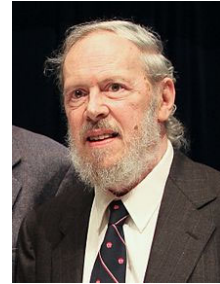# 1
Basic C Programming

1

Basic C Programming

1. In this chapter, we discuss the basic C programming concepts.

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.

2

Why Learning C Programming Language?

1. C programming language was created by Dennis Ritchie at AT&T Bell Laboratories in 1972.

2. The C programming language has a number of advantages over other conventional programming languages such as BASIC, PASCAL and FORTRAN. It is powerful, flexible, efficient, portable, structured and modular. It enables the creation of well-structured programs.

3. However, C also has a few disadvantages.

   - The free style of expression in C can make it difficult to read and understand. In addition, as C is not a strongly-typed language such as PASCAL and JAVA, it is the programmer's responsibility for ensuring the correctness of the program.

   - Pointer in C is a very useful feature. However, it can also cause programming errors that are difficult to debug and trace if it is not used properly. Nevertheless, these drawbacks can be overcome if good programming style is adopted.

4. C provides pointers that can be used for building data structures efficiently.

5. Also, C bridges well to C++ which is an object-oriented programming language that you will learn in second year of your study.

Basic C Programming

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

3

Basic C Programming

1. For basic C programming concepts, we will discuss the basic structure of a C program and the various components for a C program.

2. A sample C program is discussed to illustrate these basic components.

3. Then, we discuss the program development cycle for creating, compiling and executing a C program.

4. After that, data types, constants, variables, operators, data type conversion, mathematical library and simple input/output in C will be discussed.

5. Here, we start by discussing the basic structure of a C program.

Structure of a C Program

A simple C program structure:

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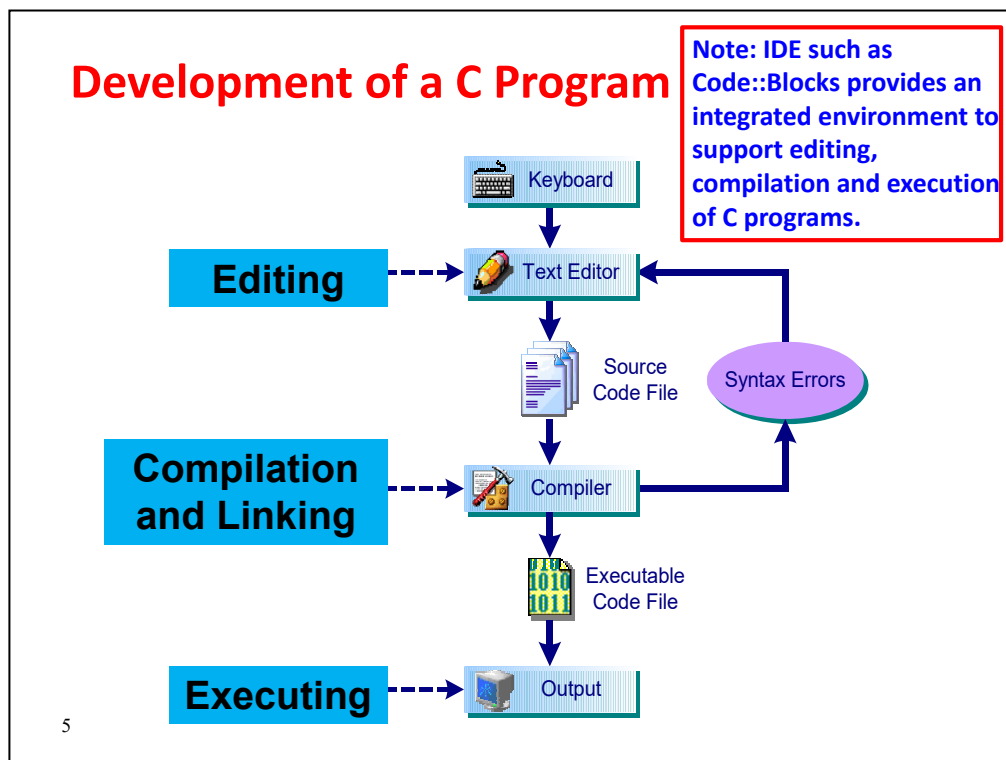
An Example C Program

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

4

Structure of a C Program

1. Here, we show a typical program structure which consists of comments, preprocessor instructions, main() function header, open brace, program statements, return statement and close brace.

2. An example C program is shown at the right hand side:

    1) The first two lines are the comments which state the purpose of the program (**/*** and ***/** can be used to enclose multiple lines comment).

    2) The **#include** preprocessor directive instructs the C compiler to add the contents of the header file **stdio.h** into the program during compilation. The **stdio.h** file is part of the standard library. It supports I/O operations that the program requires.

    3) In the **main()** function, it requires to return an integer value, so the keyword **int** is used to inform the C compiler about that.

    4) The braces **{ }** are used to enclose the **main()** function body.

    5) The line "**{ // begin body**" indicates the beginning of the function body with a comment.

    6) The **printf()** function is the library output function call to print a character string on the screen.

    7) The statement **return 0** returns the control back to the system.

    8) Finally, the last line "**} // end body**" indicates the end of the function body with a comment.

Development of a C Program

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

Editing → Keyboard → Text Editor → Source Code File

Syntax Errors

Compilation and Linking → Compiler → Executable Code File

Executing → Output

5

Development of a C Program

1. **Editing** - To develop a C program, a text editor is first used to create a file to contain the C source code. Most compilers come with editors that can be used to enter and edit source code.

2. **Compilation** - Then, the source code needs to be processed by a compiler to generate an object file. If syntax errors occur during compilation, we will need to rectify the errors, and compile the source code again until no further errors are occurred.

3. **Linking** – Then, the linker is used to link all the object files to create an executable file.

4. **Execution** - Finally, the executable file can be run and tested.

Basic C Programming

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

6

Basic C Programming

1. Here, we discuss data types, constants, variables, operators, data type conversion, and mathematical library.

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.

7

Data and Types

1. A data type describes the size (in terms of number of bytes in memory) of an object and how it may be used. Each type has its own computational properties and memory requirements. Therefore, you should select the data type for variables according to your data requirement.

2. There are three basic types of data in C for **characters**, **integers** and **floating point numbers**.

3. There are many data types defined in C as shown in the slide. However, we do not need to know all of them. We only need to focus on the following types for data in characters, integers and floating point numbers.

4. For characters, we can just use the type **char** which will take up 1 byte of memory.

5. For integers, we can just use the type **int** which will typically take up 4 bytes of memory in current computers.

6. For floating point numbers, apart from the data type **float** which takes up 4 bytes of memory, we can also use the type **double** which will typically take up 8 bytes of memory for storing larger floating point numbers.

7. It is important to note that operations involving the **int** data type are always *exact*, while the **float** and **double** data types can be *inexact*. For example, 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 */

  8

  　　　　printf("pi = %f\n", pi);

Constants

1. A constant is an object whose value is unchanged (or cannot be changed) throughout program execution.

2. There are four types of constants: integer constants, floating point constants, character constants and string constants.

3. When defining constants, we can use the preprocessor directive **#define**. The format of **#define** is **#define CONSTANT_NAME value.** For example, **#define TAX_RATE** 0.12.

4. Another way to define a constant is to use a constant variable. This is done by using the **const** qualifier as **const type variableName=value;** For example, **const** float **pi = 3.14159**;

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		

9

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

Characters - ASCII Set

1. Character takes 1 byte in the memory.
2. Character constants can be given by quoting the numerical value of a character, e.g. 65 for the character **A** in the ASCII character set, or by enclosing it with single quotes, e.g. '**A**'.
3. Some useful non-printable control characters are referred to by *escape sequence* which consists of the backslash symbol (****) followed by a single character. For example, '**\n**' represents the newline character, instead of using the ASCII number 10.
4. A **string** constant is a sequence of characters enclosed in double quotation marks. It is different from a character constant. For example, '**a**' and "**a**" are different as '**a**' is a character while "**a**" is a string. Strings will be discussed in the chapter on Character Strings.

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		

10

Variables

1. Variables are symbolic names that are used to store data in memory.

2. The syntax for variable declaration is **data_type var_name[, var_name]**;

3. During program execution, memory storage of suitable size is assigned for each variable according to its data type.

4. A variable must be declared first before it can be used in your program. Also all variables should be declared at the beginning of a function before writing program statements. Initialization of variables can also be done as part of a declaration.

5. Note that C keywords are reserved and cannot be used as variable names. Some examples of C keywords is shown in the slide.

2 October 2021
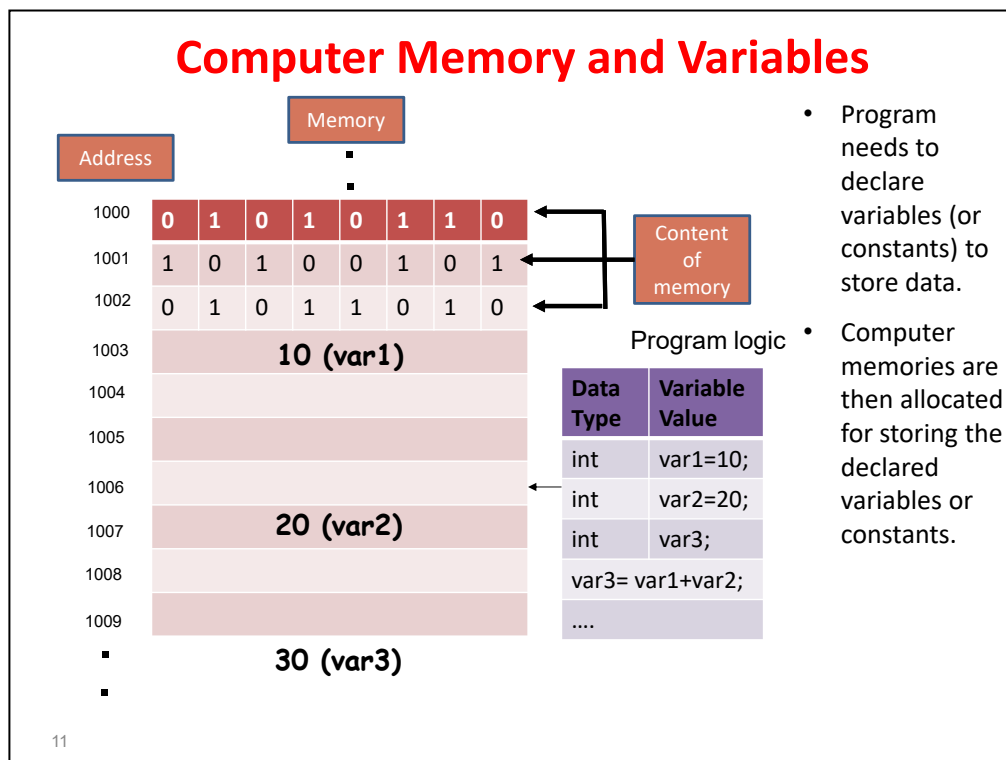
Computer Memory and Variables

1. A program needs to work with data which are stored in the main memory.

2. To do this, the program will need to declare variable to store the data.

3. In this example, three variables **var1**, **var2** and **var3** are declared. Note that **var1** and **var2** are declared with initialized values.

4. Once declared, the runtime system allocates the memory according to the corresponding data type.

5. In this example, 4 bytes of memory are allocated for the variables as they are of integer data type.

Dr Hui Siu Cheung, SCE, NTU   11

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: ?:

12

Operators

1. Operators in C are mainly classified into fundamental arithmetic operators, assignment operators, arithmetic assignment operators, increment/decrement operators, relational operators and conditional operators.

2. Arithmetic and assignment operators are quite straightforward.

3. In C, there are also increment and decrement operators, relational and conditional operators which will be discussed later.

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.

```c
#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

Increment Operators

1. The increment operator (**++**) increases a variable by 1. It can be used in two modes: *prefix* and *postfix*.

2. The format of the prefix mode is **++var_name;** where **var_name** is incremented by 1 and the value of the expression is the updated value of **var_name**.

3. The format of the postfix mode is **var_name++;** where the value of the expression is the current value of **var_name** and then **var_name** is incremented by 1.

4. Notice that one important difference between the prefix and postfix modes is on the time when that operation is performed. In the prefix mode, the variable is incremented before any operation with it, while in the postfix mode, the variable is incremented after any operation with it.

5. In the example program, it shows some examples on the use of increment operators. The initial value of the variable **num** is 4. The first **printf()** statement prints the value of 4 for **num**. The second **printf()** statement prints the value of 5 after increment. The variable **num** is assigned with the value 4 again. As the third **printf()** statement uses the postfix mode, the value of **num** is incremented by 1 after the printing has taken place. Therefore, the third **printf()** statement prints the value of 4 for **num**. The fourth statement then prints the value of 5. The fifth **printf()** statement uses the prefix mode. The value of **num** is incremented before the printing is taken place. Thus, it prints the value 6. The sixth statement also prints the value 6 for **num**.

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

14

Decrement Operators

1. The decrement operator (**--**) works in a similar way as the **increment** operator, except that the variable is decremented by 1.

2. It can be used in two modes: *prefix* and *postfix*.

   a) **--var_name** - decrement **var_name** before any operation with it in prefix mode.

   b) **var_name--** - decrement **var_name** after any operation with it in postfix mode.

3. The example code works similarly to that of the **increment** operator.

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 *c*onversion 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.

15

High

long double
double
float
long
int

Low

Data Type Conversion

1. Data type conversion is the conversion of one data type into another type.

2. Data type conversion is needed when more than one type of data objects appear in an expression/assignment. For example, the statement: **a = 2 + 3.5;** adds two numbers with different data types, i.e. *integer* and *floating point*. However, the addition operation can only be done if these two numbers are of the same data type.

3. There are three kinds of conversions that can be performed:
   a) First, explicit conversion which uses type casting operation, e.g. (int)2.7 and (int)3.5 convert float to integer values.

   b) Second, arithmetic conversion which converts the operands of an expression to the same data type in an arithmetic operation. In mix operation, it converts the operands to the type of the **higher ranking** of the two. For example, in the expression **2**+ 3.5; 2 will be converted to 2.0 for the arithmetic operation.

   c) Third, assignment conversion which converts to the data type of the result during assignment operation. 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. For example, in the assignment statement, b = **2.7 + 3.5**; the addition result 6.2 will be converted to 6 and assigned to b.

4. Note that there are possible pit-falls about type conversion, as it may cause the loss of precision. For example, 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`

16

Mathematical Library

1. All C compilers provide a set of mathematical functions in the standard library.

2. Some of the common mathematical functions include square root **sqrt()**, power **pow()**, and absolute values **abs()** and **fabs()**.

3. In order to use any of the mathematical functions, we need to place the preprocessor directive **#include <math.h>** at the beginning of the program.

Basic C Programming

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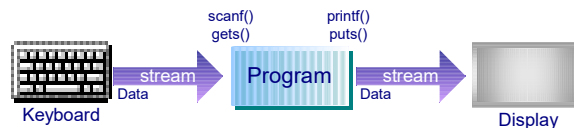- **Simple Input/Output**

17

Basic C Programming

1. Here, we discuss simple input/output in C.

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.

18

Simple Input/Output

1. 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.

2. Input from the keyboard or output to the monitor screen is referred to as standard input/output.

3. There are mainly four I/O functions, which communicate with the user's terminal:

    a) The **scanf()** and **printf()** functions perform formatted input and output respectively.

    b) The **getchar()** and **putchar()** functions perform character input and output respectively.

4. The standard I/O functions are in the library **<stdio>**. To use the I/O functions in **<stdio>**, the preprocessor directive **#include <stdio.h>** is included in order to include the header file in a program.

Simple Output: printf()

The printf() statement has the format:

printf (control-string, argument-list);

- The <u>control-string</u> 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 <u>argument-list</u> 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 <u>number</u> of items must be the same as the number of conversion specifiers.
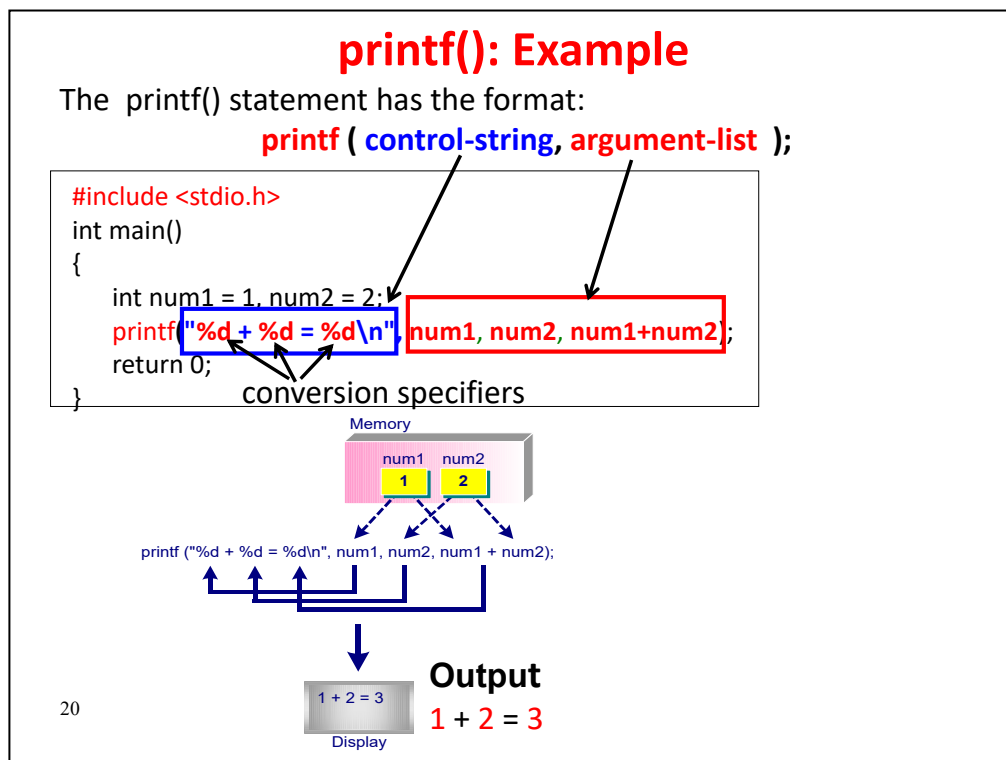- The <u>type</u> of items must also match with the conversion specifiers.

19

Simple Output: printf()

1. The **printf()** function allows us to control the format of the output. A **printf()** statement has the following format:

    printf(control-string, argument-list);

2. The **control-string** is a string constant such as **"%d + %d = %d\n"**. The string is then printed on the screen. It comprises the characters that are to be printed and the **conversion specification** such as **%d**.

3. The **argument-list** contains a list of items such as **num1**, **num2**, **num1+num2** to be printed. They contain values to be substituted into places held by the conversion specifications in the **control-string**. An item can be a constant, a variable or an expression like **num1+num2**.

4. Note that the number of items must be the same as the number of conversion specifications, and the **type** of the item must match with the **conversion specifier**.

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

Memory

num1   num2
  1   2

printf ("%d + %d = %d\n", num1, num2, num1 + num2);

Output

1 + 2 = 3

1 + 2 = 3

Display

20

Simple Output: printf() Example

1. The **printf()** function allows us to control the format of the output. A **printf()** statement has the following format:

    printf(control-string, argument-list);

2. The **control-string** is a string constant **"%d + %d = %d\n"**. The string is then printed on the screen. Two types of information are specified in the **control-string**. It comprises the characters that are to be printed and the **conversion specification** such as **%d**.

3. The **argument-list** contains a list of items such as **num1**, **num2**, **num1+num2** to be printed. They contain values to be substituted into places held by the conversion specifications in the **control-string**. An item can be a constant, a variable or an expression like **num1+num2**.

4. Note that the number of items must be the same as the number of conversion specifications, and the **type** of the item must match with the **conversion specifier**.

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*.

21

Conversion Specification

1. The format of a conversion specification is
   %[flag][minimumFieldWidth][.precision]conversionSpecifier, where **%** and **conversionSpecifier** are compulsory. The others are optional.

2. The **conversionSpecifier** specifies how the data is to be converted into displayable form.

3. Here, we focus only on using the compulsory options **%** and **conversionSpecifier.**

4. 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**

22

Conversion Specification

1. The most common types of conversion specifier are:
   a) d for decimal integers,
   b) c for characters,
   c) f for floating point numbers, and
   d) s for strings.

printf(): Example

```c
#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

23

An Example Program on using printf()

1. In the program, it prints an integer using conversion specification **%d**, and two floating point numbers using the conversion specification **%f** with different options.

2. The first **printf()** statement prints the integer number **num** which is 10.

3. The second and third **printf()** statements print the floating point numbers **i** and **j** using **%f**. The default precision of 6 is used, that is, six digits are printed after the decimal point.

4. The fourth and fifth **printf()** statements print formatted output.

5. The fourth **printf()** statement prints the floating point number with precision 2 using conversion specification **%.2f**, that is, only two digits are printed after the decimal point.

6. Similarly in the fifth **printf()** statement, it prints the floating point number using conversion specification %10.2f. That is, the field width is limited to 10 with precision 2. Also, the floating point number to be printed is right-justified in the field.

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.

24

Simple Input: The scanf() Function

1. The **scanf()** function is an input function that can be used to read formatted data.

2. A **scanf()** statement has the following format: **scanf(control-string, argument-list);**

3. The **control-string** is a string constant containing conversion specifications.

4. The **argument-list** contains the **addresses** of a list of input items. The input items may be any variables matching the type given by the conversion specification. The variable name has to be preceded by an address operator **&**. This is to tell **scanf()** the address of the variable so that **scanf()** can read the input value and store it in the memory that is allocated to the variable. Commas are used to separate each input item in the argument-list.

5. **scanf()** uses whitespace characters (such as tabs, spaces and newlines) to determine how to separate the input into different fields to be stored.

6. **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;
}
```

25

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

An Example on using scanf()

1. In the program, the first **scanf()** statement reads in two integer numbers. The user can enter the two numbers with a whitespace to separate them. However, carriage return (i.e. the enter key) can also be used to separate the two user input numbers.

2. It is important to note that the address operator (**&**) is required to be placed in front of the variable to indicate the memory address of the variable that is used to store the user data.

3. In the second **scanf()** statement, the **conversion specifier** "**%f**" is used to read a value in **float** data type, while the **conversion specifier** "**%lf**" is used to read a value in **double** data type.

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);

26

Character Input/Output

1. There are two functions in the *<stdio>* library to manage single character input and output: **putchar()** and **getchar()**.

2. The function **putchar()** takes a single argument, and prints the character. The syntax of calling **putchar()** is

    putchar(characterConstantOrVariable);

    which is equivalent to

    printf("%c", characterConstantOrVariable);

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

4. The **getchar()** function returns the next character from the input, or **EOF** if end-of-file is reached or if an error occurs. No arguments are required for **getchar()**. The syntax of calling **getchar()** is **ch = getchar();** where **ch** is a character variable. It is equivalent to **scanf("%c", &ch);**

Character Input/Output: Example

```c
/* 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;
}
```

Input Buffer : Empty

ab <Enter>

Input Buffer

| a | b | \n | | | |

Buffer position indicator

Memory

ch1 = getchar();
ch2 = getchar();

ch1   ch2

| a | b |

Input Buffer

| a | b | \n | | | |

Buffer position indicator
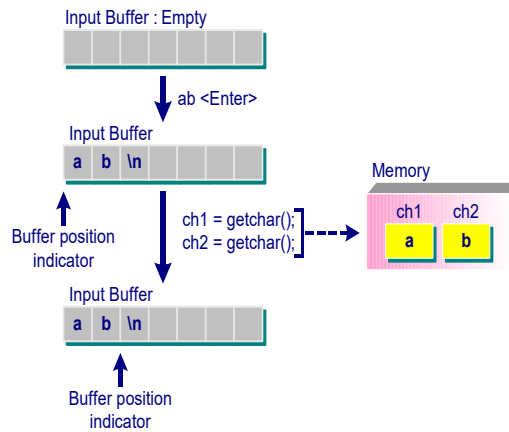
Output
1a
1a
ab   (User Input)
27   ab

Character Input/Output: Example

1. This slide gives an example on using single character input and output functions: **putchar()** and **getchar()**.

2. The **getchar()** function works with the input buffer to get user input from the keyboard. The input buffer is an array of memory locations used to store input data transferred from the user input. A *buffer position indicator* is used to keep track of the position where the data is read from the buffer.

3. The **getchar()** function retrieves the next data item from the position indicated by the buffer position indicator and moves the buffer position indicator to the next character position. However, the **getchar()** function is only activated when the **<Enter>** key is pressed.

4. Therefore, when a character is entered, the input buffer receives and stores the input data until the **<Enter>** key is encountered. The **getchar()** function then retrieves the next unread character in the input buffer and advances the buffer position indicator.

5. For example, in the program, when the user enters the data on the screen: **ab<Enter>,** the input data, namely '**a**', '**b**' and '**\n**', will then be stored in the input buffer. The buffer position indicator points at the beginning of the buffer. After reading the two characters, '**a**' and '**b**', with the statements:

    ch1 = getchar();

    ch2 = getchar();

the buffer position indicator moves two positions, and points to the buffer position that contains the newline '**\n**' character. As illustrated in this example, the two **getchar()** functions execute and read in the characters '**a**' and '**b**' only after the **<Enter>** key is pressed.

However, the newline character (**\n**) still remains in the input buffer that needs to be taken care of before processing another input request. One way to deal with the extra newline character is to flush the input buffer by using the statement **scanf('\n');** to read the newline character from the input buffer before the next input operation.

Thank You!

28

Thank You

1. Thanks for watching the lecture video.