1
Basic C Programming

Why Learning C Programming Language?

- **Advantages**
  - Powerful, flexible, efficient, portable, structured, modular.
  - Enable the creation of well-structured programs.
  - Bridge to C++ (OO Programming).

Dennis Ritchie

- **Disadvantages**
  - Free style and **not strongly-typed**.
  - The use of ***pointers*** may confuse many students. However, ***pointers*** are powerful for building data structures.

Basic C Programming

- **Structure of a C Program**
- Data Types, Constants, Variables, Operators, Data Type Conversion, Mathematical Library
- Simple Input/Output

Structure of a C Program

A simple C program structure:

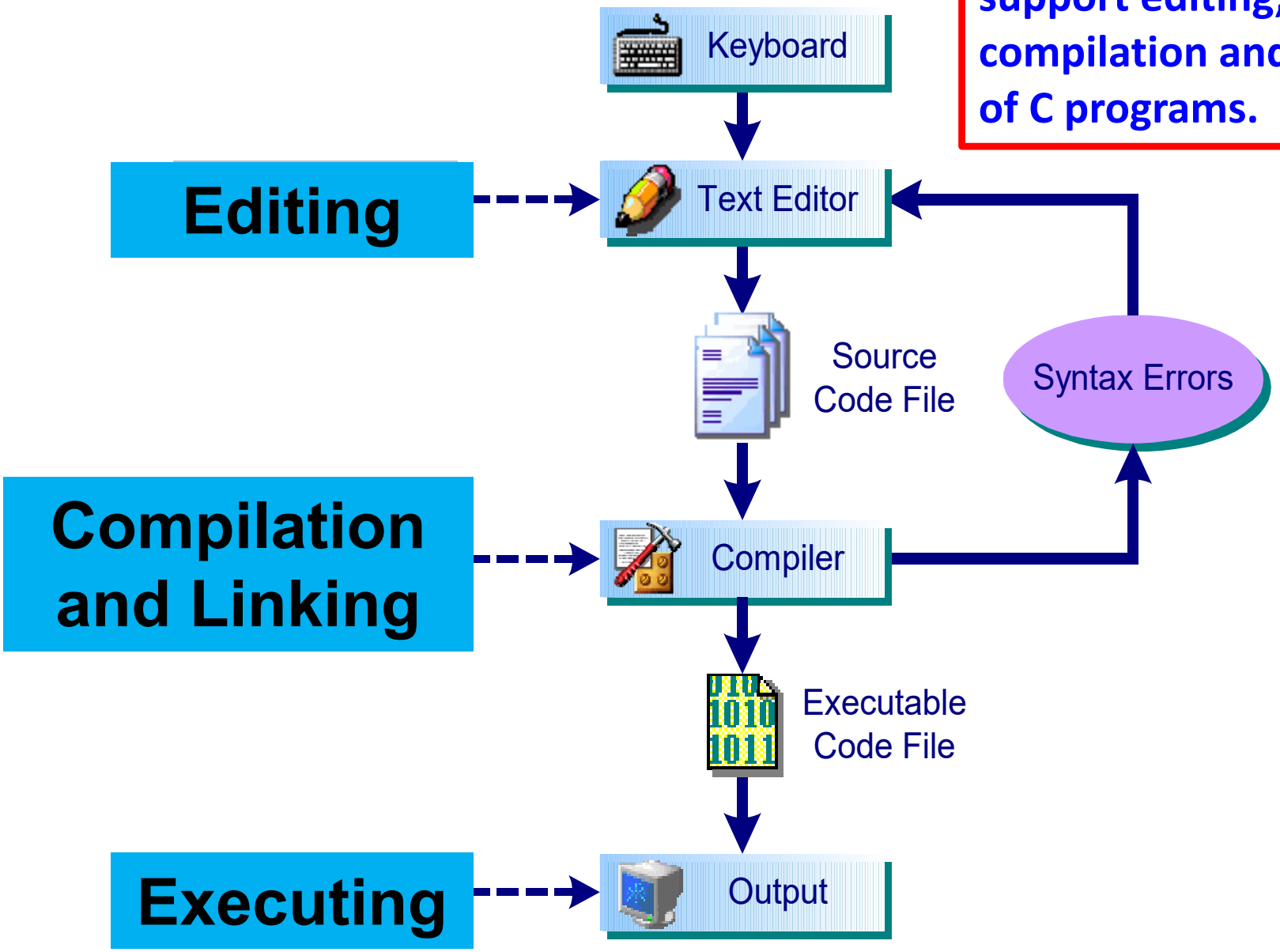
```
/* multi-line comment */
//  single line comment
preprocessor instructions
int main()
{
    statements;
    return 0;
}
```

An Example C Program

```
 /* Purpose: a program to
print Hello World! */
#include <stdio.h>
int main()
{   // begin body
    printf("Hello World! \n");
    return 0;
}   // end body
```

Development of a C Program

Note: IDE such as Code::Blocks provides an integrated environment to support editing, compilation and execution of C programs.



Basic C Programming

- Structure of a C Program
- **Data Types, Constants, Variables, Operators, Data Type Conversion, Mathematical Library**
- Simple Input/Output

Data and Types

- Data type determines the **kind of data** that a **variable** can hold, how many bytes of memory that are reserved for it and the operations that can be performed on it. (Note – the size in memory for the data type depends on machines.)
- There are mainly three kinds of data types: integers, floating points and characters.
- **Integers**
  - **int** (4 bytes or 2 bytes in some older systems)
- **Floating Points**
  - **float** (4 bytes – 32 bits)
  - **double** (8 bytes – 64 bits)
- **Characters**
  - **char** (1 byte – 8 bits)
  - 128 distinct characters in ASCII character set.

Note: Operations involving the **int** data type are always ***exact***, while the **float** and **double** data types can be ***inexact***. **E.g.**, the floating point number 2.0 may be represented as 1.9999999 internally.

Constants

- A constant is an object whose value is **unchanged** throughout the life of the program.
- Four types of constant values:
  - **Integer**: e.g. 100, –256; **Floating-point**: e.g. 2.4, –3.0;
  - **Character**: e.g. 'a', '+' ; **String**: e.g. "Hello Students "
- **Defining Constants – by using the preprocessor directive #define**

  Format: **#define** CONSTANT_NAME **value**

  E.g. **#define TAX_RATE** 0.12

```
/* define a constant TAX_RATE with 0.12 */
```

- **Defining Constants - by defining a constant variable**

  Format: **const type varName = value;**

  E.g. **const** float **pi = 3.14159**;

```
/* declare a float constant variable pi with
   value 3.14159 */
printf("pi  =  %f\n", pi);
```

ASCII Character Set (Character - 1 byte)

	0	1	2	3	4	5	6	7	8	9
0	NUL							BEL	BS	TAB
1	LF		FF	CR						
2								ESC		
3			SP	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

- **Character Constants**
  - '**A**' or 65
- **Non-printable Characters:**
  - '**\n**','**\t**', '**\a**'
- **Character vs String Constants**
  - '**a**' or "**a**"

Variables

- Variables are symbolic names that are used to store data in memory. A variable declaration always contains 2 components:
  - **data_type** (e.g. int, float, double, char, etc.)
  - **var_name** (e.g. count, numOfSeats, etc.)
- The syntax for variable declaration:

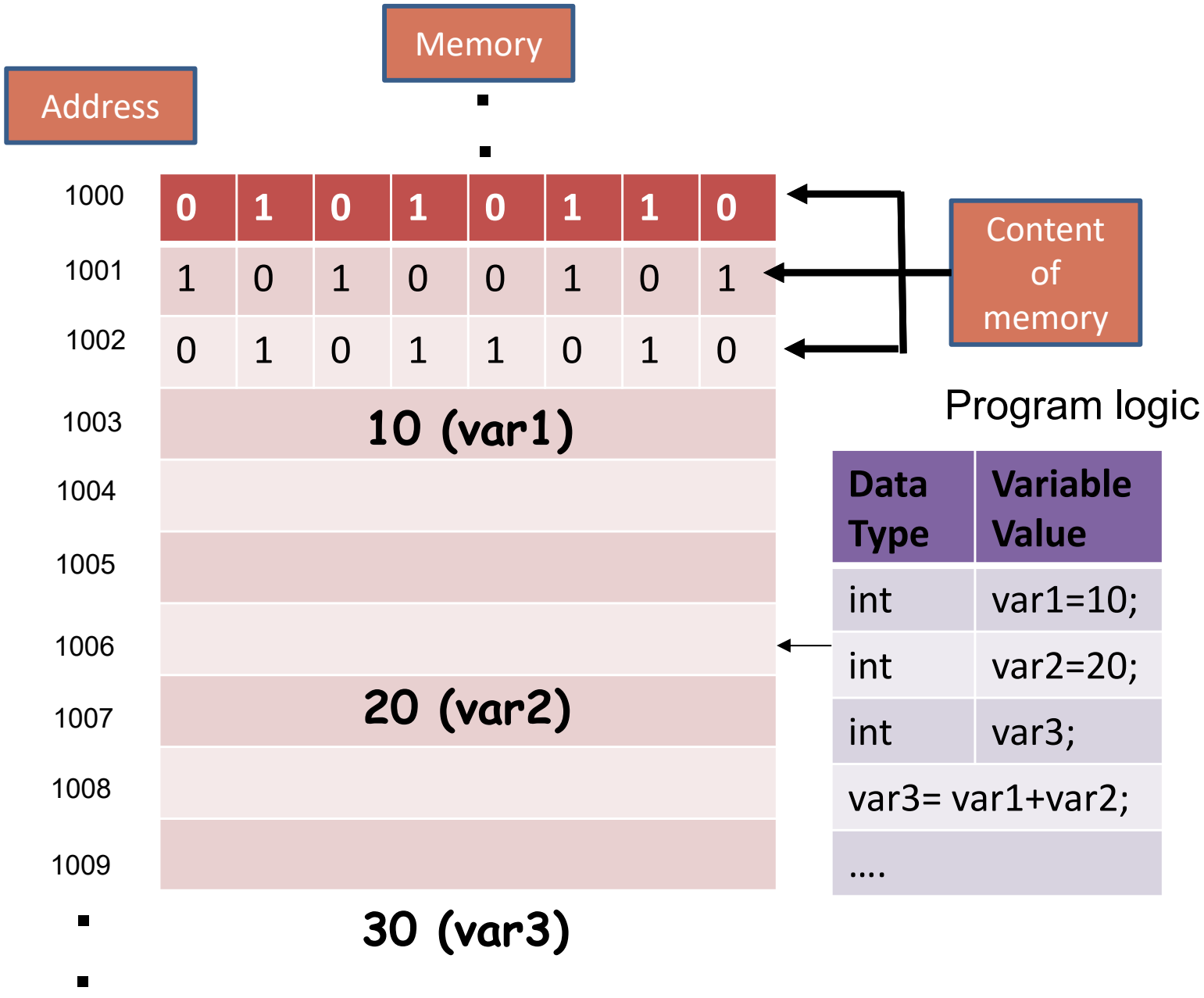
  data_type var_name[, **var_name**];
- Variables should be declared at the **beginning** of a function in your program. Examples of variable initializations:

```
int count = 20;
float temperature, result;
```

- C **keywords** are **reserved** and **cannot** be used as variable names:

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

Computer Memory and Variables



- Program needs to declare variables (or constants) to store data.
- Computer memories are then allocated for storing the declared variables or constants.

Operators

- Fundamental Arithmetic operators: +, -, *, /, %
  - E.g. 7/3 = 2; 7%3 = 1; 6.6/2.0=3.3;
- Assignment operators:
  - E.g. float amount = 25.50;
  - Chained assignment: E.g. a = b = c = 3;
- Arithmetic assignment operators: +=, -=, *=, /=,%=
  - E.g. a += 5;
- Relational operators: ==, !=, <, <=, >, >=
  - E.g. 7 >= 5
- Increment/decrement operators: ++, --
- Conditional operators: ?:

Increment Operators

- The increment operator increases a variable by 1. Two modes: *prefix* and *postfix*.
- In **prefix mode**: the format is **++**var_name
  (1) var_name is incremented by 1 and
  (2) the value of the expression is the **updated value** of var_name.
- In **postfix mode**: the format is var_name**++**
  (1) The value of the expression is the **current value** of var_name and
  (2) then var_name is incremented by 1.

```
#include <stdio.h>
int main()
{
        int     num = 4;
        printf("value of num is %d\n", num);
        num++;      // ++num; i.e., num = num+1;
        printf("value of num is %d\n", num);
        num = 4;
        printf("value of num++ is %d\n", num++);
        printf("value of num is %d\n",num);
        printf("value of ++num is %d\n", ++num);
        printf("value of num is %d\n\n",num);
        return 0;
}
```

Output

```
value of num is 4

value of num is 5

value of num++ is 4
value of num is 5
value of ++num is 6
value of num is 6
```

Decrement Operators

- The **decrement operator** (--) works in the same way as the increment operator (++), except that the variable is decremented by 1.
  - **Prefix mode (--var_name) -** decrement **var_name before** any operation with it.
  - **Postfix mode (var_name--) -** decrement **var_name after** any operation with it.

```
#include <stdio.h>
int main()
{
        int     num = 4;
        printf("value of num is %d\n", num);
        num--;   // same as --num;
        printf("value of num is %d\n", num);
        num = 4;
        printf("value of num-- is %d\n", num--);
        printf("value of num is %d\n", num);
        printf("value of --num is %d\n", --num);
        printf("value of num is %d\n", num);
        return 0;
}
```

Output

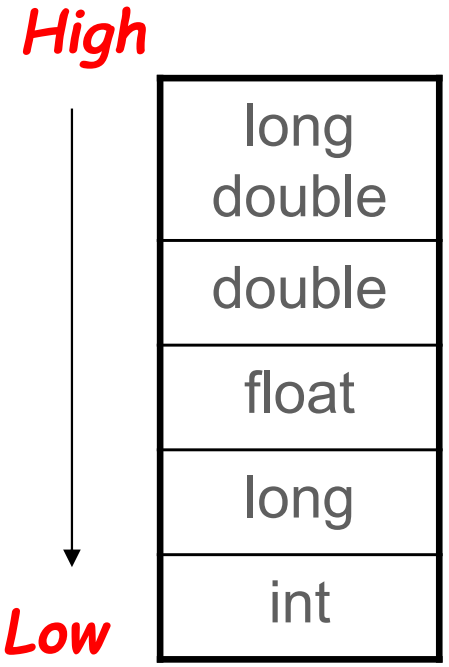
```
value of num is 4

value of num is 3

value of num-- is 4
value of num is 3
value of --num is 2
value of num is 2
```

Data Type Conversion

- Data type conversion: conversion of one data type into another type.
- Arithmetic operations require that **two numbers** in an expression/assignment are of the **same type**. E.g., the statement: **a = 2 + 3.5;** adds two numbers with different data types, i.e. *integer* & *floating point.* So conversion is needed.

- Three kinds of conversion are available:

1. ***Explicit conversion*** – it uses type casting operators, i.e. (int), (float), ..., etc.
   - e.g. (int)2.7 + (int)3.5
2. ***Arithmetic conversion*** - in mix operation, it converts the operands to the type of the **higher ranking** of the two.
   - e.g. double a; a = **2**+ 3.5; // 2 to 2.0 then add
3. ***Assignment conversion*** – it converts the type of the result of computing the expression to that of the type of the left hand side if they are different.
   - e.g. int b; b = **2.7 + 3.5**; // 6.2 to 6 then to b



Note: Possible *pit-falls* about type conversion -
Loss of precision: e.g. data conversion from **float** to **int**, the fractional part will be lost.

Mathematical Library

#include <math.h>

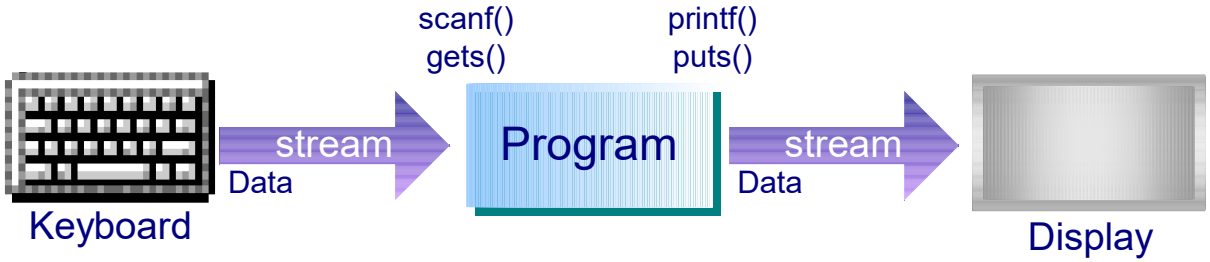
Function	Argument Type	Description	Result Type
`ceil(x)`	`double`	Return the smallest `double` larger than or equal to `x` that can be represented as an `int`.	`double`
`floor(x)`	`double`	Return the largest `double` smaller than or equal to `x` that can be represented as an `int`.	`double`
`abs(x)`	`int`	**Return the absolute value of `x`, where `x` is an `int`.**	`int`
`fabs(x)`	`double`	Return the absolute value of `x`, where `x` is a floating point number.	`double`
`sqrt(x)`	`double`	**Return the square root of `x`, where `x` >= 0.**	`double`
`pow(x,y)`	`double x, double y`	**Return x to the y power, x^y.**	`double`
`cos(x)`	`double`	Return the cosine of `x`, where `x` is in radians.	`double`
`sin(x)`	`double`	Return the sine of `x`, where `x` is in radians.	`double`
`tan(x)`	`double`	Return the tangent of `x`, where `x` is in radians.	`double`
`exp(x)`	`double`	Return the exponential of `x` with the base e, where e is 2.718282.	`double`
`log(x)`	`double`	Return the natural logarithm of `x`.	`double`
`log10(x)`	`double`	Return the base 10 logarithm of `x`.	`double`

Basic C Programming

- Structure of a C Program
- Data Types, Constants, Variables, Operators, Data Type Conversion, Mathematical Library
- **Simple Input/Output**

Simple Input/Output

- Most programs need to communicate with their environment. Input/output (or I/O) is the way a program communicates with the user. For C, the I/O operations are carried out by the I/O functions in the standard I/O libraries.
- Input from the keyboard or output to the monitor screen is referred to as standard input/output.



- The four simple Input/Output functions are:
  - **scanf()** and **printf()**: perform formatted input and output respectively.
  - **getchar()** and **putchar()**: perform character input and output respectively.
- The I/O functions are in the C library **<stdio>**. To use the I/O functions, we need to include the header file:

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

as the **preprocessor instruction** in a program.

Simple Output: printf()

The printf() statement has the format:

printf (control-string, argument-list);

- The **control-string** is a string constant. It is printed on the screen. It contains a **conversion specification** in which an item will be substituted for it in the printed output.
- The **argument-list** contains a list of items such as item1, item2, …, etc.
  - Values are to be substituted into places held by the conversion specification in the control string.
  - An item can be a constant, a variable or an expression like num1 + num2.
- The **number** of items must be the same as the number of conversion specifiers.
- The **type** of items must also match with the conversion specifiers.

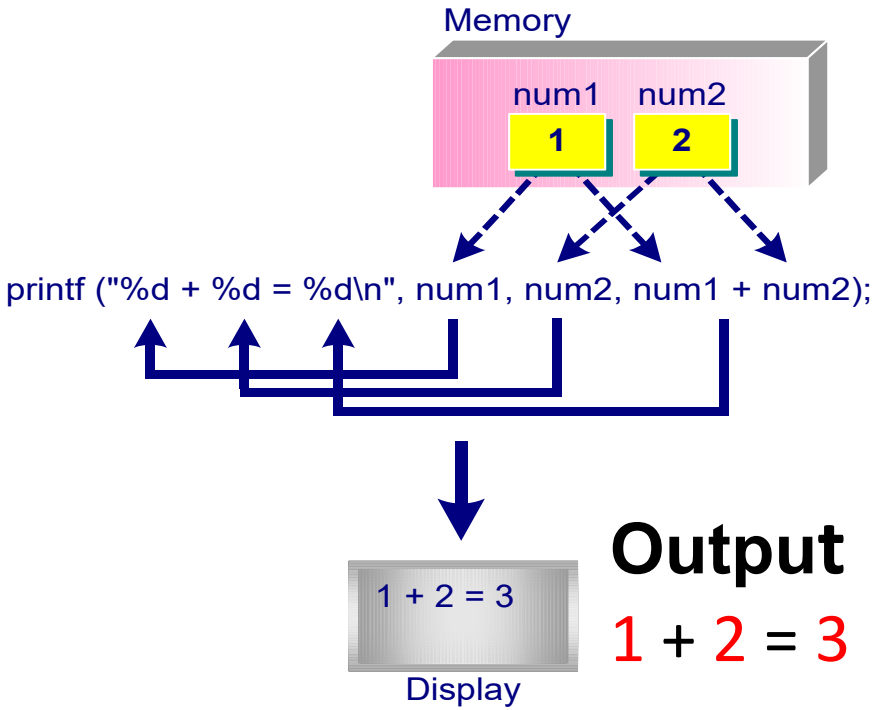
printf(): Example

The printf() statement has the format:

printf (control-string, argument-list);

```
#include <stdio.h>
int main()
{
    int num1 = 1, num2 = 2;
    printf("%d + %d = %d\n", num1, num2, num1+num2);
    return 0;
}
```

conversion specifiers



Output

1 + 2 = 3

Control-String: Conversion Specification

- A **conversion specification** is of the form

% [flag] [minimumFieldWidth] [.precision]conversionSpecifier

  - ***%*** and ***conversionSpecifier*** are compulsory. The others are optional.
  - Conversion specification specifies how the data is to be converted into displayable form.

Note:

- We will focus on using the compulsory options **%** and **conversionSpecifier.**
- If interested, please refer to the textbook for the other options such as *flag*, *minimumFieldWidth* and *precision*.

Control-String: Conversion Specification

Some common types of *Conversion Specifier*:

d	signed decimal conversion of int
o	unsigned octal conversion of unsigned
x,X	unsigned hexadecimal conversion of unsigned
c	single character conversion
f	signed decimal floating point conversion
s	string conversion

printf(): Example

```
#include <stdio.h>
int main( )
{
    int             num = 10;
    float           i = 10.3;
    double          j = 100.3456;
    printf(“int num = %d\n”, num);
    printf("float i = %f\n", i);
    printf("double j = %f\n", j);
      /* by default, 6 digits are printed
            after the decimal point */
    printf("double j = %.2f\n", j);
    printf("double j = %10.2f\n", j);
      /* formatted output */
    return 0;
}
```

Output

```
int num = 10
float i = 10.300000
double j = 100.345600

double j = 100.35
double j =      100.35
```

Simple Input: scanf()

- A **scanf()** statement has the format:

  scanf (control-string, argument-list);

- **control-string** - a string constant containing conversion specifications.
- The **argument-list** contains the **addresses** of a list of items.
  - The **items** in **scanf()** may be any variable matching the type given by the conversion specification. It cannot be a constant. It cannot be an expression like n1 + n2.
  - The **variable name** has to be preceded by an **&**. This is to tell **scanf()** the **address** of the variable so that **scanf()** can read the input value and store it in the variable's memory.
- **scanf()** uses **whitespace** characters (such as tabs, spaces and newlines) to determine how to separate the input into different fields to be stored.
- **scanf()** **stops reading** when it has read all the items as indicated by the control string or the **EOF** (end of file) is encountered.

scanf(): Example

- A scanf() statement has the format:

  scanf (control-string, argument-list);

```
#include <stdio.h>
int main( )
{
      int    n1, n2;
      float f1;
      double f2;
      printf(“Please enter 2 integers:\n”);
      scanf("%d %d", &n1, &n2);

      printf(“The sum = %d\n”, n1+n2);
      printf(“Please enter 2 floats:\n”);
      scanf("%f %lf", &f1, &f2);
      // Note: use %lf for double data
      printf(“The sum = %f\n”, f1+f2);
      return 0;
}
```

Output

```
Please enter 2 integers:
5 10
The sum = 15
Please enter 2 floats:
5.3 10.5
The sum = 15.800000
```

Character Input/Output

putchar()

- The syntax of calling putchar is

```
putchar(characterConstantOrVariable);
```

It is equivalent to

```
printf("%c", characterConstantOrVariable);
```

- The difference is that putchar is **faster** because printf() needs to process the control string for formatting. Also, it returns either the integer value of the written character or EOF if an error occurs.

getchar()

- The syntax of calling getchar is

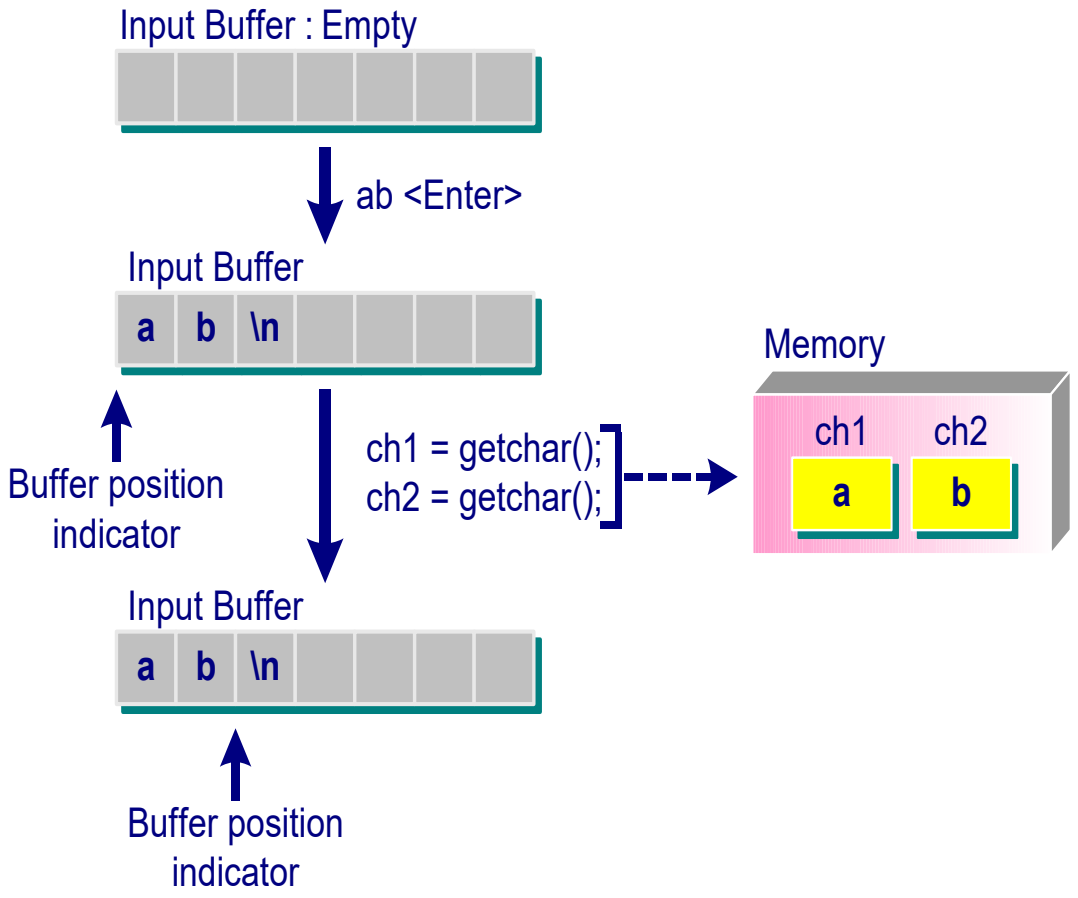
```
ch = getchar(); // ch is a character variable.
```

It is equivalent to

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

Character Input/Output: Example

```
/* example to use getchar() and putchar()  */
 #include <stdio.h>
 int main()
{
      char ch, ch1, ch2;
      putchar('1');
      putchar(ch='a');
      putchar('\n');
      printf("%c%c\n", 49, ch);
      ch1 = getchar();
      ch2 = getchar();
      putchar(ch1);
      putchar(ch2);
      putchar('\n');
      return 0;
}
```



Output

1a

1a

ab *(User Input)*

ab

Thank you !!!