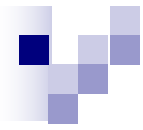
Decimal	Hex	Binary	Character	Decimal	Hex	Binary	Character
32	20	00100000	SPACE	80	50	01010000	P
33	21	00100001	!	81	51	01010001	Q
34	22	00100010	"	82	52	01010010	R
35	23	00100011	#	83	53	01010011	S
36	24	00100100	\$	84	54	01010100	T
37	25	00100101	%	85	55	01010101	U
38	26	00100110	&	86	56	01010110	V
39	27	00100111	'	87	57	01010111	W
40	28	00101000	(	88	58	01011000	X
41	29	00101001	)	89	59	01011001	Y
42	2a	00101010	*	90	5a	01011010	Z
43	2b	00101011	+	91	5b	01011011	[
44	2c	00101100	,	92	5c	01011100	\
45	2d	00101101	-	93	5d	01011101	]
46	2e	00101110	.	94	5e	01011110	^
47	2f	00101111	/	95	5f	01011111	_
48	30	00110000	0	96	60	01100000	`
49	31	00110001	1	97	61	01100001	a
50	32	00110010	2	98	62	01100010	b



51	33	00110011	3	99	63	01100011	c
52	34	00110100	4	100	64	01100100	d
53	35	00110101	5	101	65	01100101	e
54	36	00110110	6	102	66	01100110	f
55	37	00110111	7	103	67	01100111	g
56	38	00111000	8	104	68	01101000	h
57	39	00111001	9	105	69	01101001	i
58	3a	00111010	:	106	6a	01101010	j
59	3b	00111011	;	107	6b	01101011	k
60	3c	00111100	<	108	6c	01101100	l
61	3d	00111101	=	109	6d	01101101	m
62	3e	00111110	>	110	6e	01101110	n
63	3f	00111111	?	111	6f	01101111	o
64	40	01000000	@	112	70	01110000	p
65	41	01000001	A	113	71	01110001	q
66	42	01000010	B	114	72	01110010	r
67	43	01000011	C	115	73	01110011	s
68	44	01000100	D	116	74	01110100	t
69	45	01000101	E	117	75	01110101	u
70	46	01000110	F	118	76	01110110	v



71	47	01000111	G		119	77	01110111	w
72	48	01001000	H		120	78	01111000	x
73	49	01001001	I		121	79	01111001	y
74	4a	01001010	J		122	7a	01111010	z
75	4b	01001011	K		123	7b	01111011	{
76	4c	01001100	L		124	7c	01111100	
77	4d	01001101	M		125	7d	01111101	}
78	4e	01001110	N		126	7e	01111110	~
79	4f	01001111	O		127	7f	01111111	DELETE



# Qualifiers

Qualifiers add more data types.

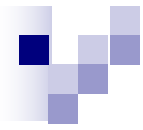
typically *size* or *sign*.

*size: short or long*

*sign: signed or unsigned*

*signed short int, unsigned short int, signed int, signed long int, long double, long int*

*signed char and unsigned char*



# A Comment

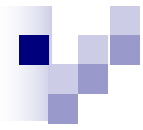
A char data type is also an integer data type.

If you want to interpret a char value as a character, you see the character it represents. If you want to view it as an integer, you see the ASCII value of that character.

For example, the upper case A has an ASCII value of 65.

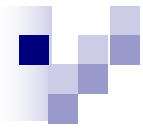
An eight-bit value representing the character A automatically represents the integer 65,

because to the computer A is recognized by its ASCII code, not by its shape, geometry or sound!



# Character assignment Example

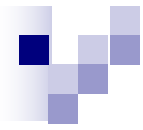
```
#include <stdio.h>
void main()
{
    char c;
    c = '#';
    printf("This is a hash symbol: %c\n",c);
}
```



# ASCII Code example

```
#include <stdio.h>
void main( )
{
    int code;
    char symbol;
    printf("Input an ASCII code (0 to 127): ");
    scanf("%d",&code);
    symbol=code;
    printf("The symbol corresponding to ASCII %d is
    %c\n",code,symbol);
}
```





# Static and Global variables

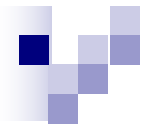
Often you shall see declarations like:

`static int`

These variables retain their content in memory as long as the program executes.

Their usage will be more clear when we study 'functions'

Global variables are visible from other function calls.



# Pointer Data Type

Pointers are addresses in memory.

In order that the user can directly manipulate memory addresses, C provides an abstraction of addresses.

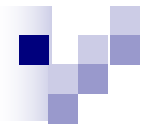
The memory location where a data item resides can be accessed by a pointer to that particular data type. C uses the special character \* to declare pointer data types.

A pointer to a double data is of data type double \*.

A pointer to an unsigned long int data is of type unsigned long int \*.

A character pointer has the data type char \*.

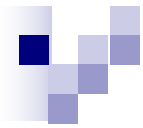
**We will study pointers more elaborately later in this course.**



# Constants

Defining a data type is not enough.

You need to assign the variables and work with specific values of various data types.



# Integer Constants

An integer constant is a non-empty sequence of decimal numbers preceded optionally by a sign (+ or -).

However, the common practice of using commas to separate groups of three (or five) digits is not allowed in C.

Nor are spaces or any character other than numerals allowed.

Here are some valid integer constants:

332

-3002

+15

-00001020304

And here are some examples that C compilers do not accept:

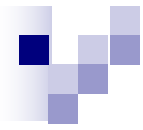
3 332

2,334

- 456

2-34

12ab56cd



# Hexadecimal values

You can also express an integer in base 16, i.e., an integer in the **hexadecimal** (abbreviated **hex**) notation.

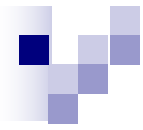
In that case you must write either 0x or 0X before the integer.

Hexadecimal representation requires 16 digits  
0, 1, ..., 15.

In order to resolve ambiguities the digits  
10, 11, 12, 13, 14, 15 are respectively denoted by  
a, b, c, d, e, f (or by A, B, C, D, E, F).

Here are some valid hexadecimal integer constants:

0x12ab56cd -0X123456 0xABCD1234 +0XaBCd12



# Real Constants

Real constants can be specified by the usual notation comprising an optional sign, a decimal point and a sequence of digits. Like integers no other characters are allowed.

Real numbers are sometimes written in the *scientific notation* (like  $3.45 \times 10^{67}$ ). The following expressions are valid for writing a real number in this fashion:  
3.45e67 +3.45e67 -3.45e-67 .00345e-32 1e-15.

You can also use E in place of e in this notation

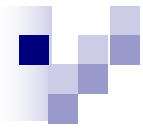


# Input Output for short and long int

```
#include<stdio.h>
void main( )
{
    short int shorti;
    long int longi;
    printf("Input short int: ");
    scanf("%hd",&shorti);
    printf("%hd\n",shorti);
    printf("Input long int: ");
    scanf("%ld",&longi);
    printf("%ld\n",longi);
    printf("shorti = %hd and longi=%ld",shorti,longi);
}
```

## **A Sample Run:**

```
Input short int: 20
Input long int: 2000000
shorti= 20 and longi= 2000000
```



# The **typedef** statement

This statement can be used to define new data types.

For example:

```
typedef unsigned long ulong;
```

*ulong is a new data type equivalent to unsigned long*

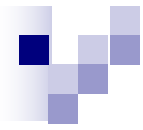
It can be used as any other data type as follows;

```
ulong u;
```

(declares u to be of the type ulong)

The size of the new data type can also be found in bytes using **sizeof(ulong)**





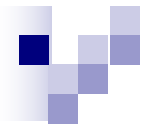
# Examples

```
int m, n, armadillo;
```

```
int platypus;
```

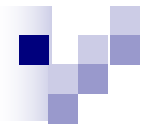
```
float hi, goodMorning;
```

```
unsigned char _u_the_charcoal;
```



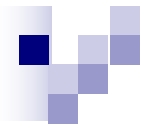
# Constants of different Integer types

Since different integer data types use different amounts of memory and represent different ranges of integers, it is often convenient to declare the intended data type explicitly.



# Constants of different Integer types

<b>Suffix</b>	<b>Data type</b>
L (or l)	long
LL (or ll)	long long
U (or u)	unsigned
UL (or ul)	unsigned long
ULL (or ull)	unsigned long long



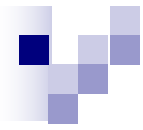
# Examples

4000000000UL

123U

0x7FFFFFFFFI

0x123456789abcdef0ULL



# Character Constants

Character constants are single printable symbols enclosed within single quotes.

Here are some examples: 'A' '7' '@' ''



# Special Characters

<b>Constant</b>	<b>Character</b>	<b>ASCII value</b>
'\0'	Null	0
'\b'	Backspace	8
'\t'	Tab	9
'\n'	New line	13
'\"'	Single Quote	39
'\\'	Backslash	92



# Try this!

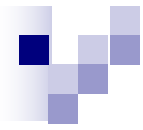
```
#include<stdio.h>
main()
{
    int i;
    for(i=0;i<10000;i++)
        printf("%c",'\a')
}
```



# Formats

- `%c` The character format specifier.
- `%d` The integer format specifier.
- `%i` The integer format specifier (same as `%d`).
- `%f` The floating-point format specifier.
- `%e` The scientific notation format specifier.
- `%E` The scientific notation format specifier.
- `%g` Uses `%f` or `%e`, whichever result is shorter.
- `%G` Uses `%f` or `%E`, whichever result is shorter.
- `%o` The unsigned octal format specifier.
- `%s` The string format specifier.
- `%u` The unsigned integer format specifier.
- `%x` The unsigned hexadecimal format specifier.
- `%X` The unsigned hexadecimal format specifier.
- `%p` Displays the corresponding argument that is a pointer.
- `%n` Records the number of characters written so far.
- `%%` Outputs a percent sign.



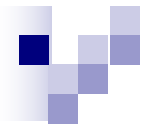


# Characters can be represented by numbers

Since characters are identified with integers in the range -127 to 128 (or in the range 0 to 255), you can use integer constants in the prescribed range to denote characters.

The particular sequence '\xuv' (synonymous with 0xuv) lets you write a character in the hex notation.

For example, '\x2b' is the integer 43 in decimal notation and stands for the character '+'.  
+



# Pointer constants

**It is dangerous to work with constant addresses.**

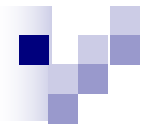
**You may anyway use an integer as a constant address.**

**But doing that lets the compiler issue you a warning message.**

**Finally, when you run the program and try to access memory at a constant address, you are highly likely to encounter a frustrating mishap known as "Segmentation fault".**

**It occurs when the memory is accessed at an illegal address (beyond what you are supposed to).**

**However there is a pointer constant that is used widely. This is called NULL. A NULL pointer points to nowhere.**



# Variables

“The only constant thing is change”

Variables help to abstract this change.

Teacher = XYZ ;

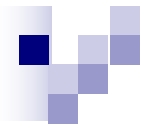
here **Teacher** is a variable

**XYZ** is one instance of the variable, and is a constant

**A variable is an entity that has a value and is known to the program by a name,**

A variable definition associates a memory location with the variable name.

At one time it can have only one value associated with it.



# Declaring variables

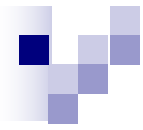
For declaring one or more variables of a given data type do the following:

First write the data type of the variable.

Then put a space (or any other white character).

Then write your comma-separated list of variable names.

At the end put a semi-colon. **Eg, int a, b;**



# Promotion and typecasting of variables

```
int i;
```

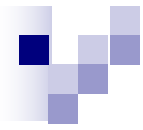
```
float f;
```

```
i = 5;
```

```
f = i;
```

The last statement assigns `i` to `f`. Since `i` is an integer and `f` is float, the conversion is automatic.

**Promotion:** This type of conversion, when the variable of lower type is converted to a higher type is called promotion.

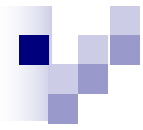


# Integral Promotion

```
int i;  
char c;  
c = 'a';  
i = c;
```

The value present in the character variable 'c', i.e the ASCII code of the character 'a' is assigned to the integer 'i'.

But i is typically represented using 2 bytes and c with 1 byte. Here comes the concept of sign extension.



# Sign extension

Conversion to a signed integer from character data type:

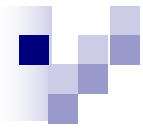
- lower 8 bits will be the character's value.

- higher 8 bits will be filled with 0 or 1, depending on the Maximum Significant Bit (MSB) of the character.

(Note: MSB is used to indicate the sign of a signed number)

This is called sign extension.

**Sign extension** takes place only if the variable of the higher type is *signed*.



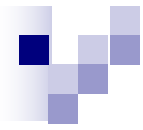
# Truncation

$f=7.5$

$i = f$

This results in the discarding of .5. The value 7 is assigned to i.

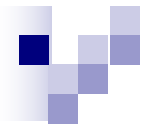




# Forcible Conversion

```
int i, j;  
float f;  
i=12; j=5;  
f = i/j;  
printf(“%f\n”,f);
```

**The output is 2.0. This is because both i and j are integers, an integer division will take place.**



# Typecasting

In order to have a floating division, either *i* or *j* should be float.

We can change say *i*, from integer to float by typecasting, using:

**(float) i**

Thus we have to change the division line to:

**f = (float) i/j;**

**The general syntax is:**

**(type) variable\_name;**

**Typecasting can also be used to convert a higher data type to a lower type, for example: (int) f**