

Programming Fundamentals for Engineers  
0702113

## 5. Basic Data Types

**MUNTASER ABULAFI**  
**YACOUB SABATIN**  
**DR. LABIB ARAFEH**

1

### C Data Types

#### Variable definition

- C has a concept of 'data types' which are used to define a variable before its use.
- The definition of a variable will assign storage for the variable and define the type of data that will be held in the location.
- The available **data types** are → *int, float, double, longdouble, char, void, enum*
  - **int** → Used to define integer numbers:  
**int Count;**  
**Count = 5;**

2

- **float** → Used to define floating point numbers:  
**float Miles;**  
**Miles = 5.6;**
- **double** → Used to define BIG **floating point** numbers (Reserves **twice** the storage for the number. On PCs this is likely to be 8 bytes.)  
**double Atoms, big;**  
**big = 312E+7;**  
**Atoms = 2500000;**

3

- **char** → Defines characters.  
**char Letter;**  
**Letter = 'x';** /\* Note: **x** is enclosed in single quotes\*/

4

## Modifiers

- *Modifiers* define the amount of **storage** allocated to the variable.
- Data types *int*, *float*, *double* have modifiers: **short**, **long**, **signed** & **unsigned**
- ANSI has the following rules:
 

|                  |                      |                           |
|------------------|----------------------|---------------------------|
| <i>short int</i> | $\leq$ <i>int</i>    | $\leq$ <i>long int</i>    |
| <i>float</i>     | $\leq$ <i>double</i> | $\leq$ <i>long double</i> |
- i.e. *short int* assigns **less than or equal to** amount of **storage** as an *int*, and *int* assigns **less than or equal long int**.

| Type               | Bytes | Range                            |
|--------------------|-------|----------------------------------|
| short int          | 2     | -16,384 to +16,383               |
| unsigned short int | 2     | 0 to +32,767                     |
| Unsigned long int  | 4     | 0 to +4,294,967,295              |
| Int                | 4     | -2,147,483,648 to +2,147,483,647 |
| long int           | 4     | -2,147,483,648 to +2,147,483,647 |
| signed char        | 1     | -128 to +127                     |
| unsigned char      | 1     | 0 to +255                        |
| float              | 4     |                                  |
| double             | 8     |                                  |
| long double        | 12    |                                  |

Note: the above numbers are for 32-bit machines, in general they depend on the machine, see the text book, p111

## Signed & Unsigned Integers

- A variable of the type *int* by default, its value is **SIGNED**. That is: variable could be **positive or negative**;
- Value of a **signed int** ranges between: a **Min. value of -32768** ( $-2^{15}$ ) & a **max. value of 32767** ( $2^{15} - 1$ ).
- An **unsigned int** can only store **positive values** with range from **0 to 65535** ( $2^{16} - 1$ ).
- To read & write:
  - **Unsigned integers**, use **u** (U) instead of **d**;
 

```
unsigned int i;
scanf("%u", &i)
printf("%u", i);    /* prints i in base 10) */
```
  - **Short integers**, put **h** (H) in front of **d, o, u, x**

```
short int s;
scanf("%hd", &s)
printf("%hd", s);
```

- Long integers, put **l** (L) in front of **d, o, u, x**  

```
long int y;
scanf("%ld", &y);
printf("%ld", y);
```
- To read value of type **double**, put **l** in front of **e, f** or **g**:  

```
double d;
scanf("%lf", &d);
```
- Note:** *printf* format string, **e, f & g** can be used to write float or double
- To read & write **Long double**, put **L** in front of **e, f** or **g**  

```
long double ld;
scanf("%Lf", &ld);
printf("%Lf", ld);
```
- **%d** → Displays an **integer** in **decimal** form
- **%e** → Displays a **floating-point** number in **exponential** format (one digit with fractions and the exponential symbol *e*).
- **%f** → Displays a **floating-point** in "**fixed decimal**" format
- **%c** → Displays a **character**
- **%s** → Displays **strings** (two characters or more)
- **%g** → Displays a **floating-point** number in either "**exponential** format" or "**fixed** format" depending on the size of the number (**%g** gives the shorter of **%e** or **%f**).

C\_Prog. Spring 2012

7

| <b>%specifier</b>      | <b>Output</b>  | <b>Example</b> |
|------------------------|--|----------------|
| <b>%c</b>              | Character  | a              |
| <b>%d</b> or <b>%i</b> | Signed decimal integer   | 392            |
| <b>%e</b>              | Scientific notation (mantissa/exponent) using e character  | 3.9265e+2      |
| <b>%E</b>              | Scientific notation (mantise/exponent) using E character   | 3.9265E+2      |
| <b>%f</b>              | Decimal floating point   | 392.65         |
| <b>%g</b>              | Use the shorter of %e or %f  | 392.65         |
| <b>%G</b>              | Use the shorter of %E or %f  | 392.65         |
| <b>%o</b>              | Signed octal   | 610            |
| <b>%s</b>              | String of characters   | sample         |
| <b>%u</b>              | Unsigned decimal integer   | 7235           |
| <b>%x</b>              | Unsigned hexadecimal integer   | 7fa            |
| <b>%X</b>              | Unsigned hexadecimal integer (capital letters)   | 7FA            |
| <b>%p</b>              | Pointer address  | B800:0000      |
| <b>%n</b>              | Nothing printed. The argument must be a pointer to a signed int, Number of characters written by printf. |                |
| <b>%%</b>              | A % followed by another % character will write % to stdout.  | 100%           |

8

C\_Prog. Spring 2012

## RADIX CHANGING

- *Radix* is the base of the system, 10 is the radix in decimal system.
- *Integer constants* may be expressed in any base by **altering** the **modifier**:
  - *Decimal* → 0-9; [base 10],
  - *octal* → 0-7; [base 8], or
  - *Hexadecimal* → 0-9, a-f; [base 16]
- Achieved by the letter that follows % sign related to *printf* argument.
- Integer constants can be defined in **octal** / **hex** using the associated prefix
  - To define an integer as an octal constant use `%o`;
 

```
int sum = %o567; //Octal
```
  - To define an integer as a hex constant use `%0x` (*zero*);
 

```
int sum = %0x7ab4;
```

```
int flag = %0X7AB4; // Note: upper or lowercase hex are ok
```

9

C\_Prog. Spring 2012

- To read unsigned integer **num**:
 

```
scanf("%u", &num); // reads num is base 10
printf("%u", num); // prints num in base 10
scanf("%o", &num); // reads num is base 8
printf("%o", num); // prints num in base 8
scanf("%x", &num); // reads num is base 16
printf("%x", num); // prints num in base 16
```
- **Example:** Prints same value in Decimal, Hex & Octal
 

```
int number = 100;
printf("In decimal the number is %d\n", number);
printf("In hex the number is %x\n",
number);
printf("In octal the number is %o\n",
number);
```

### Sample program output

```
In decimal the number is 100
In hex the number is 64
In octal the number is 144
```

10

C\_Prog. Spring 2012



- Function *toupper* → converts case to upper case;
- Need to use directive *#include <ctype.h>*:  
Ex: *ch = toupper (ch)* //Now ch is the upper version of the character.
- Can read/write characters using the conversion specification *%c*

```
char ch;
scanf("%c", &ch);
Printf("%c", ch);
```

13

## getchar & putchar

- Can read/write characters in different ways than *printf* and *scanf*;
- Defined in *stdio.h* header file.
- When *getchar* is called, it reads one character which it returns it  
*ch = getchar();* // reads a character and stores it in ch;
- *putchar* writes a single character  
*putchar (ch);*
- They are faster than *printf* and *scanf*;
- Can be used in:
  - macros, and
  - with many types of data...
  - *getchar()* can be used as 'press any key to continue'

**/\* A C Code to determine the length of a**

```

#include <stdio.h>
main ()
{
char ch;
int len=0;
printf(" Enter a message: ");
ch = getchar();
while (ch !='\n')
{len++;
ch = getchar();}
printf(" Your Message was %d characters", len);

return 0;
}

```

Enter a message: C is my beloved Language  
Your Message was 24 characters Press any key to continue\_

## The sizeof Operator

- A function to find out how many **BYTES** a variable occupies;
- Think of a byte as a container in the computer's memory.

*sizeof (typename)*

- We can find out how much memory is required to store variables of that data type, as demonstrated by this example:

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

```
printf("Size of short int is %d bytes\n", sizeof(short int));
```

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

```
printf("Size of signed int is %d bytes\n",sizeof(signed int));
```

```
printf("Size of signed short int is %d bytes\n",sizeof(signed short int));
```

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

```
printf("Size of unsigned int is %d bytes\n", sizeof(signed int));
```

```
printf("Size of unsigned short int is %d bytes\n", sizeof(unsigned short int));
```

```
printf("Size of unsigned long int is %d bytes\n\n", sizeof(unsigned
long int));
printf("Size of char is %d byte\n", sizeof(char));
printf("Size of float is %d bytes\n", sizeof(float));
printf("Size of double is %d bytes\n", sizeof(double));
```

- Note: Space required is not affected by signed/unsigned type modifiers

```
Size of int is 2 bytes
Size of short int is 2 bytes
Size of long int is 4 bytes
Size of signed int is 2 bytes
Size of signed short int is 2 bytes
Size of signed long int is 4 bytes
Size of unsigned int is 2 bytes
Size of unsigned short int is 2 bytes
Size of unsigned long int is 4 bytes
Size of char is 1 byte
Size of float is 4 bytes
Size of double is 8 bytes
```

17 C\_Prog. Spring 2012

## Type Conversion

- **C** allows mixing of the basic types in operations
- Conversions from one type to another is often necessary to enable computer perform the operation

### Example:

- Adding character to integer (character is stored as integer): `'a' + 1`
- Sometimes conversion needs to be done when adding *int* to *float* as they are stored differently: `3.6 + 4`
- Conversions can be *implicit* (automatic) or *explicit* (specified by the programmer – using *cast*)
- Implicit conversions are performed in two situations:
  - **The operands in an arithmetic or logical expressions are of different types:** `(a+b)` → *C will perform usual arithmetic conversions*
  - **Left side & right side of an assignment has different types:** `(a=b)`  
 → *C will convert right operand into the type of the left operand*

18

C\_Prog. Spring 2012

- For Assignments, **C** converts right operand to the type of the left one:

```
char c;
```

```
int i;
```

```
float f;
```

```
double d;
```

```
i=c; // c is converted to int
```

```
f=i; // i converted to float
```

```
d=f; // f converted to double
```

- Assigning float to integer drops fraction part:

```
int i;
```

```
i=842.97; // i is now 842, because it's int
```

```
i=-842.97; // i is now -842
```

19

C\_Prog. Spring 2012

## More on conversion

- **Recall:** In an entirely *int* context,  
 $40 / 17 * 13 / 3 \rightarrow \text{answer} = 8$   
 $\rightarrow 40 / 17$  rounds to 2  
 $\rightarrow 2 * 13 = 26$   
 $\rightarrow 26 / 3$  rounds to 8
- $40 / 17 * 13 / 3.0 \rightarrow \text{answer} = 8.666$   
 $\rightarrow 40 / 17$  again rounds to 2  
 $\rightarrow 2 * 13 = 26$   
 $\rightarrow$  but the **3.0** forces the final division into a double context and thus  $26.0 / 3.0 = 8.666\dots$
- $(35*2 + 10\%5 + 12\%4/5*2) \rightarrow 70$      $(5\%2 + 14 / 3 - 5) \rightarrow 0$
- $(6 * 5 ) / + 10 * (2 - 10) \rightarrow -24$      $(5\%2 > 16 * .25 \&\& 6 - 5) \rightarrow 0$

20

- Try:  
 $(40 / 17 * 13.0 / 3) \rightarrow$  answer still = 8.666...  
 $\rightarrow 40 / 17$  rounds to 2  
 $\rightarrow$  the 13.0 forces the multiplication to a double but  $2.0 * 13.0$  still equals 26.0  
 $\rightarrow$  and the 26.0 still forces the final division into a double context and  $26.0 / 3.0 = 8.666...$
- Try:  
 $(40 / 17.0 * 13 / 3), \rightarrow$  answer = 10.196  
 $\rightarrow$  the 17.0 forces the initial division into a double context and  $40.0 / 17.0 = 2.352...$   
 $\rightarrow 2.352... * 13.0 = 30.588...$   
 $\rightarrow$  and  $30.588... / 3.0 = 10.196...$

21

### ***explicit Conversion: Casts***

- Casts are used to force conversion/change of the data type of a variable;
- To change an *int* to *float*: Use the following syntax:  
 $(type-name) expression$   

```
int var1;
float var2;
var2 = (float)var1;
```
- This states  $\rightarrow$  Result of expression (**var1**) is to be a data type of **float**
- $(float)11 / 3 \rightarrow$  Forces entire expression to be evaluated in a floating point context, producing a result of **3.6666**
- $((int)7.5 * 2) / (long double)5$ 
  - The cast forces **7.5** to an **int 7**
  - Multiplication  $\rightarrow 7 * 2 = 14$  (integer context);
  - The cast forces **int 5** to a **long double 5.0**
  - Thus, **division** will be done in a **long double**, **14.0 / 5.0  $\rightarrow$  2.8**

22

C\_Prog\_Spring 2012

- **Note: Casts** only force conversion at their level,  
 – `(float)(40 / 17 * 13 / 3) → 8.0`  
 – `(float)((6 * 5) / +10 * (2 - 10)) → -24.000000`

#### More Examples:

```
int x, y;
y = 10;
x = (float)(y) / 4 * 10; → /* x = 25 */
x = (float)(y / 4) * 10; → /* x = 20 */
float f;
f = (int)(2.5 * 4.3); → /* f = 10.0 */
int NUM1 = 2;
float NUM2 = 3.6;
float RESULT;
RESULT = (float)NUM1 * NUM2; //Cast NUM1 into float
printf("Result is %f", RESULT);
```

23

C\_Prog. Spring 2012

## Type Definition

- Used to **define new data type names** to make a program more readable to the programmer;
- Lets us define our own identifiers that can be used in place of type specifiers such as **int, float, and double**:

```
typedef old_type_name new_type_name
```

- Usually we write:     *int money;*  
                               *money = 2;*

#### Examples:

```
typedef int Pounds;   /* redefines pounds as integer */
Pounds money = 2;   //pounds is used to define money as integer
```

#### Example:

```
typedef int counter; //redefines counter as an integer
counter j, n;   //Counter now used to define j and n as integers
```

24

C\_Prog. Spring 2012

## Characters and Strings

- **Strings:** Sequence of characters, surrounded by double quotations: "Hello World!", "I Love Programming!"
- Will deal with strings later as an array of characters.
- **%s** can be used as a placeholder to print strings.
  - `printf("A string: %s", "Hello, World");`
  - To in input strings, use scanf: `scanf("%s", word);`
- **Using** conversion specification gives programmer more control over appearance of output, of the form: `%m.pX` OR `%-m.pX`
  - X** = is the conversion specifier, Ex: *f, d, c, etc.*
  - m** = *Minimum field width* to be printed out (int. const.) → Optional;
  - P** = *Precision (int. const.)* → Optional; if omitted (*the fraction part*), the period (.) separating **m** and **p** is also omitted;
  - = *left justification*
  - `%13.2f` so **m=13, p=2** and **X=f**
  - `%10f` so **m=10, X=f** where **p & .** Are omitted
  - `%.2f` so **p=2, X=f** and **m** is omitted

25

C\_Prog. Spring 2012

- The conversion places the printable value into minimum **m** columns (taking more than **m** if needed) & justifying to the right if fewer than **m**
  - `x= 978: printf("%4d", x);` → `*978` (\* is white space)
  - `x= 978: printf("%-4d", x);` → `978*` (*left justify number*)
  - `x= 7452: printf("%3d", x);` → `7452`
  - `x= 7452: printf("%.2d", 9);` → `09`
- Recalling Conversion Specification:
  - `printf("%g\n", .123456);` → `0.123456` (as is)
  - `printf("%g\n", .1234567);` → `0.123457` (rounding-more than 6 digits)
  - `printf("%g\n", .00000123);` → `1.23e-006` (exponential format)

26

C\_Prog. Spring 2012

```
int i = 40;
```

```
float x = 839.21;
```

```
printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
```

```
printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);
```

- **scanf** processes the control string from left to right and each time it reaches a specifier it tries to interpret what has been typed as a value
- If you input multiple values then these are **assumed** to be separated by **white space** → (i.e spaces, newline or tabs). That is:

```
3 4 5
```

```
3
```

```
4
```

```
5
```

It doesn't matter how many spaces are included between items

Or even enters does not matter here, at input

27

C\_Prog. Spring 2012

- `scanf("%d %d", &i, &j);`  
 → Will read in two integer values into i and j.  
 → The integer values can be typed on the same line or on different lines as long as there is at least one *white space* character between them
- `scanf("%d%d%f%f", &x, &y, &z, &w);`  
 → If user enters any of the following:

```
12
```

```
-34 .41  
-11.1e5
```

**scanf** will see it as a stream of numbers & reads it successfully ... ignoring any space.

```
12-34.41-11.1e5
```

**scanf** will again read it successfully although no space between inputs! **WHY?**

C\_Prog. Spring 2012

## Character Handling Library <ctype.h>

| Prototype                           | Description  |
|-------------------------------------|--|
| <code>int isdigit( int c );</code>  | Returns true if c is a digit and false otherwise.  |
| <code>int isalpha( int c );</code>  | Returns true if c is a letter and false otherwise.   |
| <code>int isalnum( int c );</code>  | Returns true if c is a digit or a letter and false otherwise.  |
| <code>int isxdigit( int c );</code> | Returns true if c is a hexadecimal digit character and false otherwise.  |
| <code>int islower( int c );</code>  | Returns true if c is a lowercase letter and false otherwise.   |
| <code>int isupper( int c );</code>  | Returns true if c is an uppercase letter; false otherwise.   |
| <code>int tolower( int c );</code>  | If c is an uppercase letter, tolower returns c as a lowercase letter. Otherwise, tolower returns the argument unchanged.   |
| <code>int toupper( int c );</code>  | If c is a lowercase letter, toupper returns c as an uppercase letter. Otherwise, toupper returns the argument unchanged.   |
| <code>int isspace( int c );</code>  | Returns true if c is a white-space character—newline ('\n'), space (' '), form feed ('\f'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v')—and false otherwise. |
| <code>int iscntrl( int c );</code>  | Returns true if c is a control character and false otherwise.  |
| <code>int ispunct( int c );</code>  | Returns true if c is a printing character other than a space, a digit, or a letter and false otherwise.  |
| <code>int isprint( int c );</code>  | Returns true value if c is a printing character including space (' ') and false otherwise.   |
| <code>int isgraph( int c );</code>  | Returns true if c is a printing character other than space (' ') and false otherwise.  |

29

C\_Prog. Spring 2012

```
// Using functions isdigit, isalpha, isalnum, and isxdigit
#include <stdio.h>
#include <ctype.h>
int main()
{
    printf( "%s\n%s\n%s\n\n", "According to isdigit:", According to isdigit:
    isdigit('8') ? "8 is a " : "8 is not a ", "digit", 8 is a digit
    isdigit('#') ? "# is a " : "# is not a ", "digit" ); # is not a digit

    printf( "%s\n%s\n%s\n%s\n\n", "According to isalpha:", According to isalpha:
    isalpha('A') ? "A is a " : "A is not a ", "letter", A is a letter
    isalpha('b') ? "b is a " : "b is not a ", "letter", b is a letter
    isalpha('&') ? "& is a " : "& is not a ", "letter", & is not a letter
    isalpha('4') ? "4 is a " : "4 is not a ", "letter" ); 4 is not a letter

    printf( "%s\n%s\n%s\n\n", "According to isalnum:", According to isalnum:
    isalnum('A') ? "A is a " : "A is not a ", "digit or a letter", A is a digit or a letter
    isalnum('8') ? "8 is a " : "8 is not a ", "digit or a letter", 8 is a digit or a letter
    isalnum('#') ? "# is a " : "# is not a ", "digit or a letter" ); # is not a digit or a letter

    printf( "%s\n%s\n%s\n\n", "According to
    isxdigit:",
    isxdigit('F') ? "F is a " : "F is not a ", "hexadecimal digit", F is a hexadecimal digit
    isxdigit('J') ? "J is a " : "J is not a ", "hexadecimal digit", J is not a hexadecimal digit
    isxdigit('7') ? "7 is a " : "7 is not a ", "hexadecimal digit", 7 is a hexadecimal digit
    isxdigit('$') ? "$ is a " : "$ is not a ", "hexadecimal digit", $ is not a hexadecimal digit
    isxdigit('f') ? "f is a " : "f is not a ", "hexadecimal digit" ); f is a hexadecimal digit

    return 0;
}
```

30

C\_Prog. Spring 2012

```

Using functions islower, isupper, tolower, toupper */
#include <stdio.h>
#include <ctype.h>
main()
{
    printf( "%s\n%s\n%s\n%s\n", "According to islower:",
    islower('p') ? "p is a " : "p is not a ", "lowercase letter",
    islower('P') ? "P is a " : "P is not a ", "lowercase letter",
    islower('5') ? "5 is a " : "5 is not a ", "lowercase letter",
    islower('!') ? "! is a " : "! is not a ", "lowercase letter");

    printf( "%s\n%s\n%s\n%s\n", "According to isupper:",
    isupper('D') ? "D is an " : "D is not an ", "uppercase letter",
    isupper('d') ? "d is an " : "d is not an ", "uppercase letter",
    isupper('8') ? "8 is an " : "8 is not an ", "uppercase letter",
    isupper('$') ? "$ is an " : "$ is not an ", "uppercase letter");

    printf( "%s%c\n%s%c\n%s%c\n%s%c\n",
    "u converted to uppercase is ", toupper('u'),
    "7 converted to uppercase is ", toupper('7'),
    "$ converted to uppercase is ", toupper('$'),
    "L converted to lowercase is ", tolower('L') );
    return 0; }

```

According to islower:  
 p is a lowercase letter  
 P is not a lowercase letter  
 5 is not a lowercase letter  
 ! is not a lowercase letter

According to isupper:  
 D is an uppercase letter  
 d is not an uppercase letter  
 8 is not an uppercase letter  
 \$ is not an uppercase letter

u converted to uppercase is U  
 7 converted to uppercase is 7  
 \$ converted to uppercase is \$  
 L converted to lowercase is l

31 C\_Prog. Spring 2012

```

Using functions isspace, iscntrl, ispunct, isprint, isgraph */
#include <stdio.h>
#include <ctype.h>
main()
{
    printf( "%s\n%s\n%s\n%s\n", "According to isspace:",
    "Newline", isspace('\n') ? " is a " : " is not a ",
    "whitespace character", "Horizontal tab",
    isspace('\t') ? " is a " : " is not a ",
    "whitespace character",
    isspace('%') ? "% is a " : "% is not a ", "whitespace character" );

    printf( "%s\n%s\n%s\n%s\n", "According to iscntrl:",
    "Newline", iscntrl('\n') ? " is a " : " is not a ",
    "control character", iscntrl('$') ? "$ is a " :
    "$ is not a ", "control character" );

    printf( "%s\n%s\n%s\n%s\n", "According to ispunct:",
    ispunct(';') ? "; is a " : "; is not a ",
    "punctuation character",
    ispunct('Y') ? "Y is a " : "Y is not a ",
    "punctuation character",
    ispunct('#') ? "# is a " : "# is not a ",
    "punctuation character");

```

According to isspace:  
 Newline is a whitespace character  
 Horizontal tab is a whitespace character  
 % is not a whitespace character

According to iscntrl:  
 Newline is a control character  
 \$ is not a control character

According to ispunct:  
 ; is a punctuation character  
 Y is not a punctuation character  
 # is a punctuation character

32 C\_Prog. Spring 2012

```

printf( "%s\n%s%s\n%s%s%s\n\n", "According to isprint:",
isprint('$') ? "$ is a " : "$ is not a ",
"printing character",
"Alert", isprint('\a') ? " is a " : " is not a ",
"printing character" );

printf( "%s\n%s%s\n%s%s%s\n", "According to isgraph:",
isgraph('Q') ? "Q is a " : "Q is not a ",
"printing character other than a space",
"Space", isgraph(' ') ? " is a " : " is not a ",
"printing character other than a space" );

return 0; }

```

According to isprint:  
\$ is a printing character  
Alert is not a printing character

According to isgraph:  
Q is a printing character other than a space  
Space is not a printing character other than a space