

BBM 101

Introduction to Programming I

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

```
int main()
```

```
{
```

```
    printf("Goodbye World\n")
```

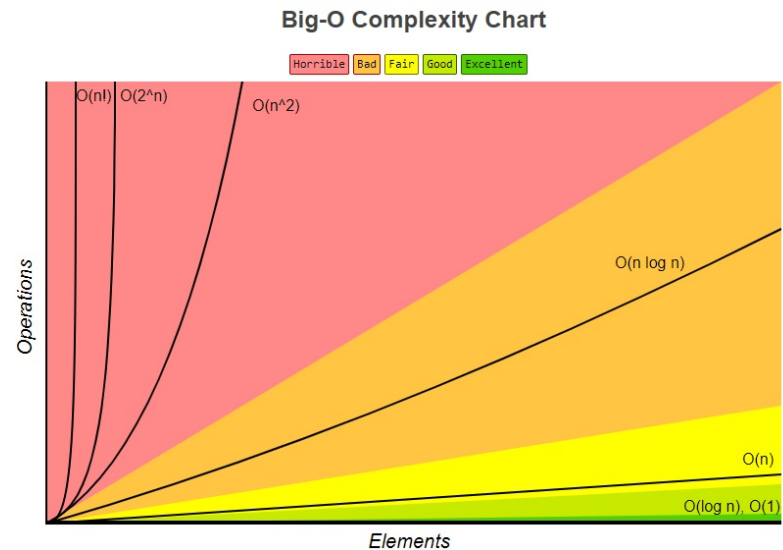
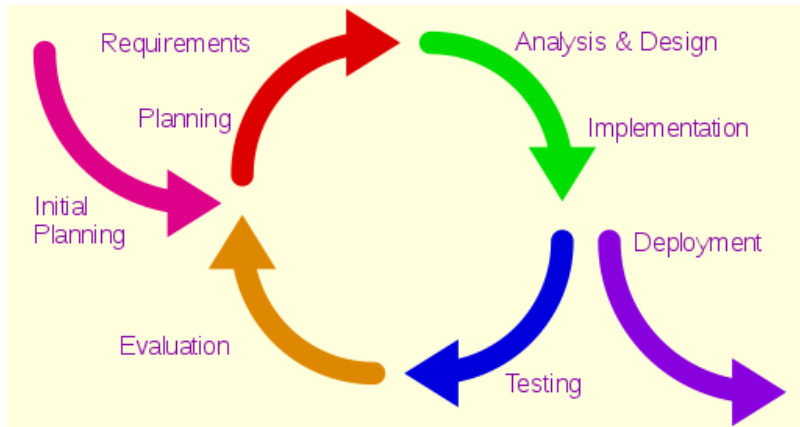
```
    return 0;
```

```
}
```

Lecture #10 – C for Python Programmers



Last time... How to Develop a Program, Algorithmic Complexity



Today

- How Python & C are similar
- How Python & C are different
- Variables, Operators, If-Else Statements, Arrays in C

Creating computer programs

- Each programming language provides a set of primitive operations
- Each programming language provides mechanisms for combining primitives to form more complex, but legal, expressions
- Each programming language provides mechanisms for deducing meanings or values associated with computations or expressions

Recall our goal

- Learn the syntax and semantics of a programming language
- Learn how to use those elements to translate “recipes” for solving a problem into a form that the computer can use to do the work for us
- Computational modes of thought enable us to use a suite of methods to solve problems

Recall: Dimensions of a PL

- **Low-level vs. High-level**

- Distinction according to the level of abstraction
- In low-level programming languages (e.g. Assembly), the set of instructions used in computations are very simple (nearly at machine level)
- A high-level programming language (e.g. C, Java) has a much richer and more complex set of primitives.

Recall: Dimensions of a PL

- **General vs. Targeted**

- Distinction according to the range of applications
- In a general programming language, the set of primitives support a broad range of applications.
- A targeted programming language aims at a very specific set of applications.
 - **e.g.**, MATLAB (matrix laboratory) is a programming language specifically designed for numerical computing (matrix and vector operations)

Recall: Dimensions of a PL

- **Interpreted vs. Compiled**

- Distinction according to how the source code is executed
- In interpreted languages (e.g. Python), the source code is executed directly at runtime (by the interpreter).
 - Interpreter control the the flow of the program by going through each one of the instructions.
- In compiled languages (e.g. C), the source code first needs to be translated to an object code (by the compiler) before the execution.
- More later today!

Recall: Dimensions of a PL

- **Functional**

- Treats computation as the evaluation of mathematical functions (e.g. Lisp, Scheme, Haskell, etc.)

- **Imperative**

- describes computation in terms of statements that change a program state (e.g. FORTRAN, BASIC, Pascal, C, etc.)

- **Logical (declarative)**

- expresses the logic of a computation without describing its control flow (e.g. Prolog)

- **Object oriented**

- uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs (e.g. C++, Java, C#, Python, etc.)

C (1973)

- Developed by Ken Thompson and Dennis Ritchie at AT&T Bell Labs for use on the UNIX operating system.
 - now used on practically every operating system
 - popular language for writing system software
- Features:
 - An extremely simple core language, with non-essential functionality provided by a standardized set of library routines.
 - Low-level access to computer memory via the use of pointers.
- C ancestors: C++, C#, Java

Python

- Created by Guido van Rossum in the late 1980s
- Allows programming in multiple paradigms: object-oriented, structured, functional
- Uses dynamic typing and garbage collection

Building a simple program in C (as compared to Python)

- Compilers versus interpreters
- Keywords
- Variable declarations
- Arrays
- Whitespace and Braces
- The `printf()` function
- The `scanf()` function
- If-Else Statements

Compilers versus interpreters

- One major difference between C and Python is how the programs written in these two languages are executed.
- With C programs, you usually use a *compiler* when you are ready to see a C program execute.
- By contrast, with Python, you typically use an *interpreter*.

Compilers versus interpreters

- An **interpreter** reads the user-written program and performs it directly.
- A **compiler** generates a file containing the translation of the program into the machine's native code.
 - The compiler does not actually execute the program!
 - Instead, you first execute the compiler to create a native executable, and then you execute the generated executable.

The Programming Process in C

- After creating a C program, executing it is a two step process:

```
me@computer:~$ gcc my_program.c -o my_program  
me@computer:~$ ./my_program
```

The Programming Process in C

```
me@computer:~$ gcc my_program.c -o my_program
```

```
me@computer:~$ ./my_program
```

- invokes the compiler, named *gcc*.
- The compiler reads the source file `my_program.c` containing the C codes
- It generates a new file named `my_program` containing a translation of this code into the binary code used by the machine.

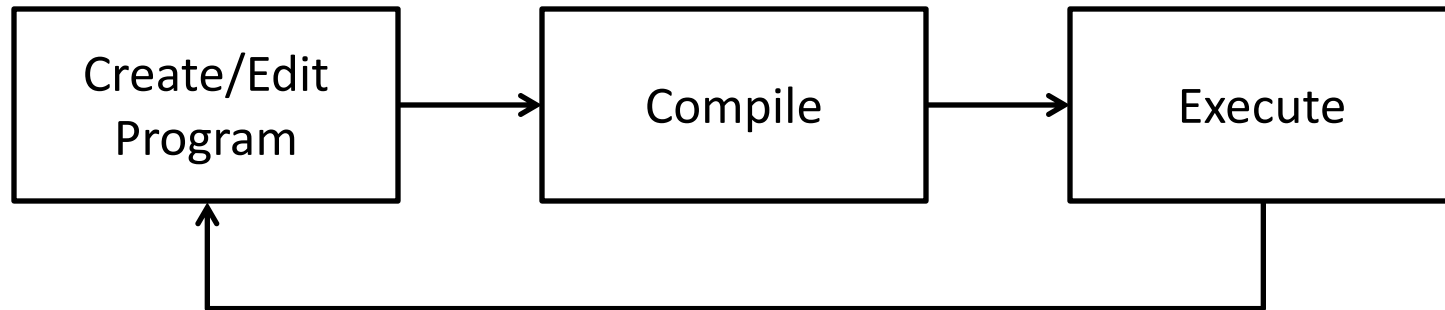
The Programming Process in C

```
me@computer:~$ gcc my_program.c -o my_program
```

```
me@computer:~$ ./my_program
```

- tells the computer to execute this binary code.
- As it is executing the program, the computer has no idea that `my_program` was just created from some C program.

The Programming Process in C



“The cycle ends once the programmer is satisfied with the program, e.g., performance and correctness-wise.”

Compilers versus interpreters

- An **interpreter** reads the user-written program and performs it directly.
- A **compiler** generates a file containing the translation of the program into the machine's native code.
- Being compiled has some radical implications to language design.
- C is designed so the compiler can tell everything it needs to know to translate the C program without actually executing the program.

Keywords

- 32 words defined as keywords in C
- have predefined uses and cannot be used for any other purpose in a C program

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
Const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

General Form of a C Program

Preprocessor directives

Declarations – variables
functions

```
main function {  
    declarations  
  
    statements  
}
```

A Simple C Program

```
/* Hello world! Example*/  
  
#include <stdio.h>  
  
int main()  
{  
    printf("Hello world!\n");  
    return 0;  
}
```

```
Hello world!
```

Identifiers

- A sequence of letters, digits, and the special character ‘_’ satisfying:

$$\text{identifier} = \alpha \{ \alpha + \# \}^*$$

with $\alpha = \{A, \dots, Z, a, \dots, z, _ \}$, $\# = \{0, \dots, 9\}$, and
* means “0 or more”

- Case-sensitive (as in Python)
e.g. `Ali` and `ali` are two different identifiers.
- Identifiers are used for:
 - Variable names
 - Function names

Identifiers (Cont.)

- Sample valid identifiers

x

a1

_xyz_33

integer1

Double

- Sample invalid identifiers

xyz.1

gx²

114West

int

pi*r*r

Variable declarations

- C requires **variable declarations**, informing the compiler about the variable before the variable is actually used.
- In C, the variable declaration defines the variable's **type**.
- **No such thing in Python!**

Declaring a Variable

- Declaring a variable is simple enough.
- You enter the variable's type, some whitespace, the variable's name, and a semicolon:

```
double x;
```

- Value assignment is similar to Python:

```
x=3;
```

- x will actually hold the floating-point value 3.0 rather than the integer 3.
- However, once you declare a variable to be of a particular type, **you cannot change its type!**

Declaring a Variable

- In C, variable declarations belong at the top of the function in which they are used.
- If you forget to declare a variable, the compiler will refuse to compile the program:
 - A variable is used but is not declared.
- To a Python programmer, it seems a pain to have to include these variable declarations in a program, though this gets easier with more practice.

Variable Declarations (Cont.)

- A declaration consists of a data type name followed by a list of (one or more) variables of that type:

```
char c;  
int ali, bora;  
float rate;  
double trouble;
```

- A variable may be initialized in its declaration.

```
char c = 'a';  
int a = 220, b = 448;  
float x = 1.23e-6;      /*0.00000123*/  
double y = 27e3;      /*27,000*/
```

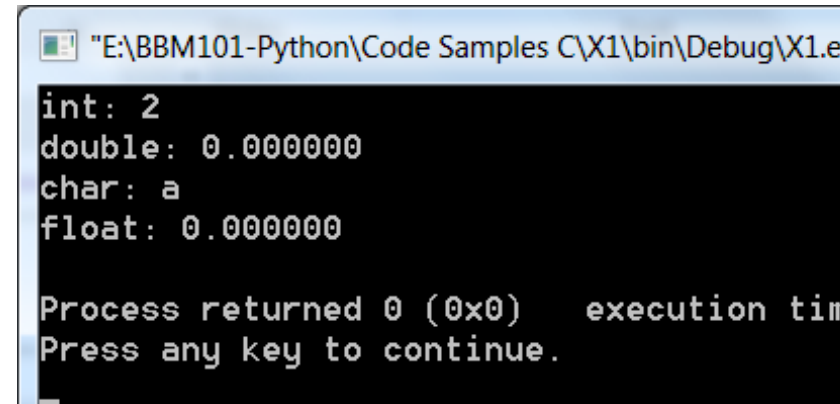
Variable Declarations (Cont.)

- Variables that are not initialized may have garbage values.

```
#include <stdio.h>

int main()
{
    int a;
    double b;
    char c;
    float d;

    printf("int: %d \n",a);
    printf("double: %lf \n",b);
    printf("char: %c \n",c);
    printf("float: %f \n",a);
    return 0;
}
```

A screenshot of a Windows command prompt window showing the output of a C program. The window title is "E:\BBM101-Python\Code Samples C\X1\bin\Debug\X1.e...". The output shows: "int: 2", "double: 0.000000", "char: a", and "float: 0.000000". Below this, it says "Process returned 0 (0x0) execution time" and "Press any key to continue." The output demonstrates that the variables 'a', 'b', 'c', and 'd' were not initialized, leading to unexpected values (2, 0.000000, 'a', and 0.000000 respectively).

Basic types in C

- C's list of basic types is quite constrained.
 - int** for an integer
 - char** for a single character
 - float** for a single-precision floating-point number
 - double** for a double-precision floating-point number
- Data Type Modifiers
 - **signed / unsigned**
 - **short / long**

Basic Data Types

Type	Size in Bytes	Range
signed char	1	-127 to +127
unsigned char	1	0 to 255
short int	2	-32,767 to +32,767
unsigned short int	2	0 to 65535
int	4	-32,767 to +32,767
unsigned int	4	0 to 65,535
long int	8	-2,147,483,647 to +2,147,483,647
unsigned long int	8	0 to 4,294,967,295
float	4	$\sim 10^{-37}$ to $\sim 10^{38}$
double	8	$\sim 10^{-307}$ to $\sim 10^{308}$
long double	16	$\sim 10^{-4931}$ to $\sim 10^{4932}$

No Boolean type for representing true/false

- This has major implications for a statement like **if**, where you need a test to determine whether to execute the body. C's approach is to treat the integer 0 as *false* and all other integer values as *true*.
- Example

```
int main() {
    int i = 5;
    if (i) {
        printf("in if\n");
    }
    else {
        printf("in else\n");
    }
    return 0;
}
```

prints "in if" when executed since the value of (**i**) is 5 which is not 0

No Boolean type in C!

- C's operators that look like they should compute Boolean values (like `==`, `&&`, and `||`) actually compute **int** values instead.
- In particular, they compute 1 to represent *true* and 0 to represent *false*.
- This means that you could legitimately type the following to count how many of a, b, and c are positive.

```
pos = (a > 0) + (b > 0) + (c > 0);
```

No Boolean type in C!

- **This quirk — that C regards all non-zero integers as true — is generally regarded as a mistake**, and it leads to confusing programs, so most expert C programmers avoid using the shortcut, preferring instead to **explicitly compare to zero** as a matter of good programming style.
- You may use external library «*stdbool.h*» or define your own constant (we'll see later) as true and false.

Arithmetic Operators

Major operators in C and Python

C operator precedence

++ -- (postfix)

+ - ! (unary)

* / %

+ - (binary)

< > <= >=

== !=

&&

||

= += -= *= /= %=

Python operator precedence

**

+ - (unary)

* / % //

+ - (binary)

< > <= >= == !=

not

and

or

- They look similar but there are some significant differences

Arithmetic Operators

- For arithmetic calculations
 - Use $+$ for addition, $-$ for subtraction, $*$ for multiplication and $/$ for division
 - Integer division truncates remainder
 - $7 / 5$ evaluates to 1
 - Modulus operator ($\%$) returns the remainder
 - $7 \% 5$ evaluates to 2
- Arithmetic operators associate left to right.
- Operator precedence
 - Example: Find the average of three variables a, b and c
 - Do not use: $a + b + c / 3$
 - Use: $(a + b + c) / 3$

Arithmetic Operators (Cont.)

Operator(s)	Operation(s)	Order of evaluation (precedence)
()	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they are evaluated left to right.
*, /, or %	Multiplication, Division, Modulus	Evaluated second. If there are several, they are evaluated left to right.
+ or -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Logical Operators

- `&&` (logical AND)
 - Returns true if both conditions are true
- `||` (logical OR)
 - Returns true if either of its conditions are true
- `!` (logical NOT, logical negation)
 - Reverses the truth/falsity of its condition
 - Unary operator, has one operand
- Useful as conditions in loops

<u>Expression</u>	<u>Result</u>
<code>true && false</code>	<code>false</code>
<code>true false</code>	<code>true</code>
<code>!false</code>	<code>true</code>

All Operators

Operators						Associativity	Type
++	--	+	-	!	(type)	right to left	unary
*	/	%				left to right	multiplicative
+	-					left to right	additive
<	<=	>	>=			left to right	relational
==	!=					left to right	equality
&&						left to right	logical AND
						left to right	logical OR
?:						right to left	conditional
=	+=	-=	*=	/=	%=	right to left	assignment
,						left to right	comma
Operator precedence and associativity.							

Operators in C – Important Distinctions

- C does not have an exponentiation operator like Python's `**` operator. For exponentiation in C, you'd want to use the library function `pow()`. For example, `pow(1.1, 2.0)` computes 1.1^2 .
- C uses symbols rather than words for the Boolean operations AND (`&&`), OR (`||`), and NOT (`!`).
- The precedence level of NOT (the `!` operator) is very high in C. This is almost never desired, so you end up needing parentheses most times you want to use the `!` operator.

Operators in C – Important Distinctions

- C's operators `++` and `--` are for incrementing and decrementing a variable. Thus, the statement `"i++"` is a shorter form of the statement `"i = i + 1"` (or `"i += 1"`).
- C's division operator `/` does integer division if both sides of the operator have an `int` type; that is, any remainder is ignored with such a division.
 - Thus, in C the expression `"13/5"` evaluates to `2`, while `"13/5.0"` is `2.6`: The first has integer values on each side, while the second has a floating-point number on the right.

Operators in C - Example

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

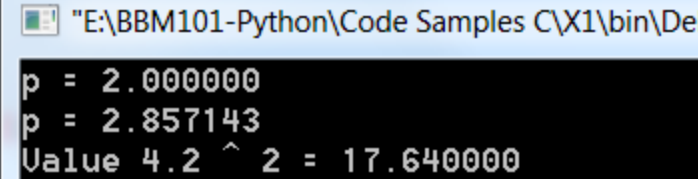
int main()
{
    int a = 5, b = 6, c = 10;
    float p;

    p = ((a == 5) + (b < 10) + (c > 15) ) * (c++)
/ (++b);
    printf("p = %f\n", p);

    a = 5, b = 6, c = 10;

    p = ((a == 5) + (b < 10) + (c > 15) ) * (c++)
/ (float) (++b);
    printf("p = %f\n", p);

    // you have to include math.h for pow function
    printf("Value 4.2 ^ 2 = %lf\n", pow(4.2, 2));
}
```



```
"E:\BBM101-Python\Code Samples C\X1\bin\De
p = 2.000000
p = 2.857143
Value 4.2 ^ 2 = 17.640000
```

For online C editor: [Tutorialspoint](https://www.tutorialspoint.com)

Operators in C – Important Distinctions

- C defines assignment as an operator, whereas Python defines assignment as a statement.
- The value of the assignment operator is the value assigned.
- A consequence of C's design is that an assignment can legally be part of another statement.
- Example:

```
#include <stdio.h>

int main()
{
    char a;
    while((a=getchar()) != 'e')
    {
        printf("char: %c\n", a);
    }
    return 0;
}
```

Type Conversion and Casting

- In an operation, if operands are of mixed data types, the compiler will convert one operand to agree with the other using the following hierarchy structure:

long double (highest)

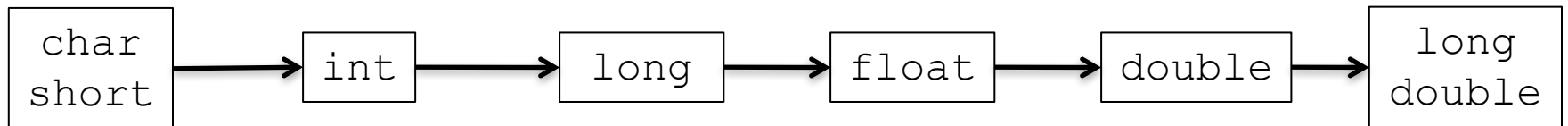
double

float

long

int

char/short (lowest)



Type Conversion and Casting (Cont.)

- implicit (automatic) type conversion
 - done automatically by the compiler whenever data from different types is intermixed.

– `int i;`

`double x = 17.7;`

`i = x;`

`i=17`

– `float x;`

`int i = 17;`

`x = i;`

`x=17.0`

Type Casting Example

```
#include <stdio.h>

int main()
{
    int a = 17;
    float b = 2.275;
    char c = 'B', d = 'e';

    printf("Integer value of b is %d\n" , (int)b);
    printf("Float value of a is %f\n", (float)a);
    printf("Int value(ASCII code) of c variable is %d\n", (int)c);
    printf("Double value(ASCII code) of d variable is %lf\n", (double)d);

    printf("\n\n");

    return 0;
}
```

"E:\BBM101-Python\Code Samples C\X1\bin\Debug\X1.exe"

```
Integer value of b is 2
Float value of a is 17.000000
Int value(ASCII code) of c variable is 66
Double value(ASCII code) of d variable is 101.000000
```

Whitespace

- In Python, whitespace characters like tabs and newlines are important:
 - You separate your statements by placing them on separate lines, and you indicate the extent of a block using indentation.
 - like the body of a **while** or **if** statement
- C does not use whitespace except for separating words.
- Most statements are terminated with a semicolon ';', and blocks of statements are indicated using a set of braces, '{' and '}'.

Whitespace

C fragment

```
disc = b * b - 4 * a * c;
if (disc < 0)
{
    num_sol = 0;
}
else
{
    t0 = -b / a;
    if (disc == 0)
    {
        num_sol = 1;
        sol0 = t0 / 2;
    }
    else
    {
        num_sol = 2;
        t1 = sqrt(disc) / a;
        sol0 = (t0 + t1) / 2;
        sol1 = (t0 - t1) / 2;
    }
}
}
```

Python equivalent

```
disc = b * b - 4 * a * c
if disc < 0:
    num_sol = 0
else:
    t0 = -b / a
    if disc == 0:
        num_sol = 1
        sol0 = t0 / 2
    else:
        num_sol = 2
        t1 = disc ** 0.5 / a
        sol0 = (t0 + t1) / 2
        sol1 = (t0 - t1) / 2
```


Whitespace

- As said, whitespace is insignificant in C.
- The computer would be just as happy if the previous code fragment is written as follows:

```
disc=b*b-4*a*c;if (disc<0) {  
num_sol=0;}else{t0=-b/a;if (  
disc==0) {num_sol=1;sol0=t0/2  
;}else{num_sol=2;t1=sqrt(disc/a;  
sol0=(t0+t1)/2;sol1=(t0-t1)/2;}}
```

- However, do not write your programs like this!

The `printf()` function

- In Python, displaying results for the user is accomplished by using `print`.
- In C, instead you use the `printf()` function which is provided by the C's standard library.
- The way the parameters to `printf()` work is a bit complicated but also quite convenient.

The `printf()` function

- The first parameter is a string specifying the format of what to print, and the following parameters indicate the values to print.
- Consider the following example:

```
printf("# solns: %d\n", num_sol);
```

- `"# solns: %d\n"` is the format string, `num_sol` is the value to be printed.
- The percent character is special to `printf()`.
 - It says to print a value specified in a subsequent parameter.
 - `%d` for integers/decimals
- If the value stored in `num_sol` is 2, the output is:

```
# solns: 2
```

The `printf()` function

- Like Python, C allows you to include escape characters in a string using a backslash:
 - The “`\n`” sequence represents the newline character,
 - The “`\t`” sequence represents the tab character,
 - “`\"`” sequence represents the double-quote character,
 - “`\\`” sequence represents the backslash character.
- These escape characters are part of C syntax, not part of the `printf()` function.

The `printf()` function

- Let's look at another example.

```
printf("N. of solns: %d\n", num_sol);  
printf("solns: %f, %f", sol0, sol1);
```

- Let's assume `num_sol` holds 2, `sol0` holds 4, and `sol1` holds 1.
- When the computer reaches these two `printf()` function calls, it executes them sequentially.
- The output is:

```
N. of solns: 2  
solns: 4.0, 1.0
```

The `printf()` function

- There's a variety of characters that can follow the percent character in the formatting string.
 - `%d`, as we've already seen, says to print an `int` value in decimal form.
 - `%f` says to print a `double` value in decimal-point form.
 - `%e` says to print a `double` value in scientific notation (for example, `3.000000e8`).
 - `%c` says to print a `char` value.
 - `%s` says to print a string.
- There's no variable type for representing a string, but C does support some string facilities using arrays of characters.

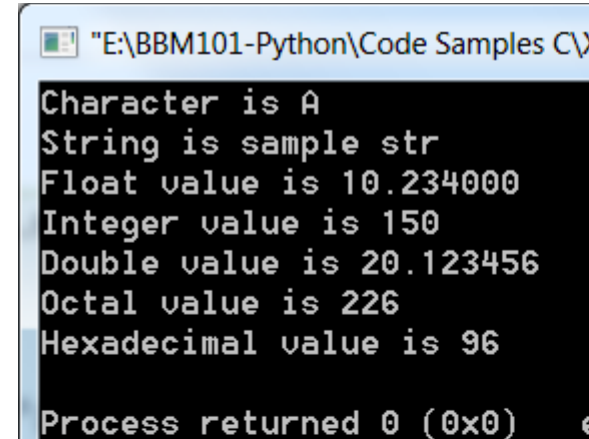
The `printf()` example

```
#include <stdio.h>

int main()
{
    char ch = 'A';
    char str[15] = "sample str";
    float flt = 10.234;
    int no = 150;
    double dbl = 20.123456;

    printf("Character is %c \n", ch);
    printf("String is %s \n", str);
    printf("Float value is %f \n", flt);
    printf("Integer value is %d\n", no);
    printf("Double value is %lf \n", dbl);
    printf("Octal value is %o \n", no);
    printf("Hexadecimal value is %x \n", no);

    return 0;
}
```



```
"E:\BBM101-Python\Code Samples C\
Character is A
String is sample str
Float value is 10.234000
Integer value is 150
Double value is 20.123456
Octal value is 226
Hexadecimal value is 96
Process returned 0 (0x0)
```

For online C editor:
[Tutorialspoint](https://www.tutorialspoint.com/)

The `scanf ()` function

- In Python, obtaining value from the user (by standard input) is accomplished by using `input`.
- In C, instead you use the `scanf ()` function which is provided by the C's standard library.
- The way the parameters to `scanf ()` work is quite similar with `printf ()` function.

The `scanf ()` function

- The first parameter is a string specifying the format of what to print to user.
- Consider the following example:

```
int age;  
printf("Enter your age:");  
scanf("%d", &age);
```

- The format specifier `%d` is used in `scanf ()` statement. So that, the value entered is received as an integer and `%s` for string.
- Ampersand is used before variable name "`ch`" in `scanf ()` statement as `&ch`.

The `scanf ()` function

- `&` character means, store the value on the variables address position on memory. We will talk about it later.

```
printf("Enter any character \n");
scanf("%c", &ch);
printf("Entered character is %c \n", ch);
printf("Enter any string (upto 100 character )\n");
scanf("%s", &str);
printf("Entered string is %s \n", str);
int anInt; long l;
scanf("%d %ld", &anInt, &l);
```

If Else Statements

- **An `if` statement**

- Works very similarly to Python's `if` statement

- The only major difference is the syntax:

- In C, an `if` statement's condition must be enclosed in parentheses, there is no colon following the condition, and the body has a set

- of braces enclosing it.

- As we've already seen, C does not have an `elif` clause as in Python; instead, C programmers use the optional-brace rule and write “`else if`”.

- Example:

```
if (x < 0) { printf("negative"); }
```

Braces (example on If Else)

- Several statements, like the **if** statement, include a body that can hold multiple statements.
- Typically the body is surrounded by braces ('{' and '}') to indicate its extent. But when the body holds only a single statement, the braces are optional.
- Example:

```
if (first > second)
    max = first;
else
    max = second;
```

Braces (example on If Else)

- C programmers use this quite often when they want one of several **if** tests to be executed.
- Example:

```
disc = b * b - 4 * a * c;  
if (disc < 0) {  
    num_sol = 0;  
}  
else {  
    if (disc == 0) {  
        num_sol = 1;  
    }  
    else {  
        num_sol = 2;  
    }  
}
```

Notice that the **else** clause here holds just one statement (an **if...else** statement), so we can omit the braces around it.

Braces (example on If Else)

- C programmers use this quite often when they want one of several **if** tests to be executed.
- Example:

```
disc = b * b - 4 * a * c;  
if (disc < 0) {  
    num_sol = 0;  
}  
else  
    if (disc == 0) {  
        num_sol = 1;  
    }  
    else {  
        num_sol = 2;  
    }
```

But this situation arises often enough that C programmers follow a special rule for indenting in this case — a rule that allows all cases to be written at the same level of indentation.

Braces (example on If Else)

- C programmers use this quite often when they want one of several **if** tests to be executed.
- Example:

```
disc = b * b - 4 * a * c;
if (disc < 0) {
    num_sol = 0;
}
else if (disc == 0) {
    num_sol = 1;
}
else {
    num_sol = 2;
}
```

Arrays

- Python supports many types that combine the basic atomic types into a group: tuples, lists, strings, dictionaries, sets.
- C's support is much more rudimentary: The *only* composite type is the **array**
 - Similar to Python's list except that an array in C cannot grow or shrink — its size is fixed at the time of creation.

- Example:

```
double pops[50];  
pops[0] = 897934;  
pops[1] = pops[0] + 11804445;
```

- Another way to make an array, if you know all the elements upfront, is:

```
char vowels[6] = {'a', 'e', 'i', 'o', 'u', 'y'};
```


Defining Arrays

- When defining arrays, specify
 - Name
 - Type of array
 - Number of elements

```
arrayType arrayName [numberOfElements] ;
```

Examples:

```
int c[10];  
float myArray[3284];
```

- Defining multiple arrays of same type
 - Format similar to regular variables
 - Example:

```
int b[100], x[27];
```

Defining Arrays

```
double x[8];
```

x[0] x[1] x[2] x[3] x[4] x[5] x[6] x[7]

16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5
------	------	-----	-----	-----	------	------	-------

```
i=5
```

```
x[i-1] = x[i]
```

```
x[i] = x[i+1]
```

```
x[i]-1 = x[i]
```

Illegal assignment statement!

Examples Using Arrays

- Initializers

```
int n[5] = {1, 2, 3, 4, 5};
```

- If not enough initializers, rightmost elements become 0

```
int n[5] = { 0 }
```

All elements 0

- C arrays have no bounds checking

- If size omitted, initializers determine it

```
int n[ ] = { 1, 2, 3, 4, 5 };
```

- 5 initializers, therefore 5 element array

Arrays

- C does not have an support for accessing the length of an array once it is created; that is, there is nothing analogous to Python's `len(pops)`
 - **But you may find the length by `sizeof` function. We'll discuss it later.**
- What happens if you access an array index outside the array, like accessing `pops[50]` or `pops[-100]`?
 - With Python, this will terminate the program with a friendly message pointing to the line at fault and saying that the program went beyond the array bounds.
 - C is not nearly so friendly. **When you access beyond an array bounds, it blindly does it.**

Examples Using Arrays

```
double x[8];
```

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5

```
i=5
```

```
printf("%.1f", x[i]+1);           13.0
printf("%.1f", x[i]+i);           17.0
printf("%.1f", x[i+1]);           14.0
printf("%.1f", x[i+i]);           invalid
printf("%.1f", x[2*i]);           invalid
printf("%.1f", x[2*i-3]);         -54.5
printf("%.1f", x[(int)x[4]]);     6.0
printf("%.1f", x[i++]);           12.0
printf("%.1f", x[--i]);           12.0
```

Arrays example

```
#include <stdio.h>

int main()
{
    int a[10];

    printf(" Element at index %d : %d \n", 0 , a[0]);
    printf(" Element at index %d : %d \n", 1 , a[1]);
    printf(" Element at index %d : %d \n", 55 , a[55]);
    printf(" Element at index %d : %d \n", -50 , a[-50]);
}
```

"E:\BBM101-Python\Code Samples C\X1\bin\Debug

```
Element at index 0 : -2
Element at index 1 : 1954419042
Element at index 55 : 2686956
Element at index -50 : 1955014944
```

For online C editor: [Tutorialspoint](https://www.tutorialspoint.com)

Arrays – Warning!

- **The lack of array-bounds checking can lead to very difficult bugs**, where a variable's value changes mysteriously somewhere within hundreds of functions, and you as the programmer must determine where an array index was accessed out of bounds. This is the type of bug that takes a lot of time to uncover and repair.
- **Every once in a while, you'll see a C program crash, with a message like “segmentation fault”**. It won't helpfully include any indication of what part of the program is at fault: all you get is those two words. Such errors usually mean that the program attempted to access an invalid memory location. **This may indicate an attempt to access an invalid array index.**

Comments

- In C's original design, all comments begin with a slash followed by an asterisk (“*/**”) and end with an asterisk followed by a slash (“**/*”).
- The comment can span multiple lines.
- Example:

```
/* gcd - returns the greatest common  
 * divisor of its two parameters */  
int gcd(int a, int b) {  
    ...
```


Comments

- C++ introduced a single-line comment that has proven so handy that most of today's C compilers also support it.
- It starts with two slash characters (“//”) and goes to the end of the line.
- Example:

```
...  
  
// assign sth. to x  
x= a * r * r;  
  
...
```

Constants

- `#define` directive tells the preprocessor to substitute all future occurrences of some word with something else.
- You can use `const` prefix to declare constants.
- Example:

```
#define PI 3.14159  
  
printf("area: %f\n", PI * r * r);
```

- The preprocessors automatically translate the above expression into:

```
printf("area: %f\n", 3.14159 * r * r);
```

Constants – True/False

```
#include <stdio.h>

#define TRUE 1
#define FALSE 0

int main()
{
    const int ANOTHER_TRUE = 1;
    const int ANOTHER_FALSE = 0;

    if(ANOTHER_TRUE == TRUE) {
        printf(" It's True!!! ");
    }else{
        printf(" It's False :( ");
    }
    return 0;
}
```