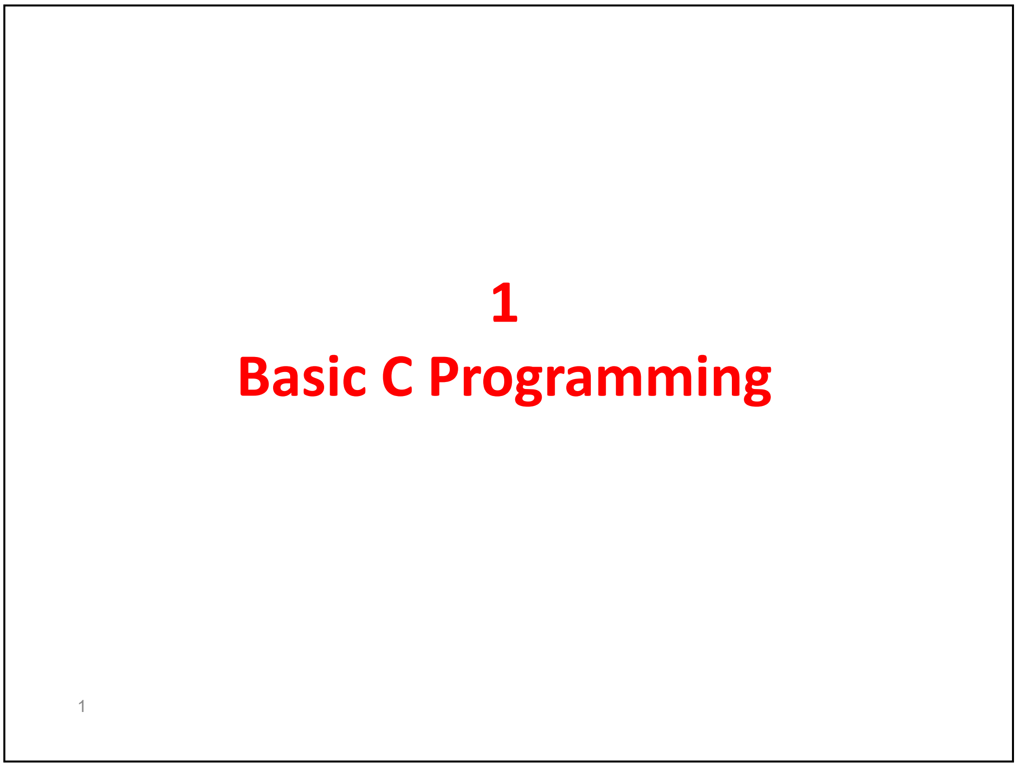
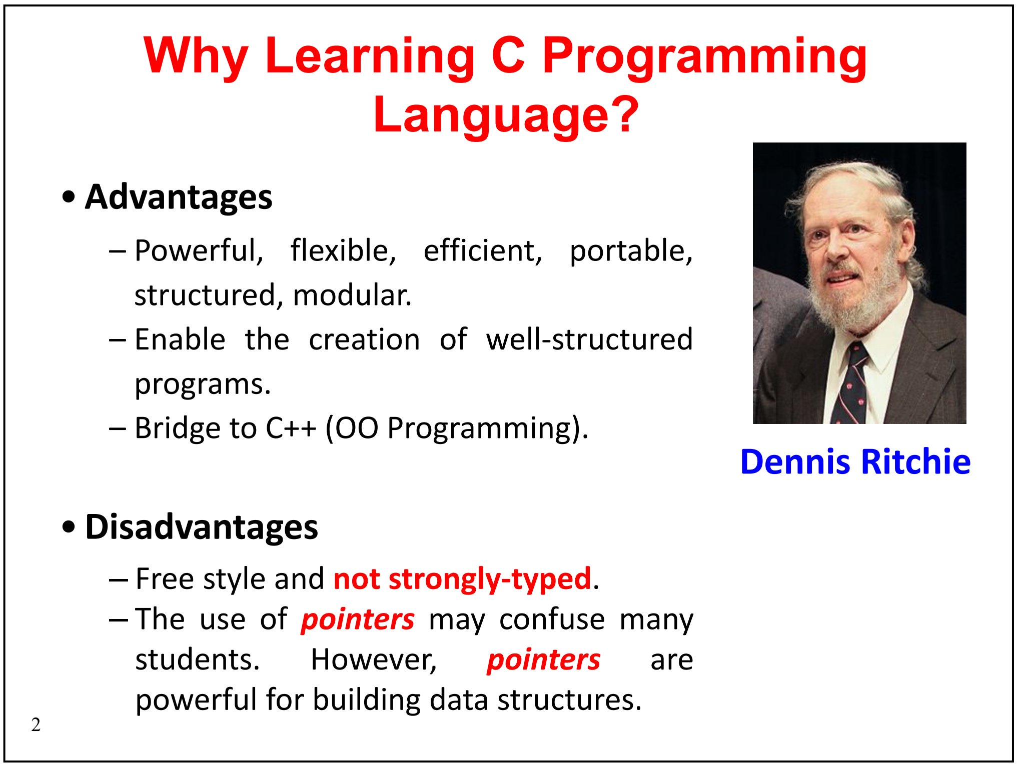


1
Basic C Programming

1

Basic C Programming

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



Why Learning C Programming Language?

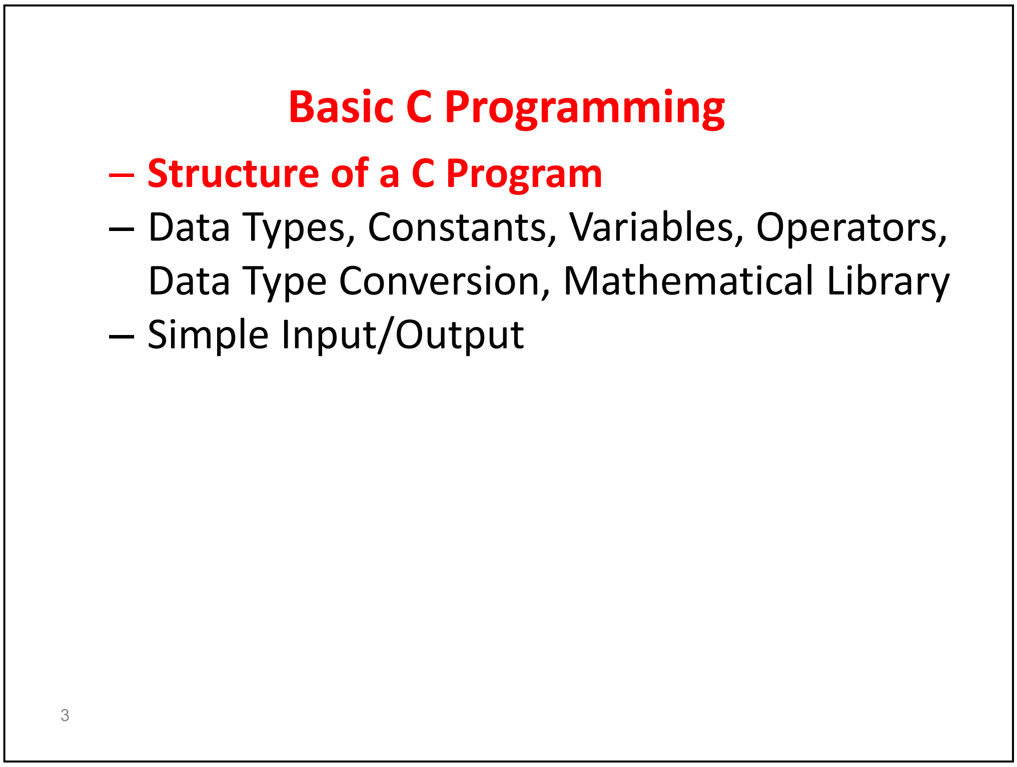
- **Advantages**
  - Powerful, flexible, efficient, portable, structured, modular.
  - Enable the creation of well-structured programs.
  - Bridge to C++ (OO Programming).
- **Disadvantages**
  - Free style and **not strongly-typed**.
  - The use of ***pointers*** may confuse many students. However, ***pointers*** are powerful for building data structures.

Dennis Ritchie

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.



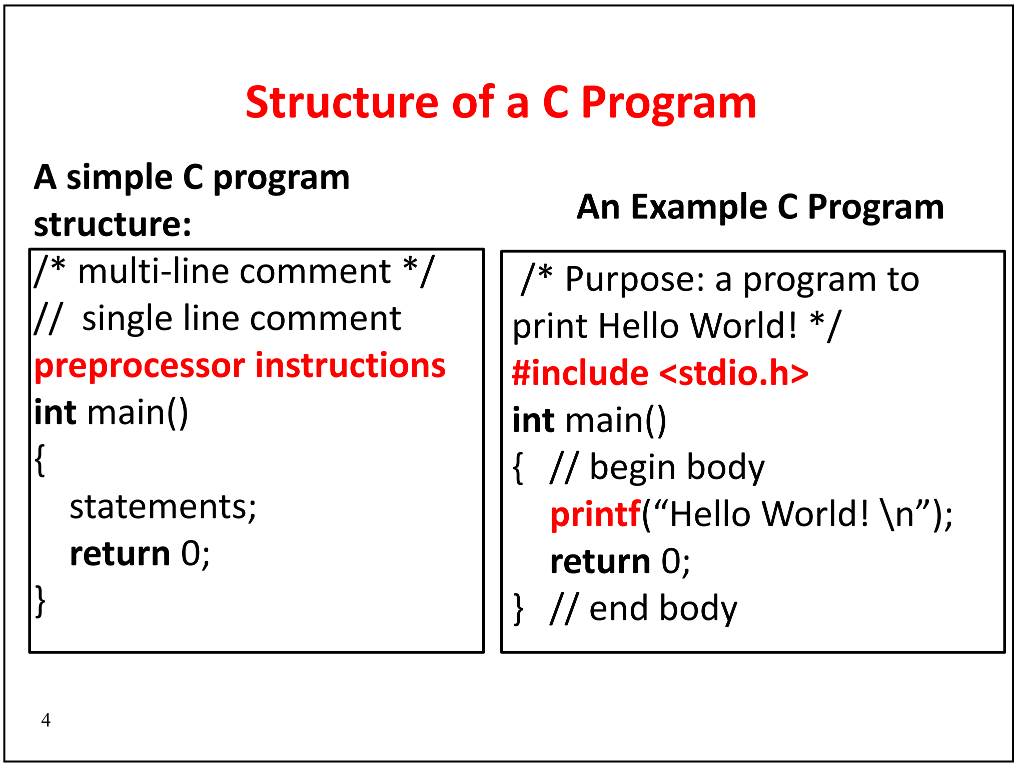
Basic C Programming

- 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;
}
```

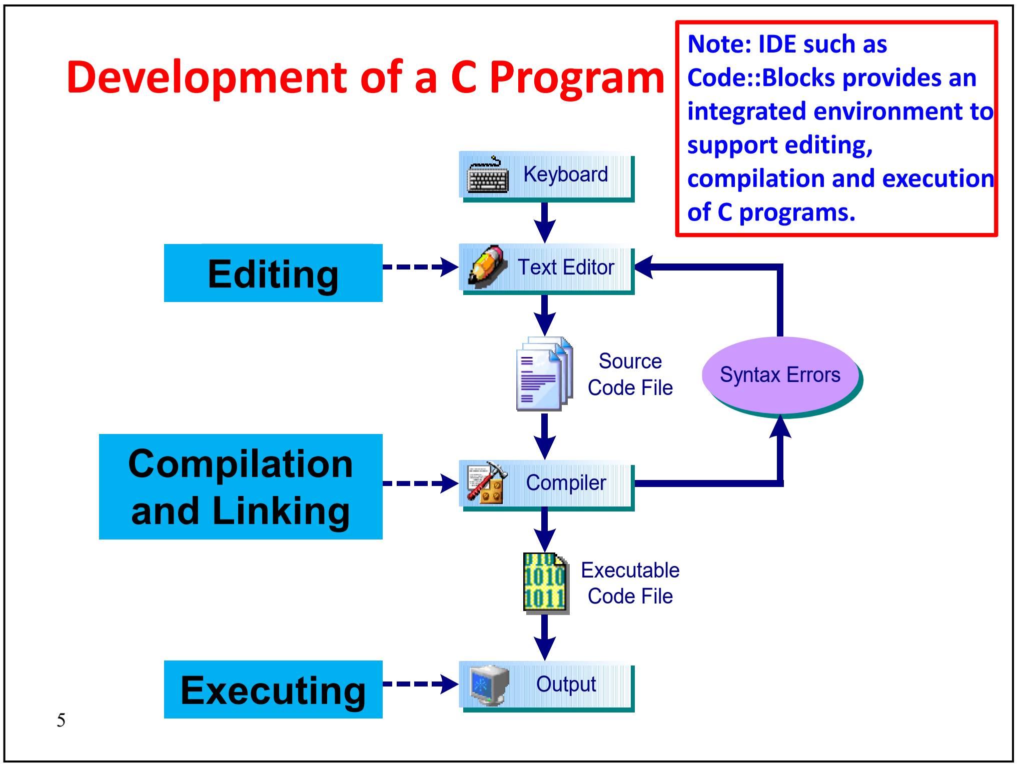
An Example C Program

```
 /* 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.

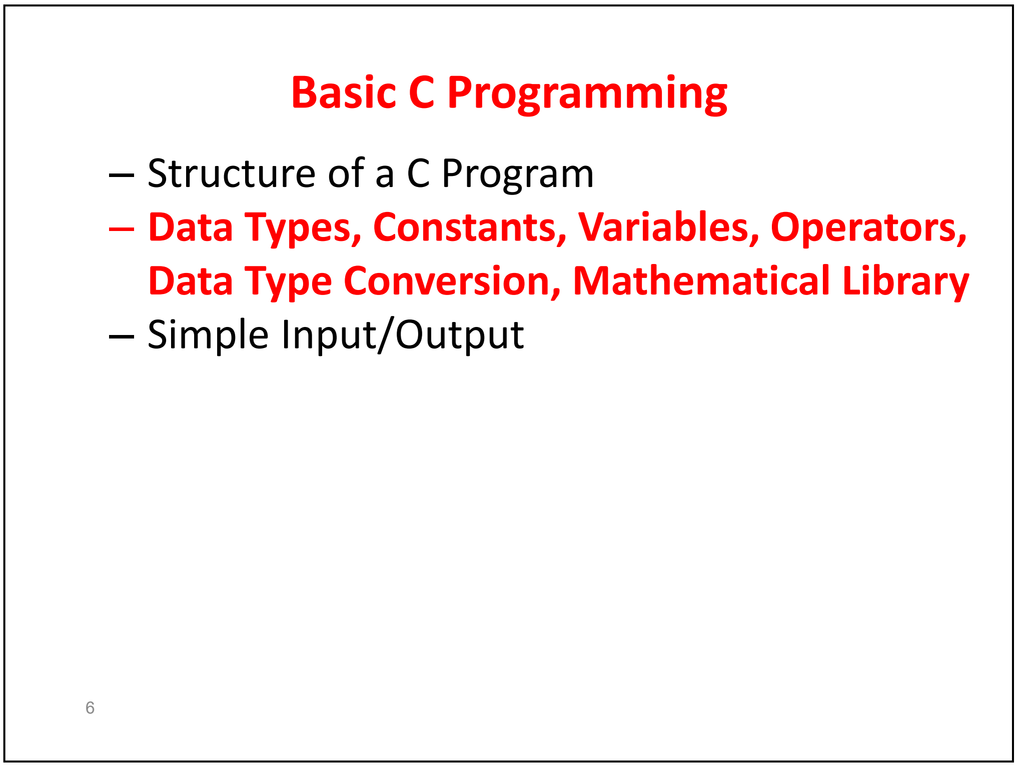


Development of a C Program

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

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

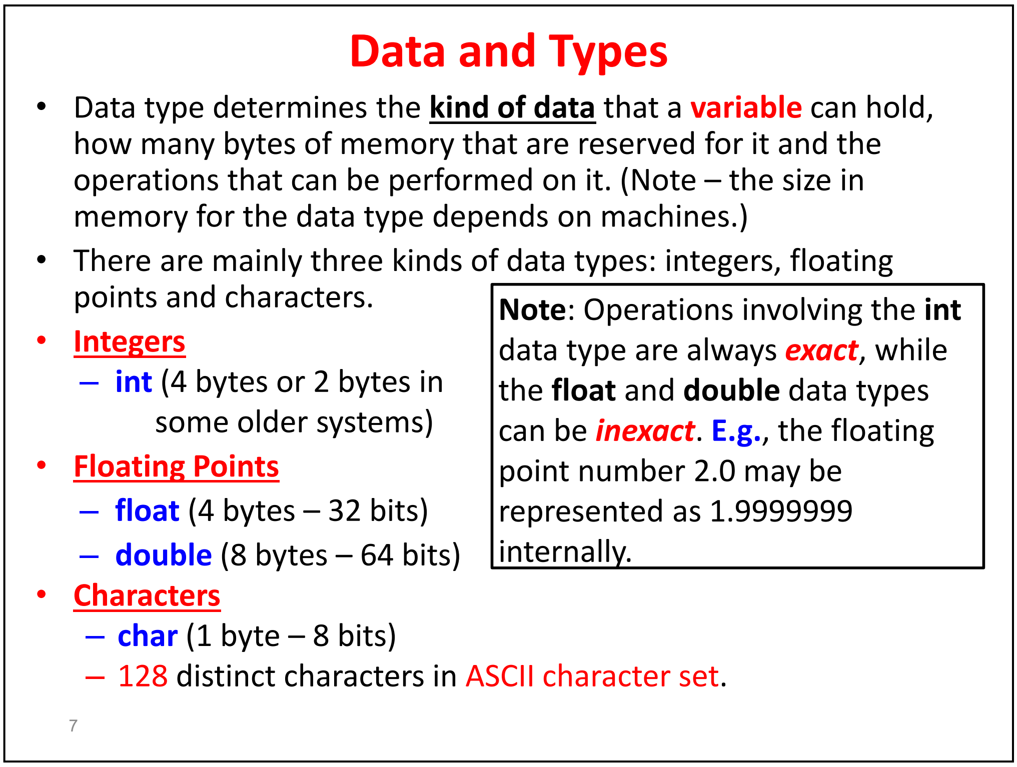
– Structure of a C Program

– **Data Types, Constants, Variables, Operators, Data Type Conversion, Mathematical Library**

– Simple Input/Output

Basic C Programming

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



Data and Types

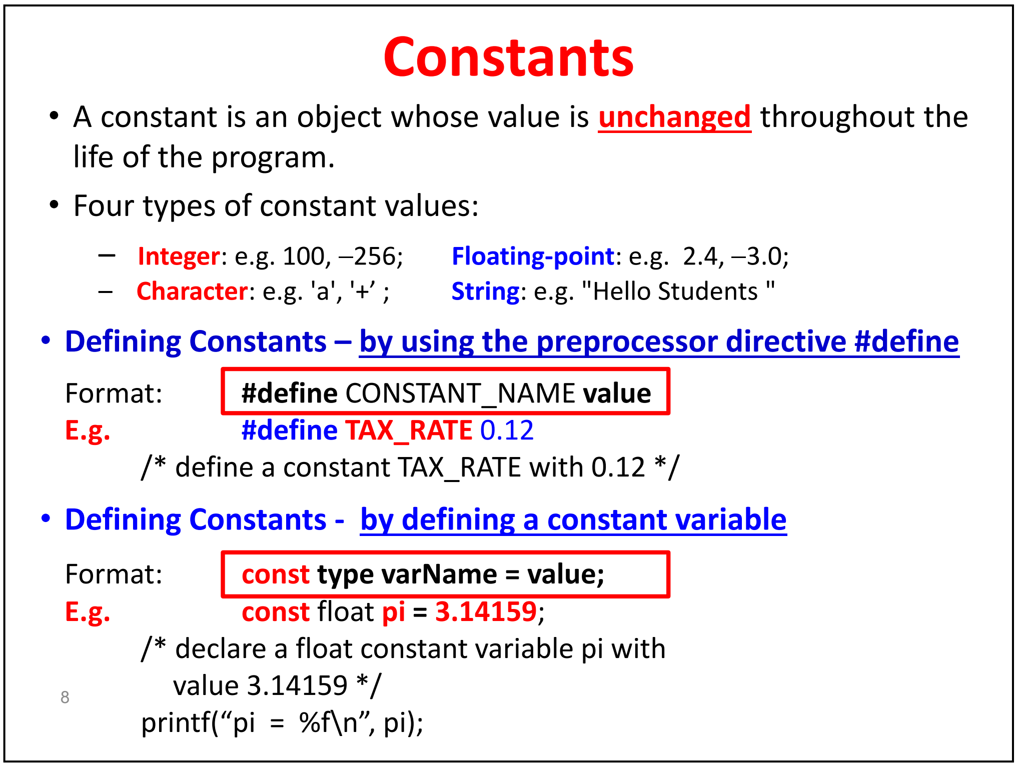
- Data type determines the **kind of data** that a **variable** can hold, how many bytes of memory that are reserved for it and the operations that can be performed on it. (Note – the size in memory for the data type depends on machines.)
- There are mainly three kinds of data types: integers, floating 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**

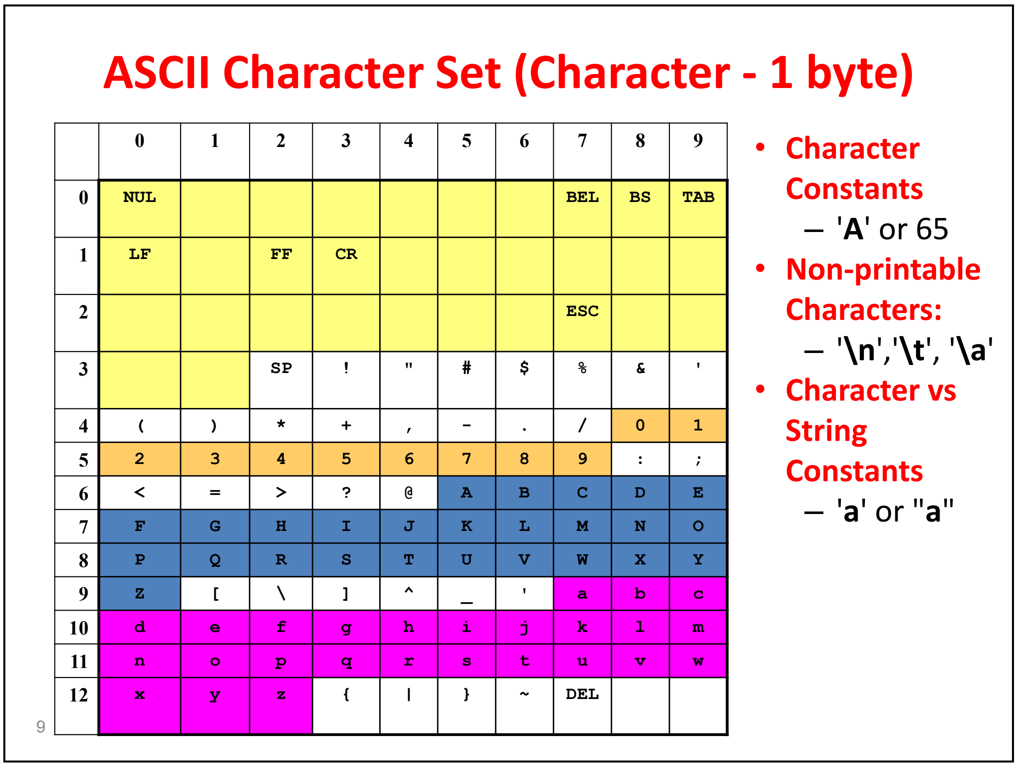
Format: **const type varName = value;**

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

```
/* declare a float constant variable pi with
   value 3.14159 */
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**;



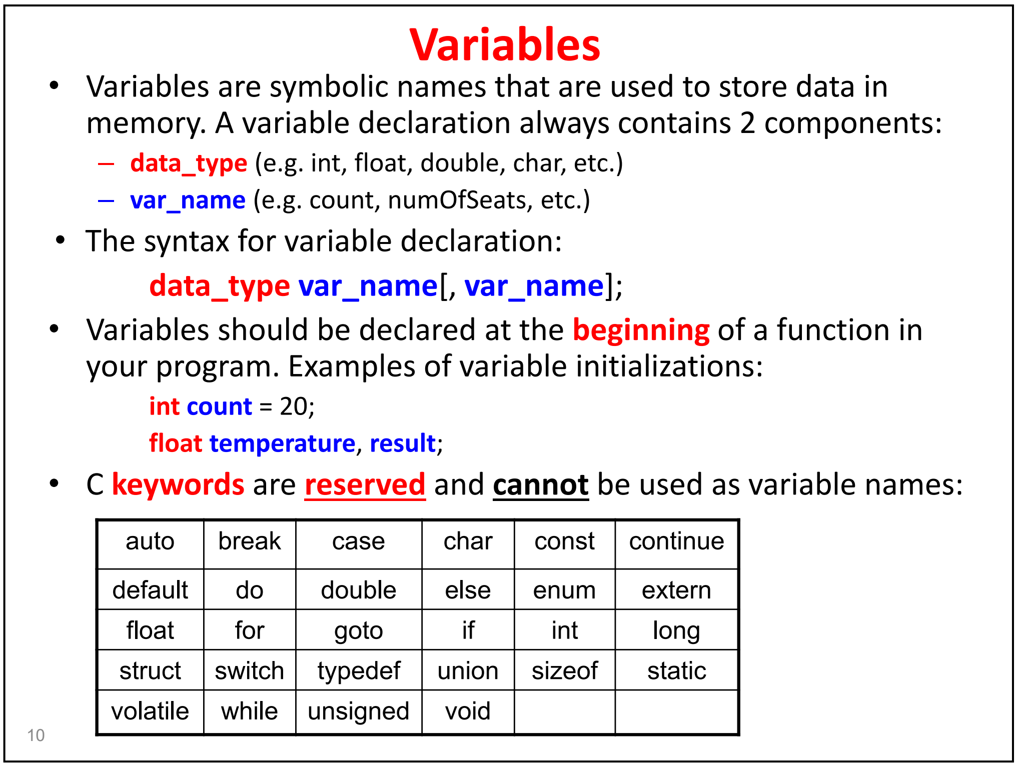
ASCII Character Set (Character - 1 byte)

	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		

- **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:

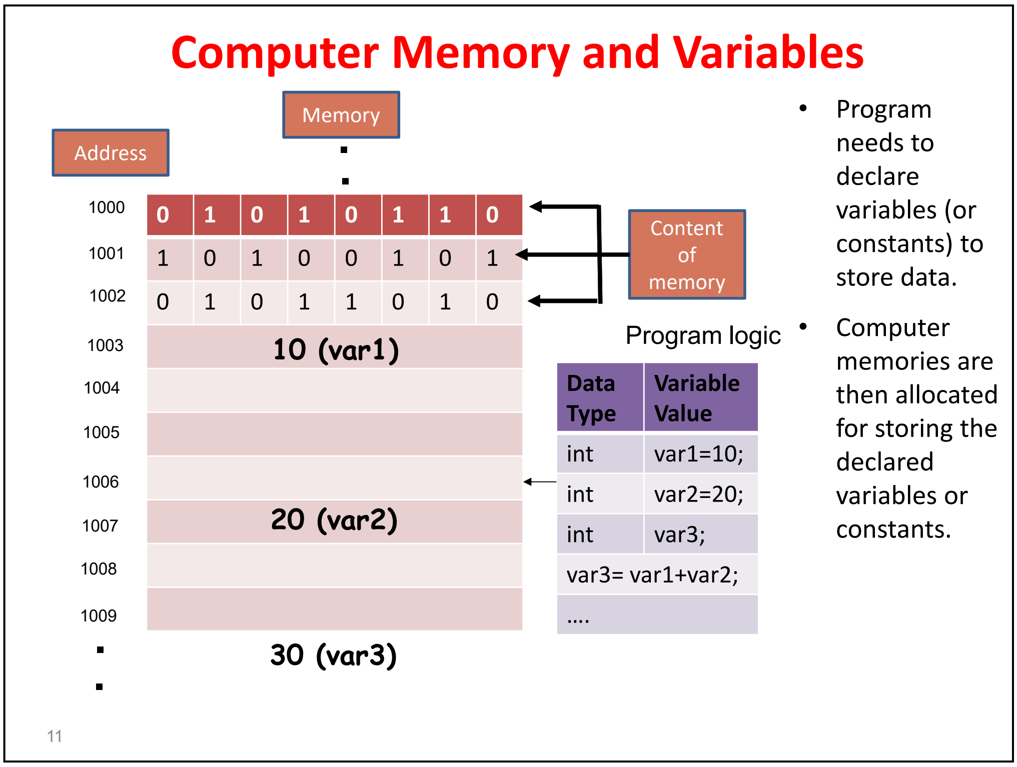
  int count = 20;

  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		

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.



Computer Memory and Variables

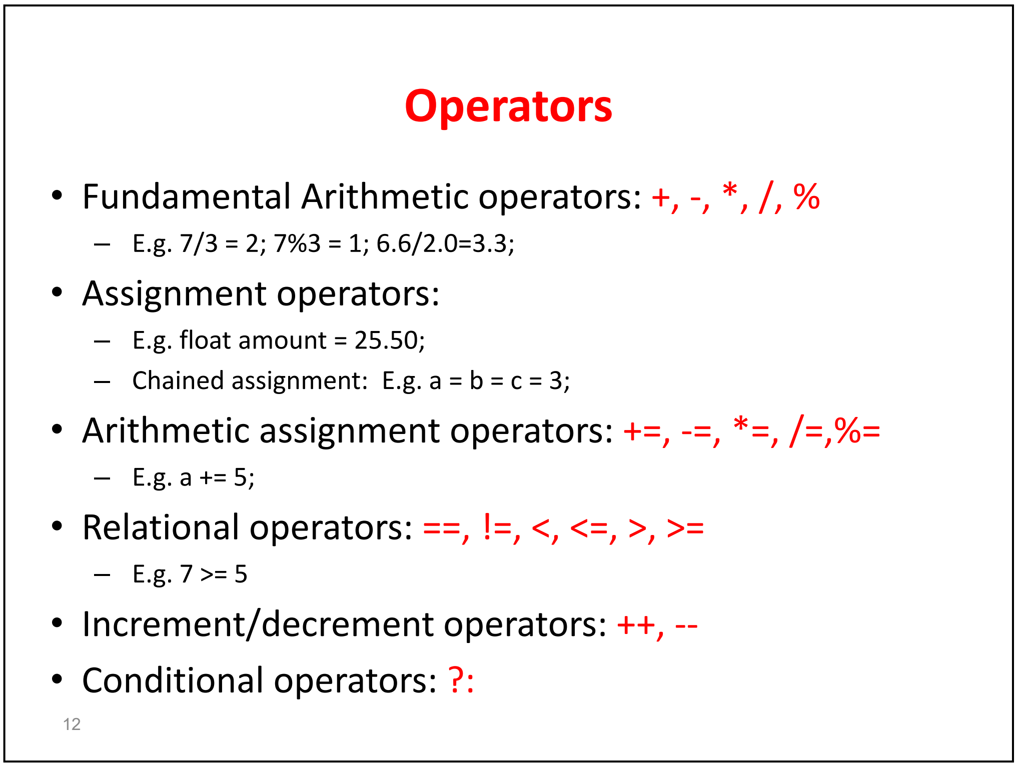
Address	Memory
1000	0 1 0 1 0 1 1 0
1001	1 0 1 0 0 1 0 1
1002	0 1 0 1 1 0 1 0
1003	10 (var1)
1004	
1005	
1006	
1007	20 (var2)
1008	
1009	
	30 (var3)

Data Type	Variable Value
int	var1=10;
int	var2=20;
int	var3;
var3= var1+var2;	
….	

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

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.



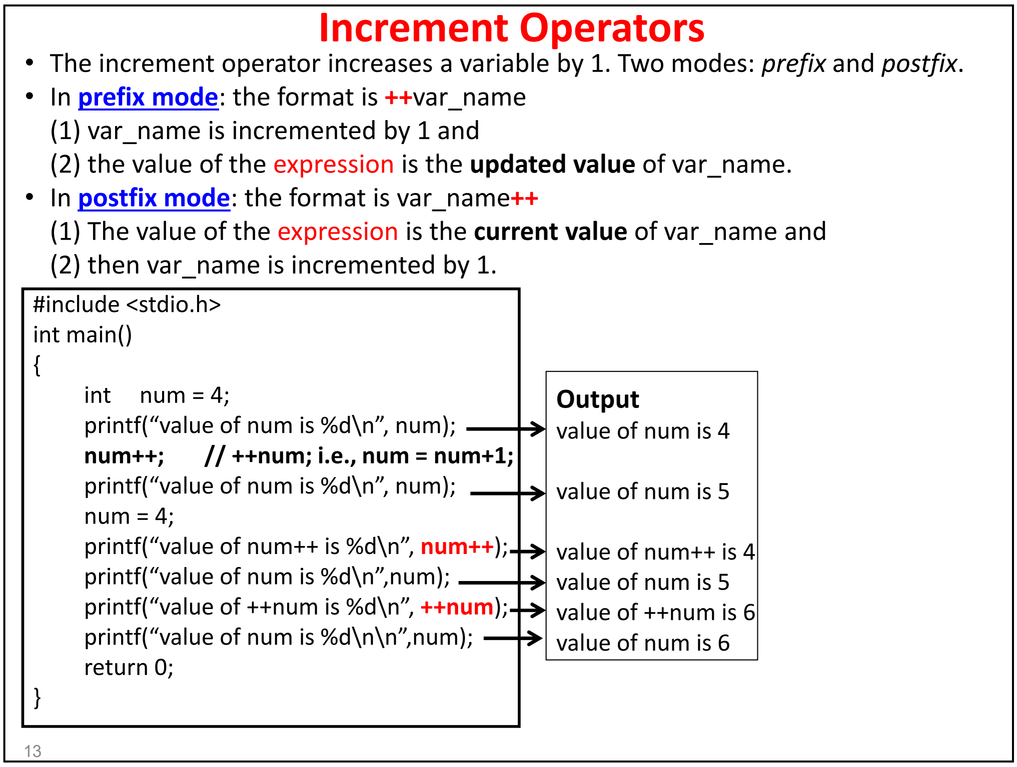
Operators

- 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.

```
#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;
}
```

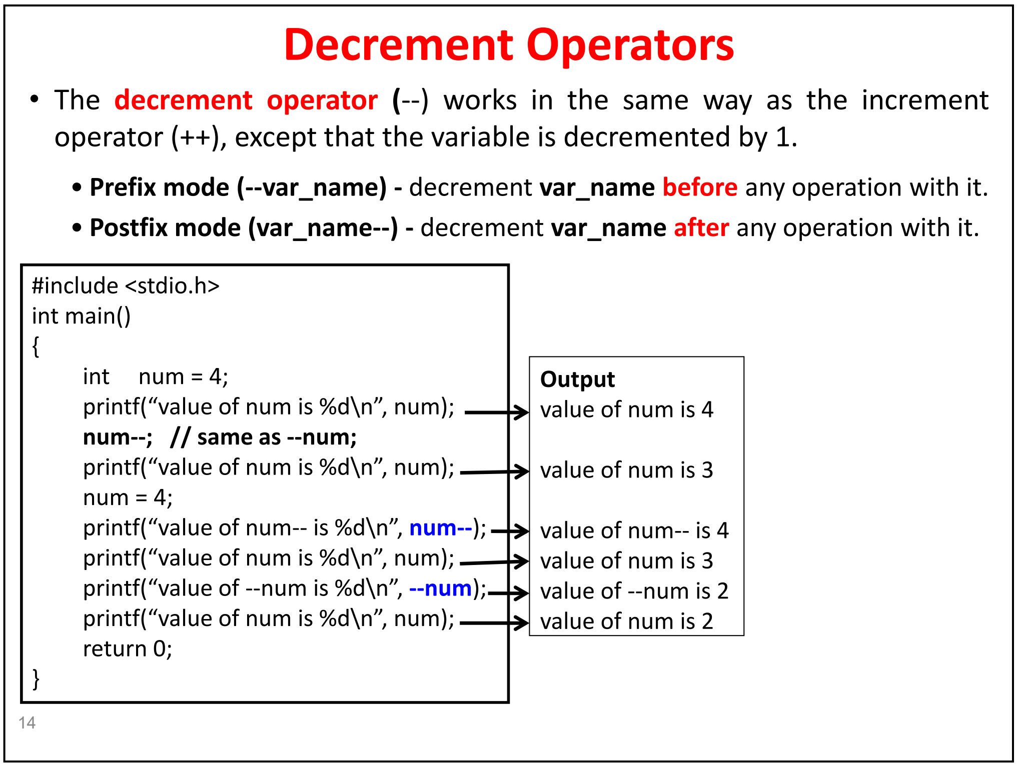
Output

```
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
```

13

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.

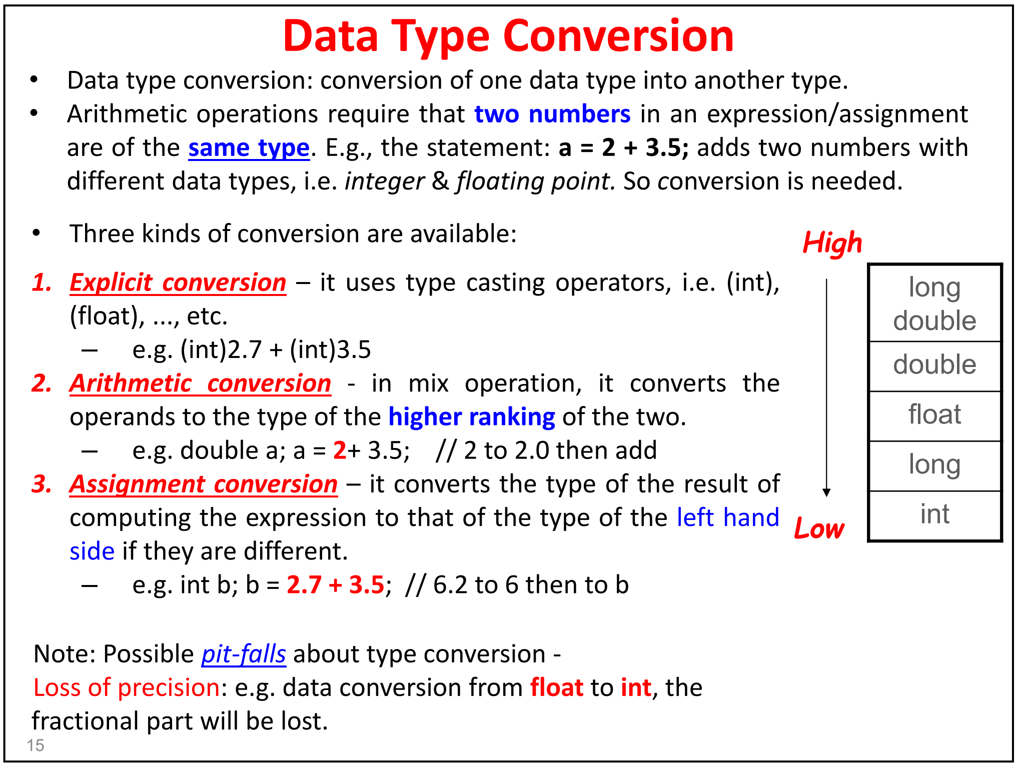
```
#include <stdio.h>
int main()
{
        int    num = 4;
        printf(“value of num is %d\n”, num);
        num--;   // same as --num;
        printf(“value of num is %d\n”, num);
        num = 4;
        printf(“value of num-- is %d\n”, num--);
        printf(“value of num is %d\n”, num);
    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

<u>Decrement Operators</u>

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 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

High

long double

double

float

long

int

Low

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

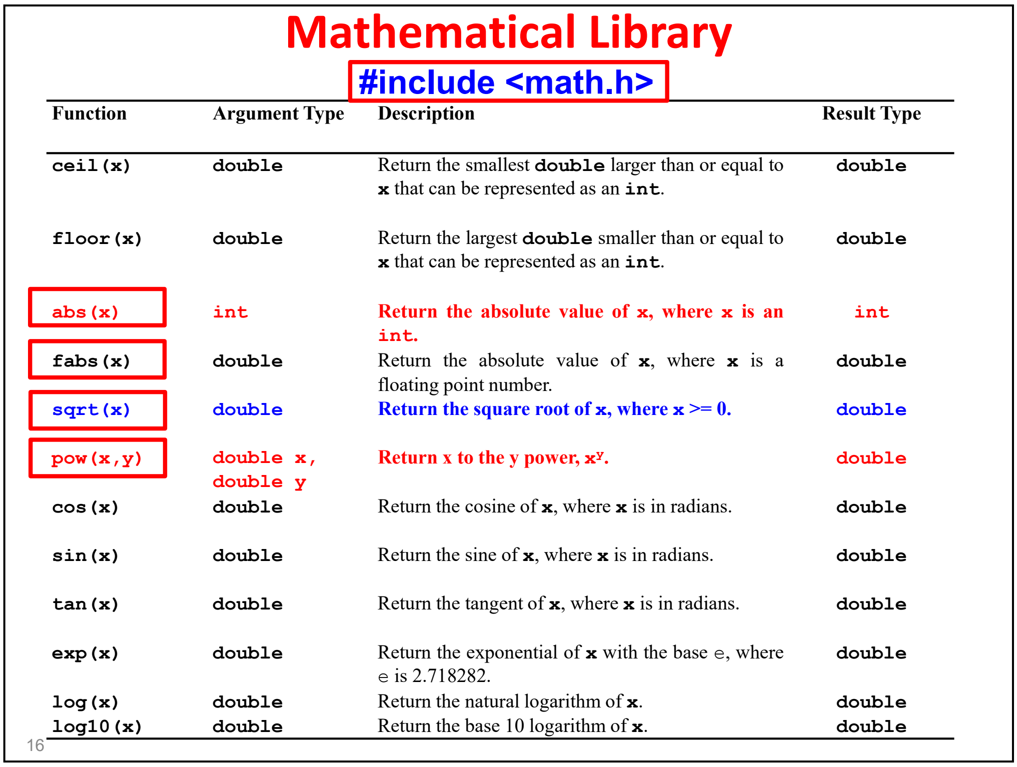
15

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.



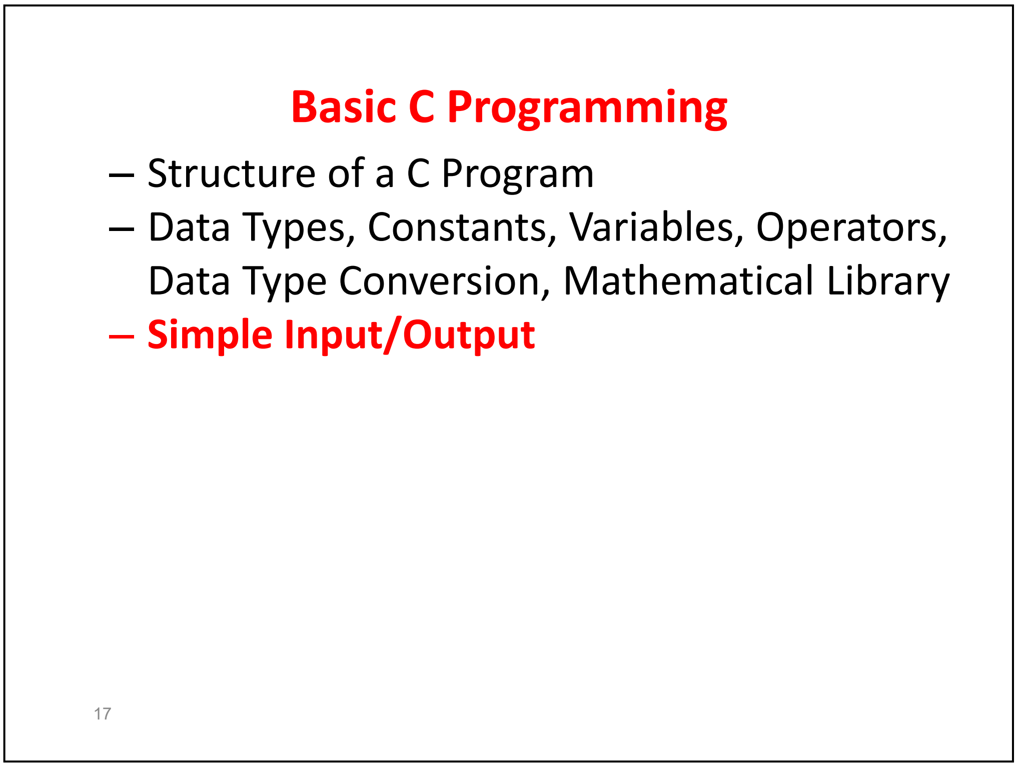
Mathematical Library

#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

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.



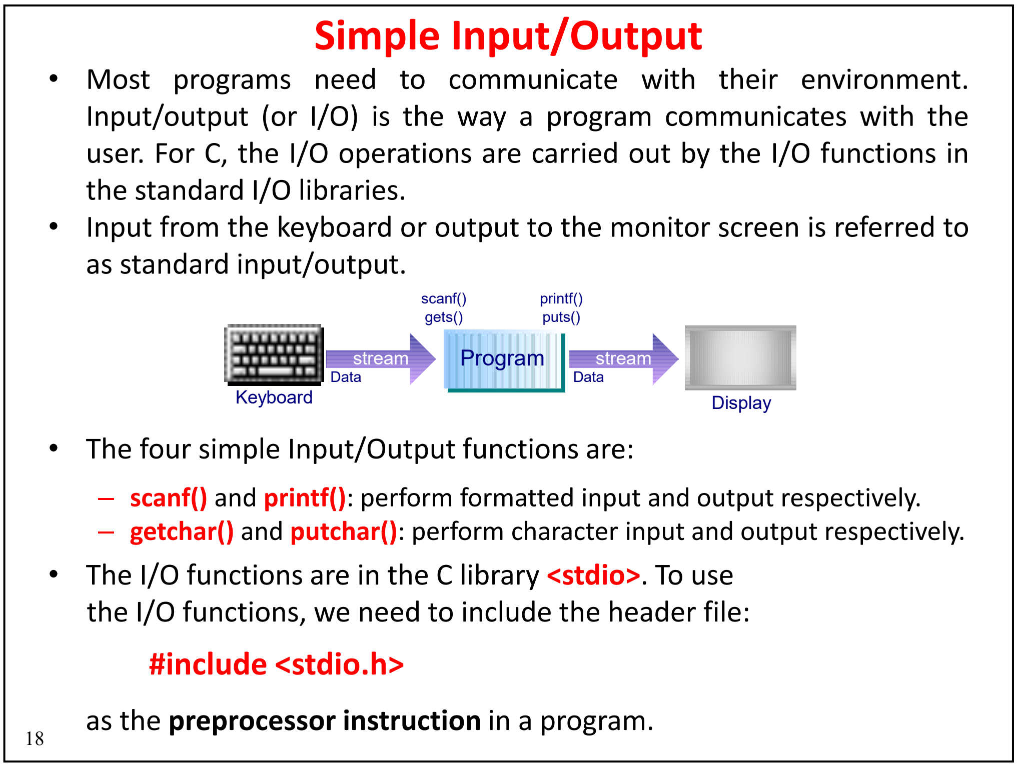
Basic C Programming

- 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.

scanf()
gets()
printf()
puts()
Keyboard
stream
Data
Program
stream
Data
Display

- 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:

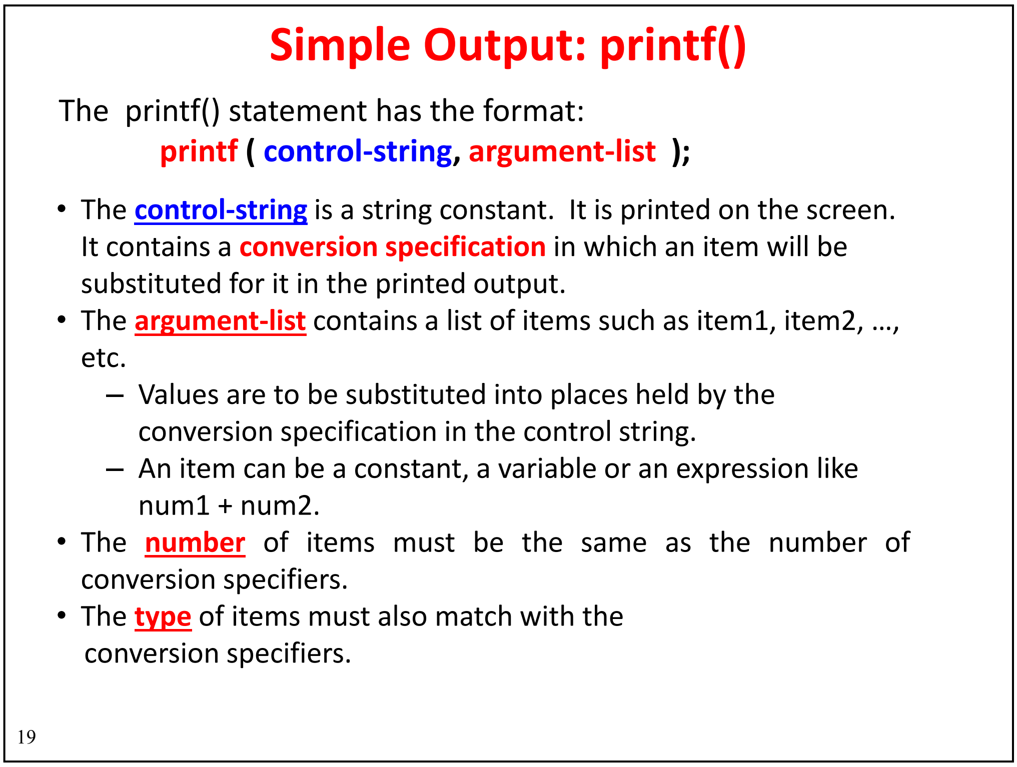
  #include <stdio.h>

  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 **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.

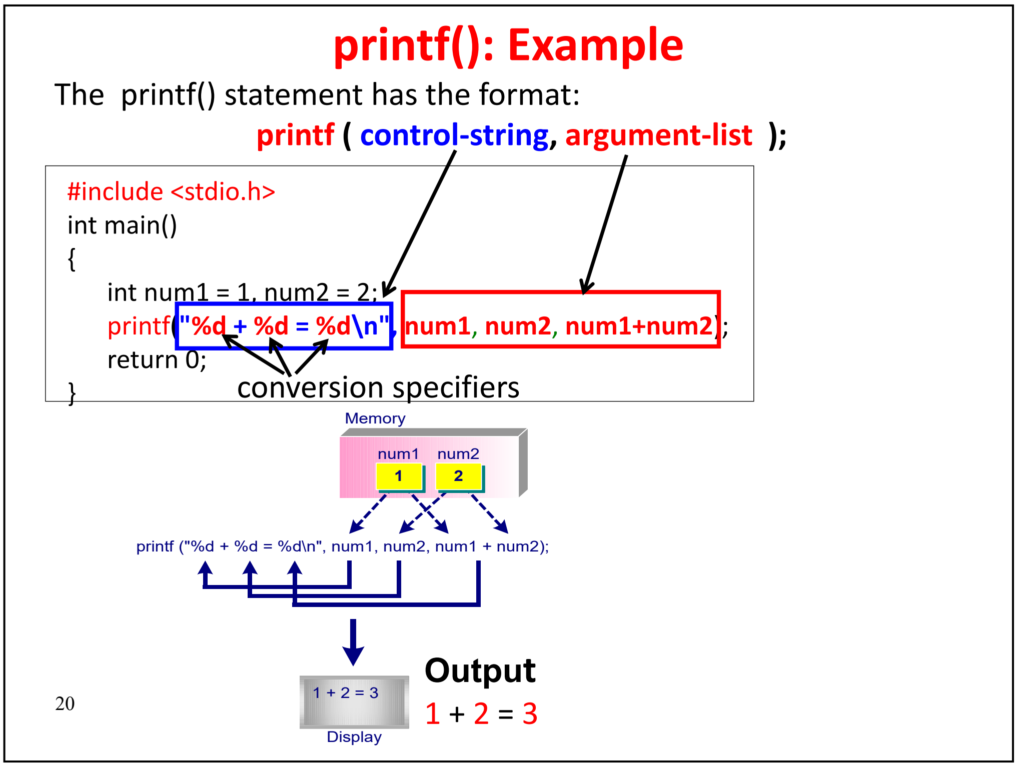
19

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

Output

1 + 2 = 3

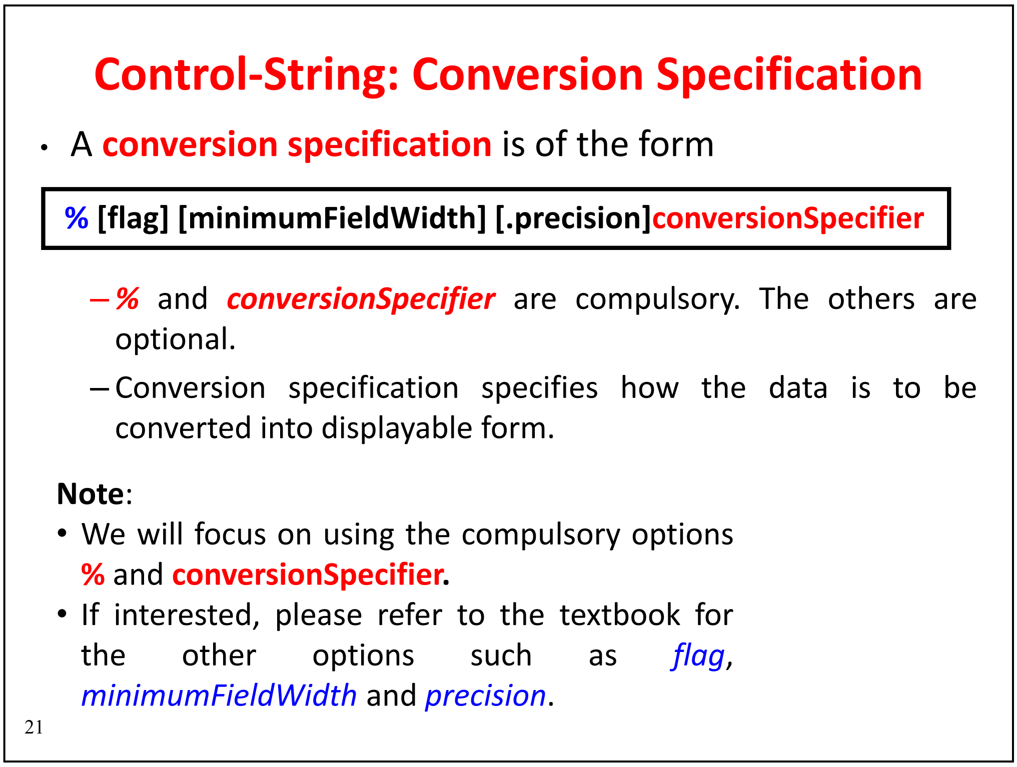
20

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.

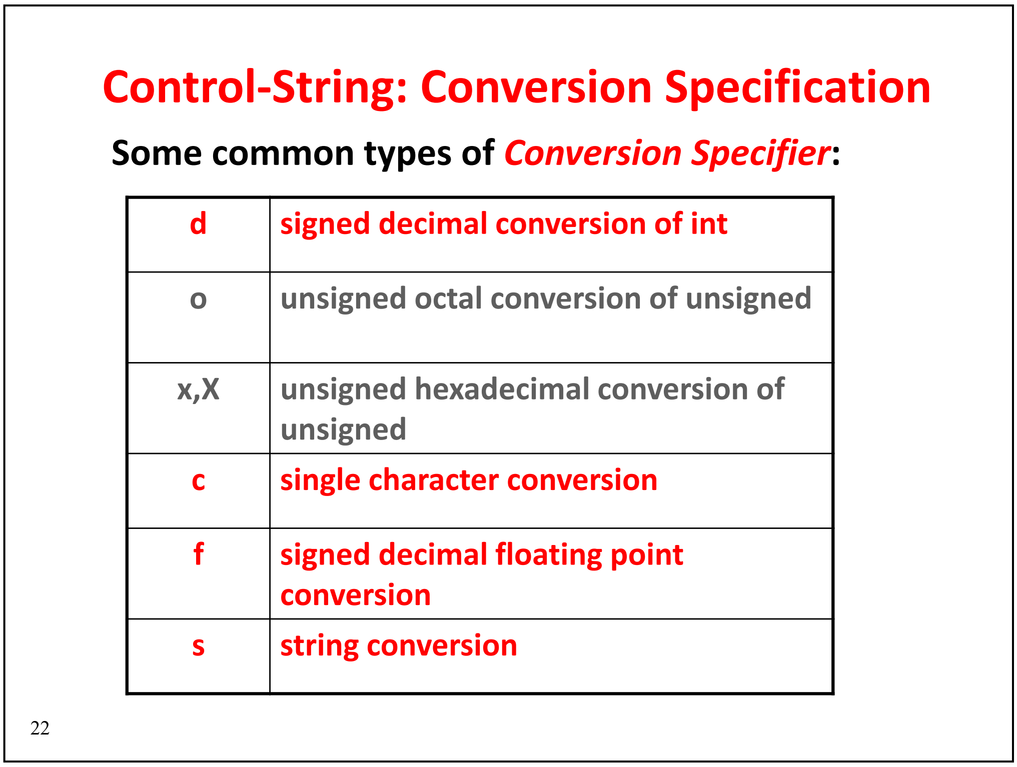
Note:

- 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

<u>**Conversion Specification**</u>

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

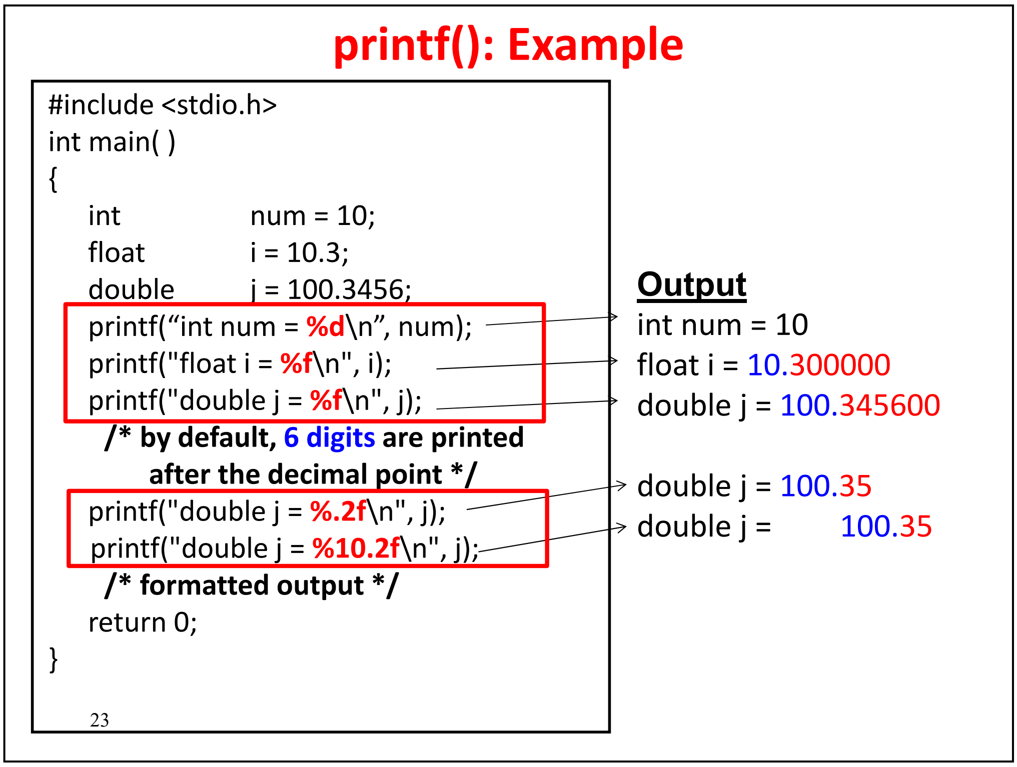
Some common types of *Conversion Specifier*:

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

```
#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;
}
```

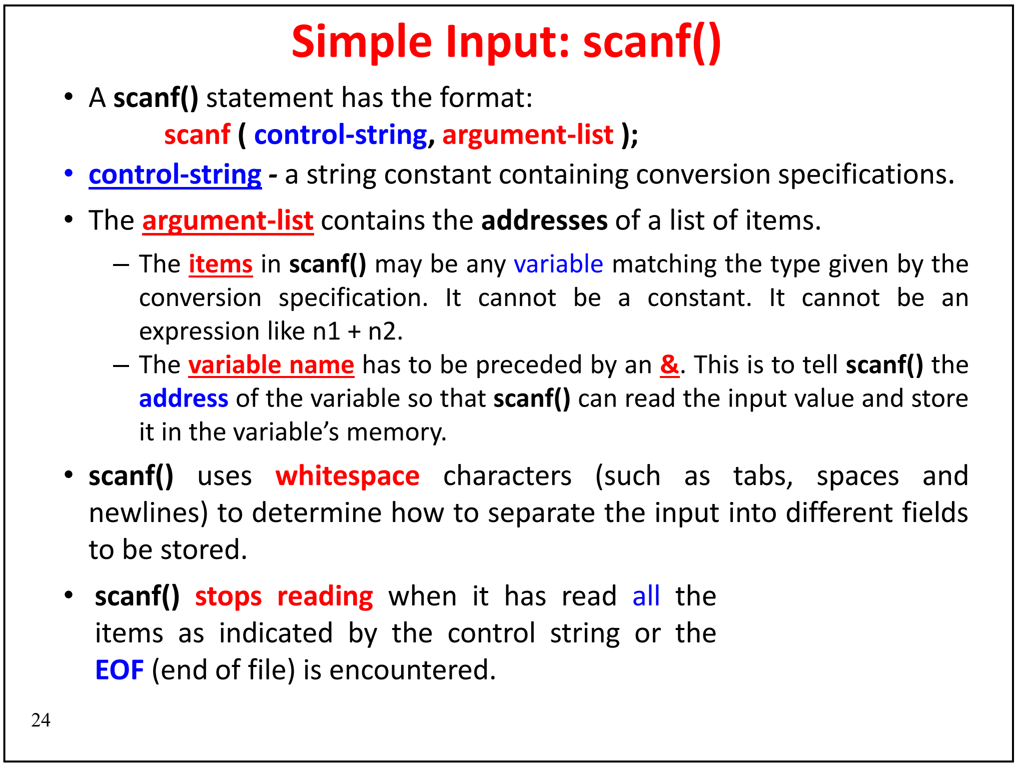
Output

```
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()

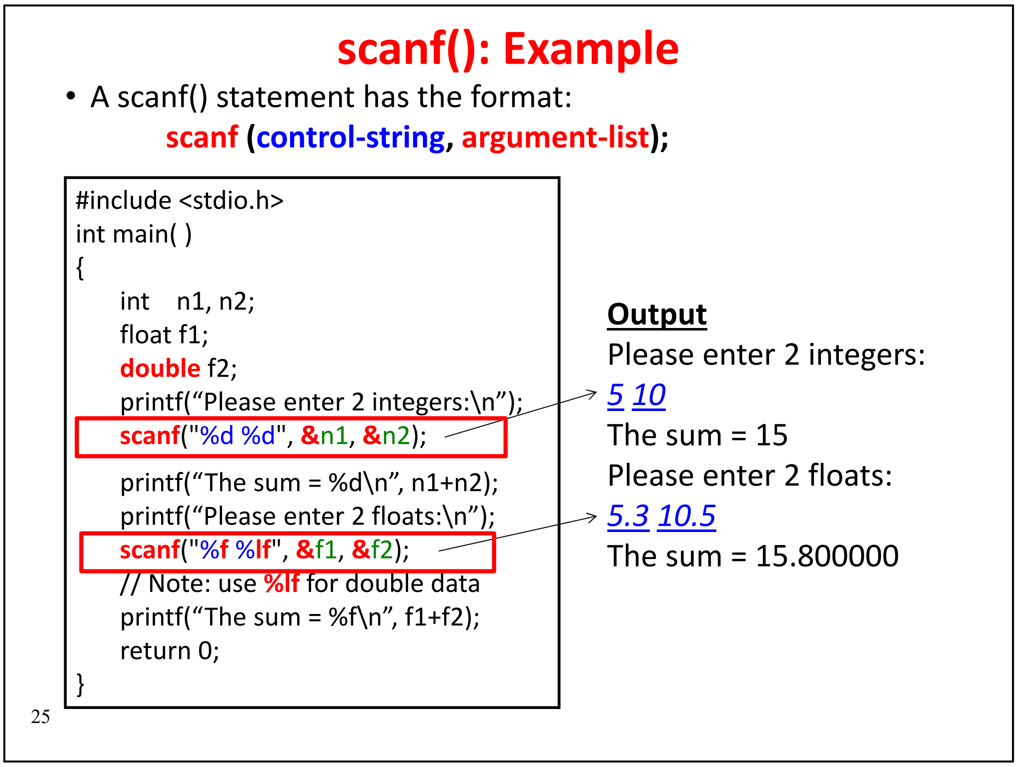
- A **scanf()** statement has the format:

  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);

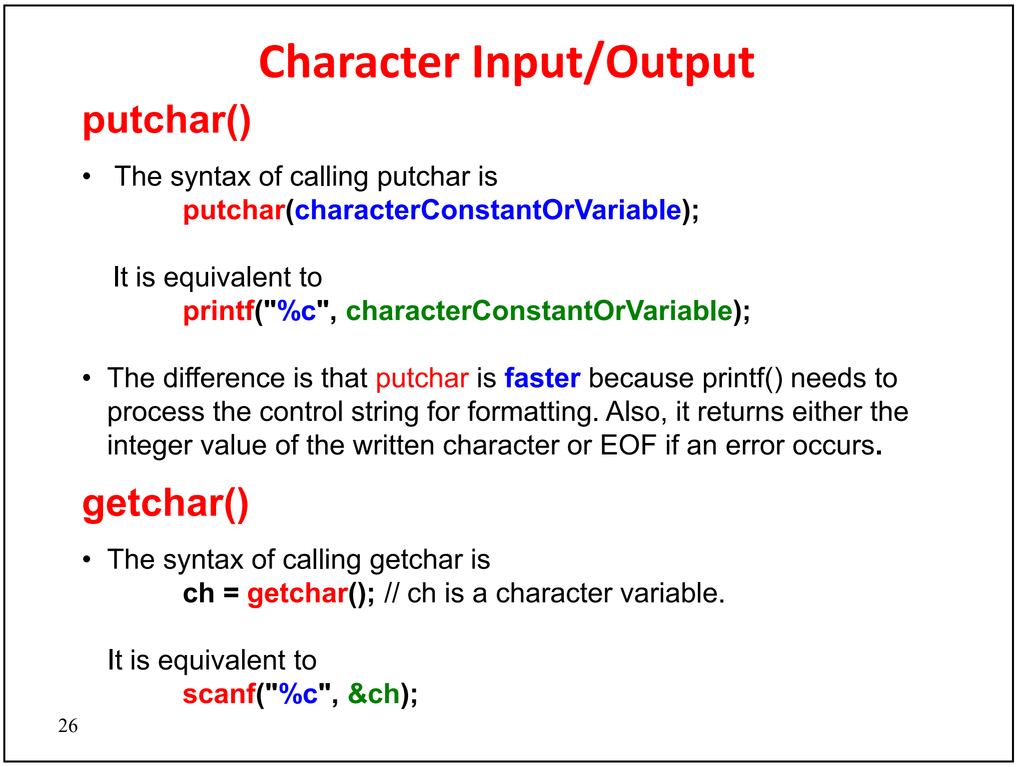
```
#include <stdio.h>
int main( )
{
      int    n1, n2;
      float f1;
      double f2;
      printf(“Please enter 2 integers:\n”);
      scanf("%d %d", &n1, &n2);
      printf(“The sum = %d\n”, n1+n2);
      printf(“Please enter 2 floats:\n”);
      scanf("%f %lf", &f1, &f2);
      // Note: use %lf for double data
      printf(“The sum = %f\n”, f1+f2);
      return 0;
}
```

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);

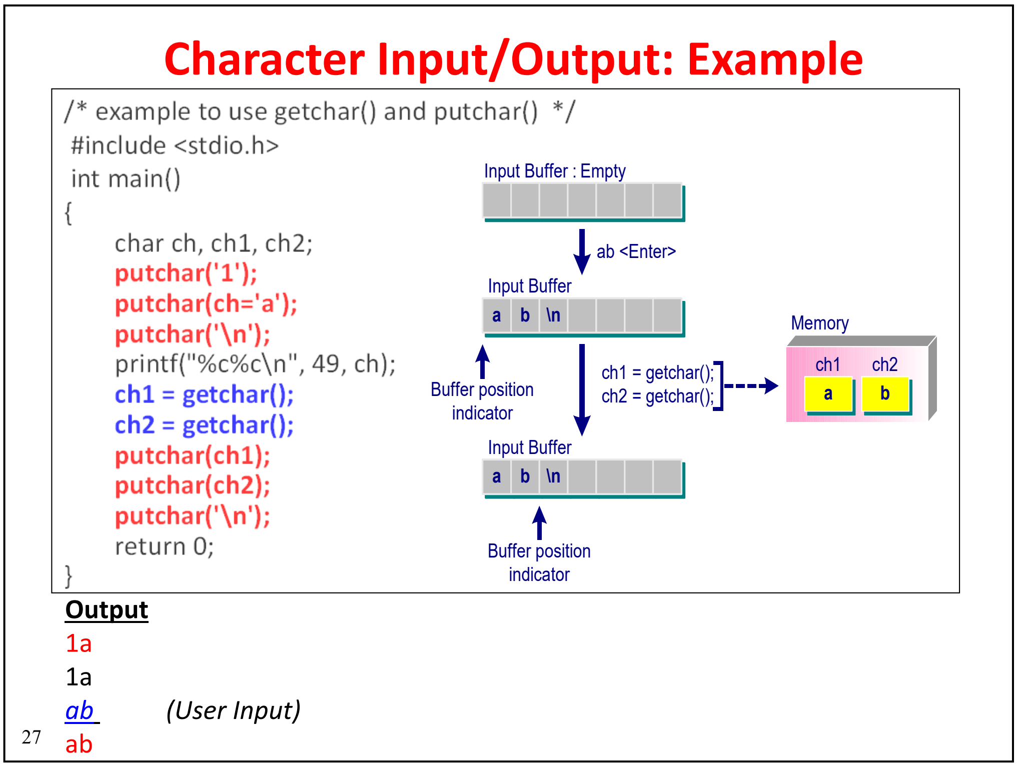
Character Input/Output

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

```
/* 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
```

27

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.

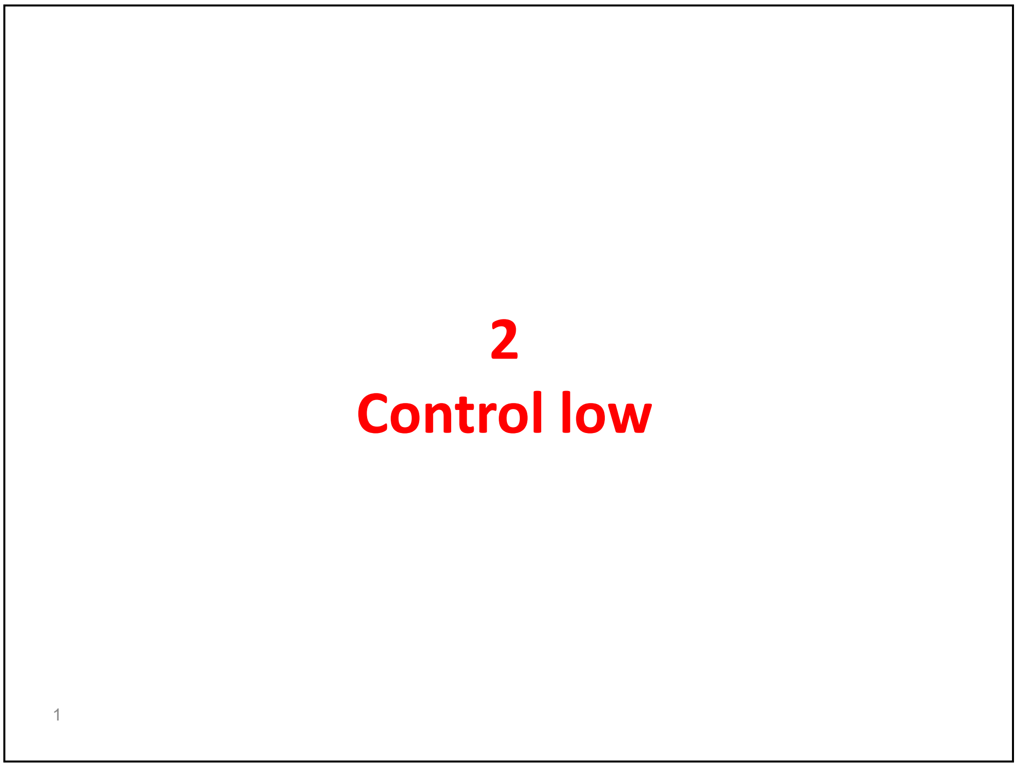
However, the newline character (**\n**) still remains in the input buffer that needs to be taken care of before processing another input request. One way to deal with the extra newline character is to flush the input buffer by using the statement **scanf('\n');** to read the newline character from the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

Thank You!

28

Thank You

1. Thanks for watching the lecture video.

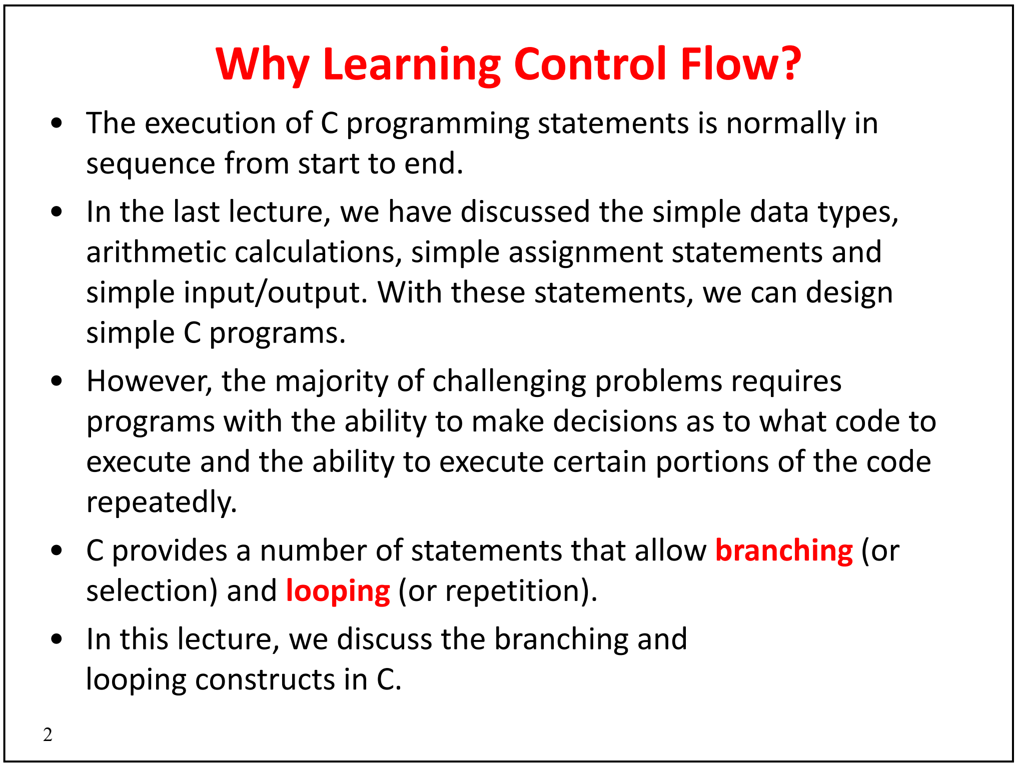


2
Control low

1

Control Flow

1. In this lecture, we discuss the Control Flow in the C programming language.



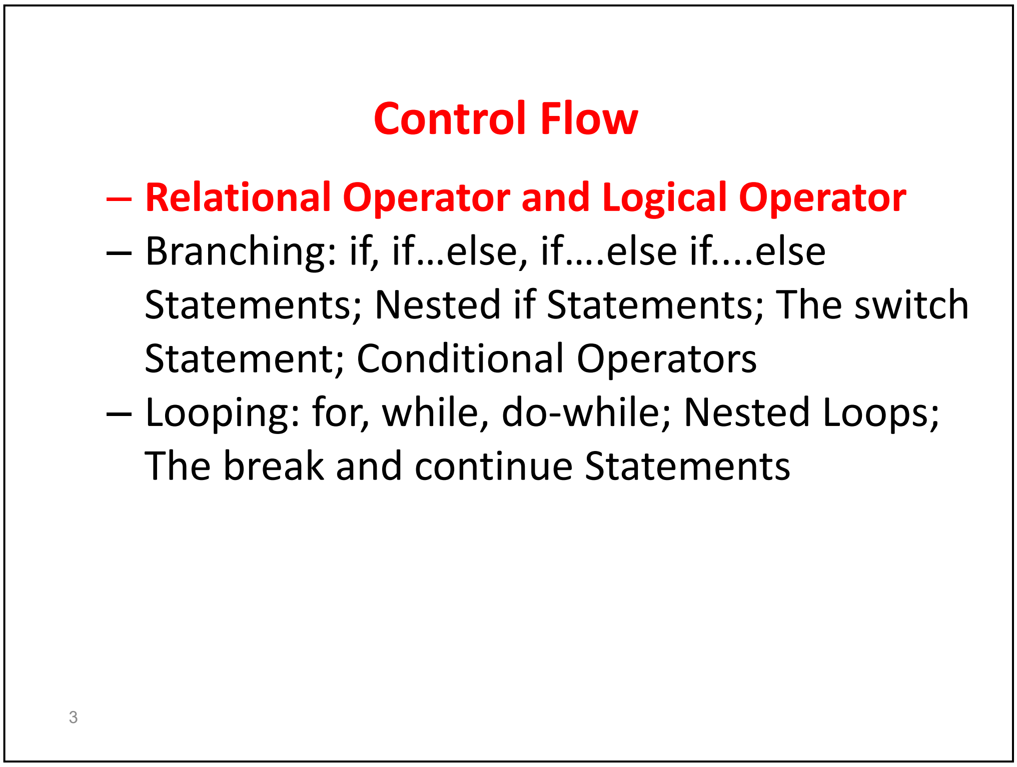
Why Learning Control Flow?

- The execution of C programming statements is normally in sequence from start to end.
- In the last lecture, we have discussed the simple data types, arithmetic calculations, simple assignment statements and simple input/output. With these statements, we can design simple C programs.
- However, the majority of challenging problems requires programs with the ability to make decisions as to what code to execute and the ability to execute certain portions of the code repeatedly.
- C provides a number of statements that allow **branching** (or selection) and **looping** (or repetition).
- In this lecture, we discuss the branching and looping constructs in C.

2

Why Learning Control Flow?

1. The execution of C programming statements is normally in sequence from start to end.
2. In the last lecture, we have discussed the simple data types, arithmetic calculations, simple assignment statements and simple input/output. With these statements, we can design simple C programs.
3. However, the majority of challenging problems requires programs with the ability to make decisions as to what code to execute and the ability to execute certain portions of the code repeatedly.
4. C provides a number of statements that allow branching (or selection) and looping (or repetition).
5. In this lecture, we discuss the branching and looping constructs in C.



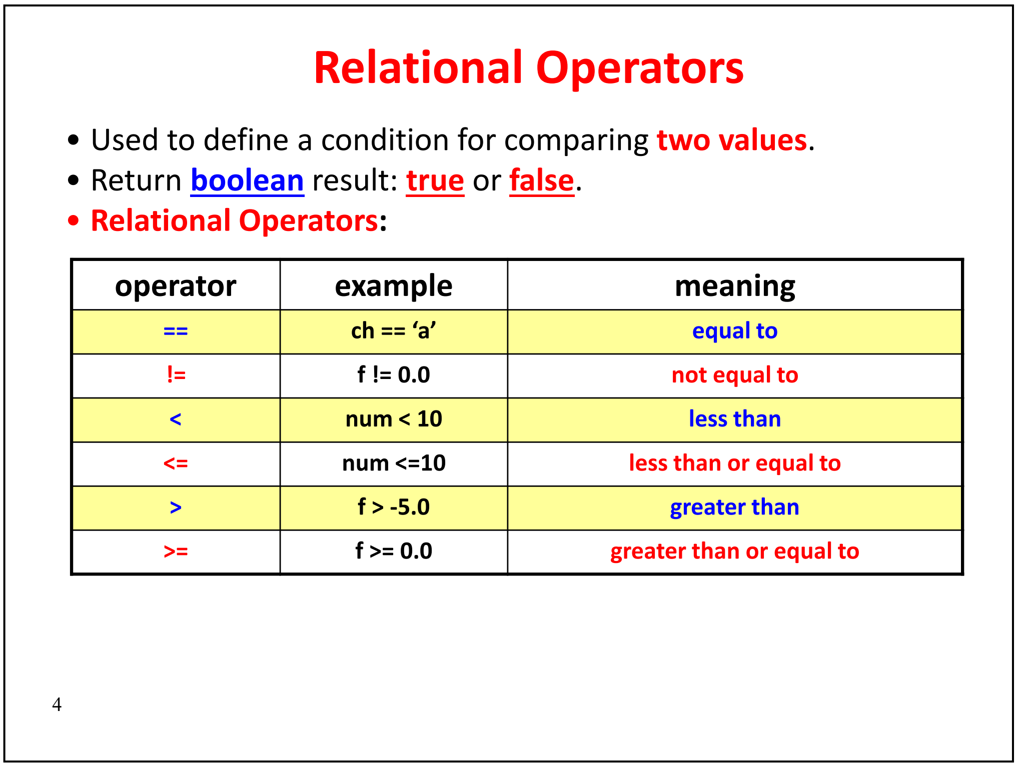
Control Flow

- **Relational Operator and Logical Operator**
- Branching: if, if…else, if….else if....else Statements; Nested if Statements; The switch Statement; Conditional Operators
- Looping: for, while, do-while; Nested Loops; The break and continue Statements

3

Control Flow

1. We first discuss the relational operators and logical operators.
2. Then, we discuss the branching statements and looping statements.
3. Here, we start by discussing the relational operators and logical operators.



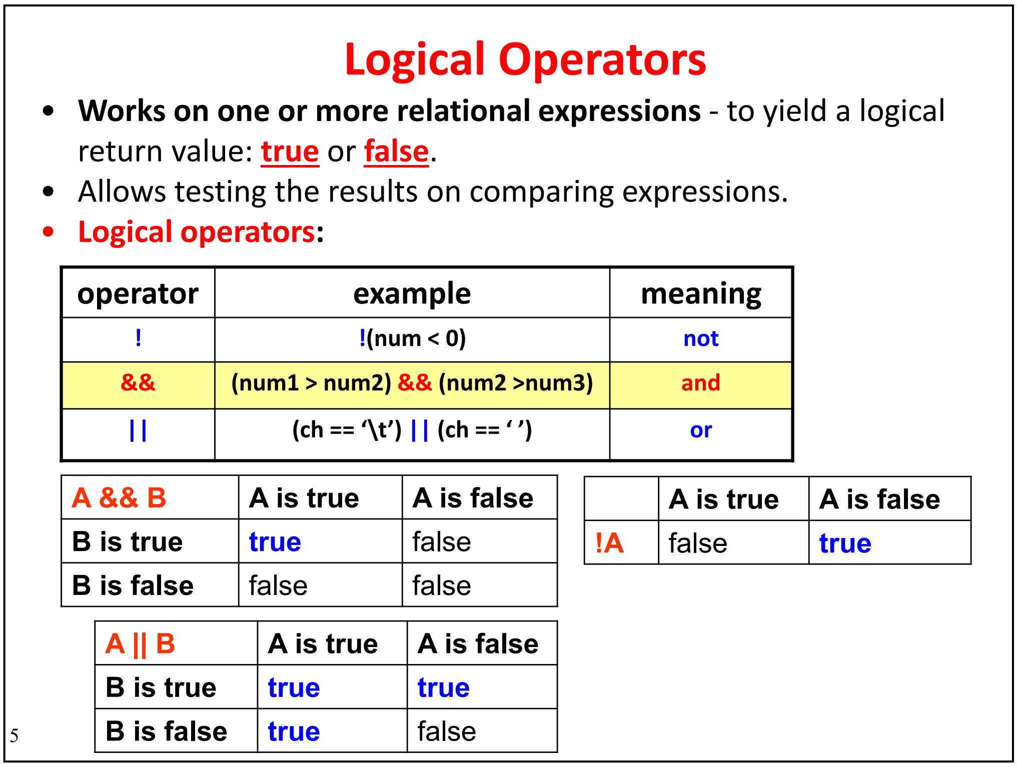
Relational Operators

- Used to define a condition for comparing **two values**.
- Return **boolean** result: **true** or **false**.
- **Relational Operators**:

operator	example	meaning
==	ch == ‘a’	equal to
!=	f != 0.0	not equal to
<	num < 10	less than
<=	num <=10	less than or equal to
>	f > -5.0	greater than
>=	f >= 0.0	greater than or equal to

Relational Operators

1. In a branching operation, the decision on which statements to be executed is based on a comparison between two values. To support this, C provides relational operators.
2. Relational expressions involving relational operators are an essential part of control structures for branching and looping.
3. Relational operators in C include equal to (==), not equal to (!=), less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=). These operators are binary. They perform computations on their operands and return the result as either true or false. If the result is true, an integer value of 1 will be returned, and if the result is false, then the integer value of 0 will be returned.



Logical Operators

- **Works on one or more relational expressions** - to yield a logical return value: true or false.
- Allows testing the results on comparing expressions.
- Logical operators:

operator	example	meaning
!	!(num < 0)	not
&&	(num1 > num2) && (num2 >num3)	and
||	(ch == '\t') || (ch == ' ')	or

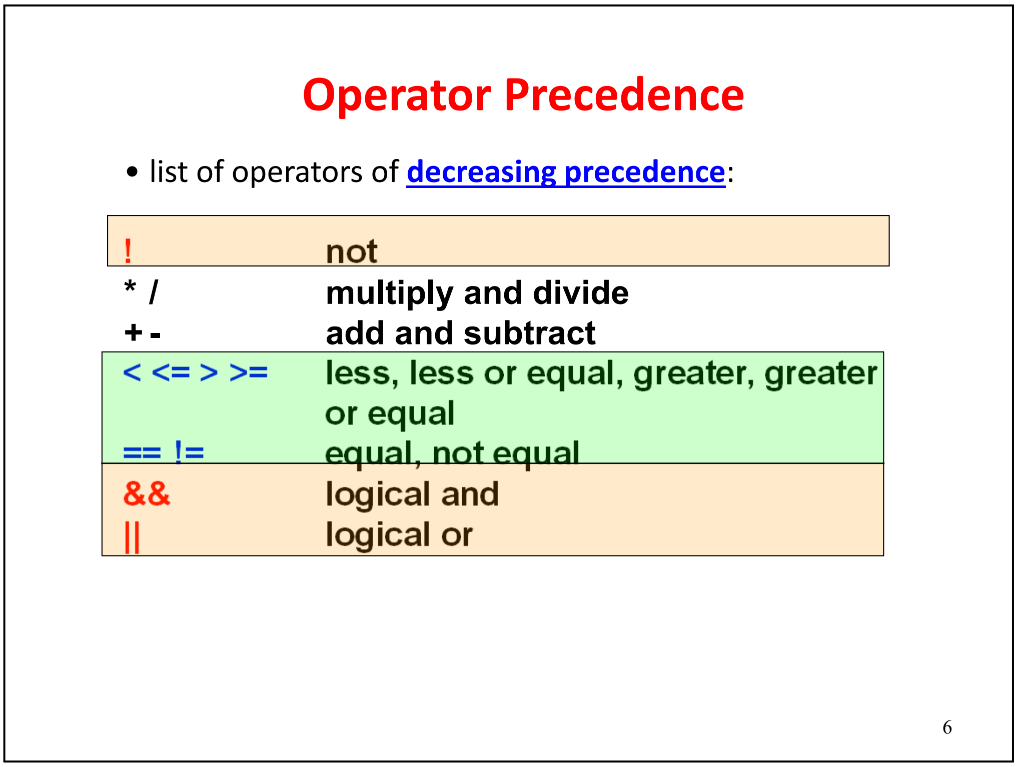
A && B	A is true	A is false
B is true	true	false
B is false	false	false

	A is true	A is false
!A	false	true

A || B	A is true	A is false
B is true	true	true
B is false	true	false

Logical Operators

1. Logical operators work on one or more relational expressions to yield either the logical value true or false. Both logical **and** (**&&**) and logical **or** (**||**) operators are binary operators, while the logical **not** operator (**!**) is a unary operator. Logical operators allow testing and combining of the results of comparison expressions.
2. Logical **not** operator (**!**) returns true when the operand is false and returns false when the operand is true. Logical **and** operator (**&&**) returns true when both operands are true, otherwise it returns false. Logical **or** operator (**||**) returns false when both operands are false, otherwise it returns true.



Operator Precedence

- list of operators of **decreasing precedence**:

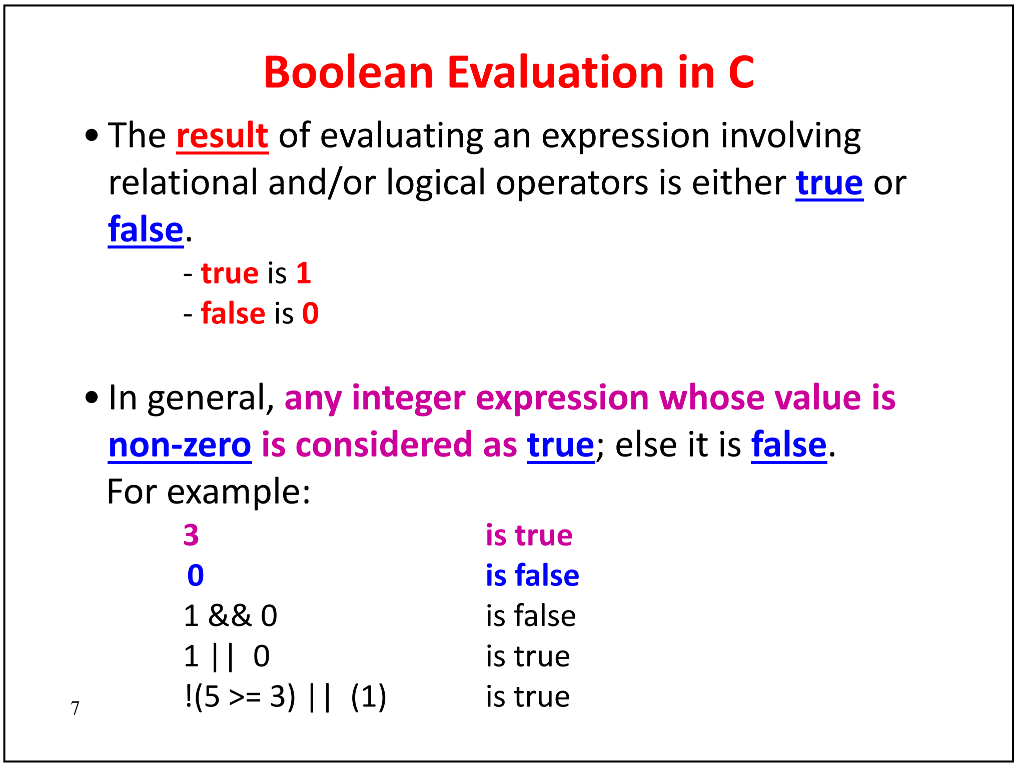
!	not
* /	multiply and divide
+ -	add and subtract
< <= > >=	less, less or equal, greater, greater or equal
== !=	equal, not equal
&&	logical and
||	logical or

Operator Precedence

1. Operator precedence determines the order of operator execution. The list of operators of **decreasing precedence** is shown in the slide:

!	not
* /	multiply and divide
+ -	add and subtract
< <= > >=	less, less or equal, greater, greater or equal
== !=	equal, not equal
&&	logical and
||	logical or

2. Note that the logical **not** (**!**) operator has the highest priority. It is followed by the multiplication, division, addition and subtraction operators. The logical **and** (**&&**) and **or** (**||**) operators have a lower priority than the relational operators.



Boolean Evaluation in C

- The **result** of evaluating an expression involving relational and/or logical operators is either **true** or **false**.
  - **true** is **1**
  - **false** is **0**

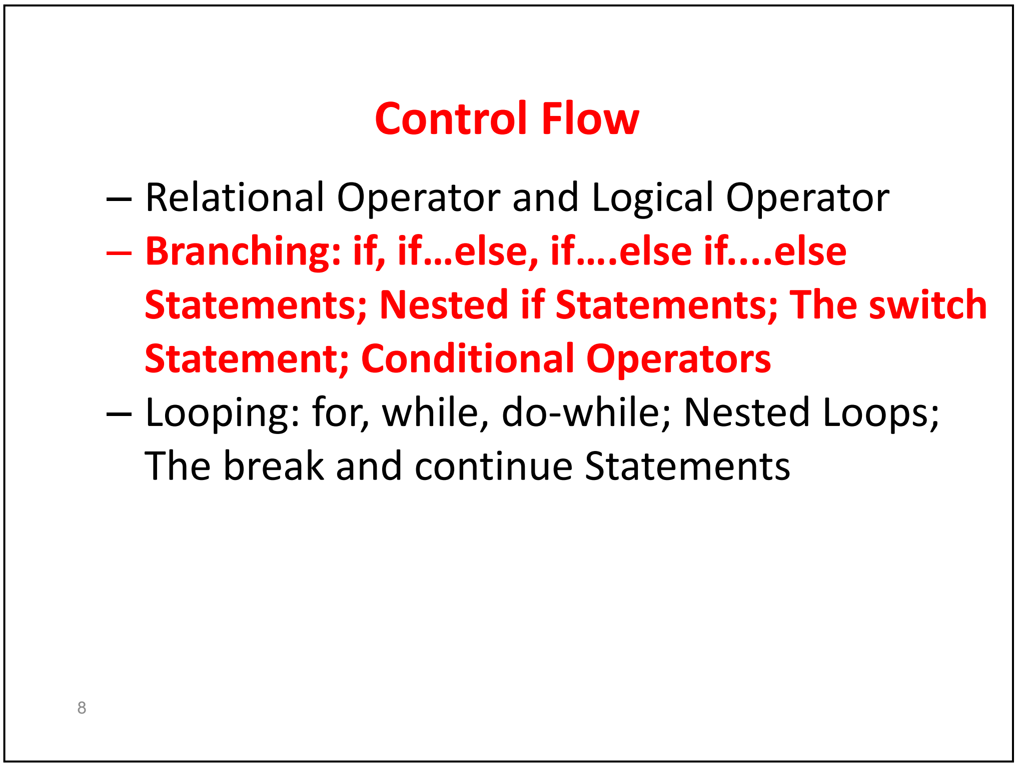
- In general, **any integer expression whose value is non-zero is considered as true**; else it is **false**.
  For example:

3	**is true**
0	**is false**
1 && 0	is false
1 || 0	is true
!(5 >= 3) || (1)	is true

7

Boolean Evaluation

1. The result of evaluating an expression involving relational and/or logical operators is either 1 or 0. When the result is true, it is 1. Otherwise it is 0.
2. Since C uses 0 to represent a false condition, any integer expression whose value is *non-zero* is considered as *true*; otherwise it is *false*.
3. Therefore, in the example, 3 is true, and 0 is false. (1 && 0) is false and (1 || 0) is true.

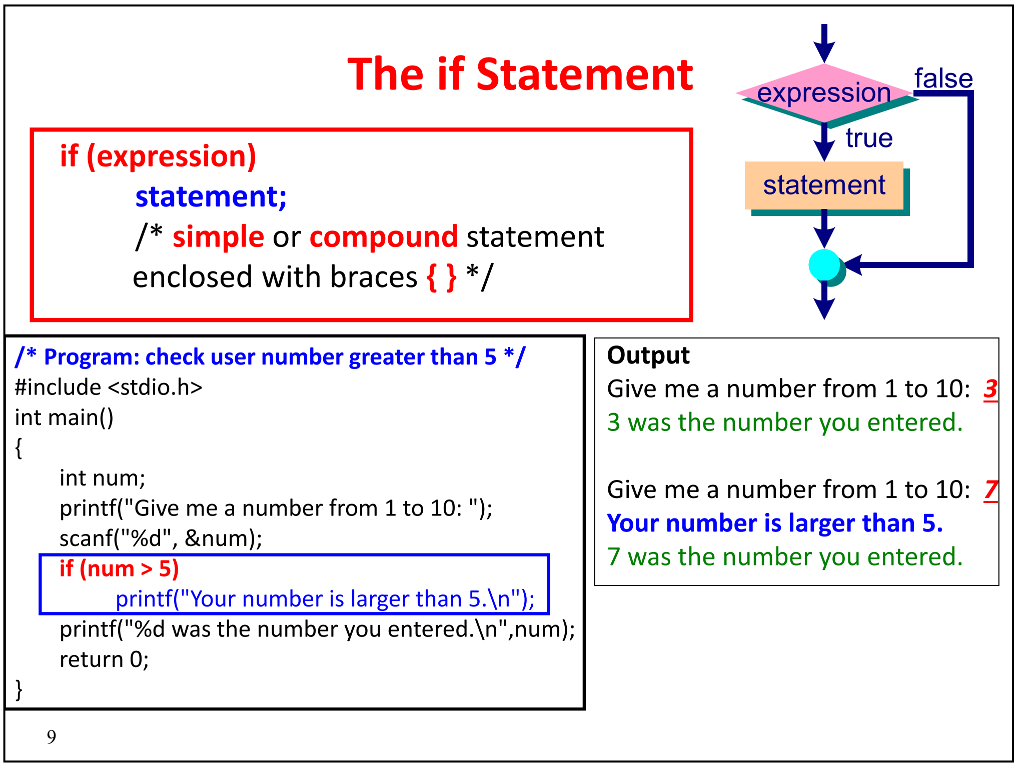


Control Flow

- Relational Operator and Logical Operator
- **Branching: if, if…else, if….else if….else Statements; Nested if Statements; The switch Statement; Conditional Operators**
- Looping: for, while, do-while; Nested Loops; The break and continue Statements

Control Flow

1. Here, we discuss the branching statements.



The if Statement

if (expression)
statement;
/* **simple** or **compound** statement
enclosed with braces **{ }** */

expression
false
true
statement

```
/* Program: check user number greater than 5 */
#include <stdio.h>
int main()
{
      int num;
      printf("Give me a number from 1 to 10: ");
      scanf("%d", &num);
      if (num > 5)
            printf("Your number is larger than 5.\n");
      printf("%d was the number you entered.\n",num);
      return 0;
}
```

Output

```
Give me a number from 1 to 10:  3
3 was the number you entered.

Give me a number from 1 to 10:  7
Your number is larger than 5.
7 was the number you entered.
```

The if Statement

1. The simplest form of the **if** statement is

   if (expression)

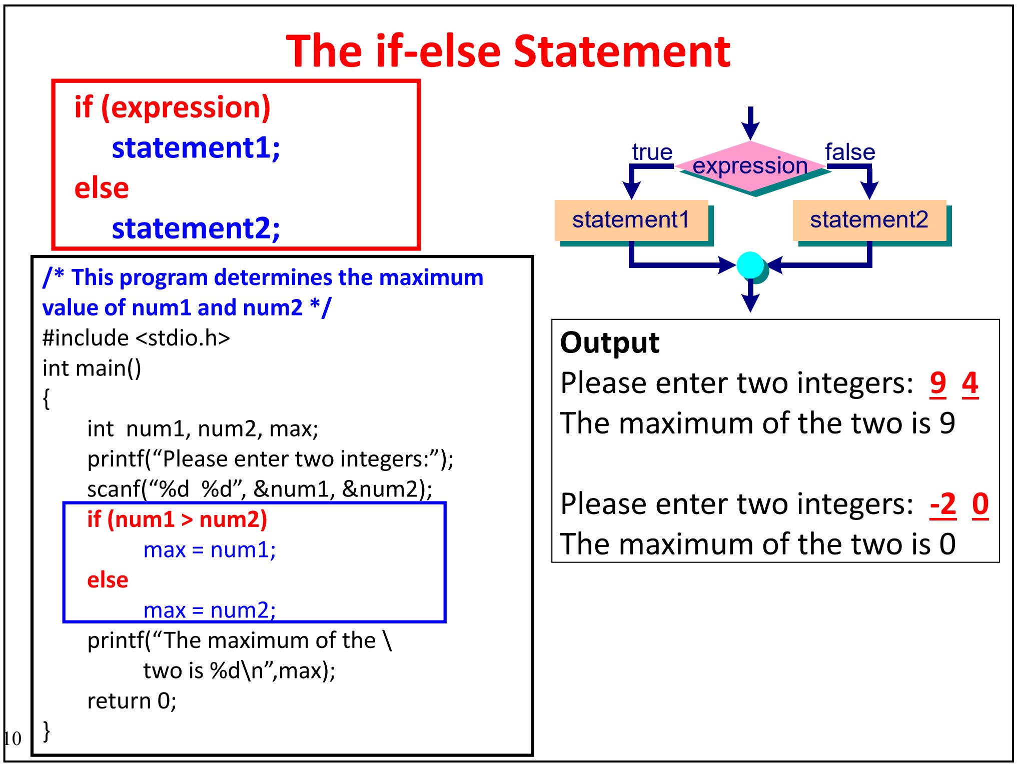
   statement;

   if is a reserved keyword.

2. If the **expression** is evaluated to be true (i.e. non-zero), then the **statement** is executed. If the **expression** is evaluated to be false (i.e. zero), then the **statement** is ignored, and the control is passed to the next program statement following the **if** statement.

3. The **statement** may be a single statement terminated by a semicolon or a compound statement enclosed by braces.

4. In the example program, it asks the user to enter a number from 1 to 10, and then reads the user input. The expression in the **if** statement contains a relational operator. It checks to see whether the user input is greater than 5. If the relational expression is false, then the subsequent **printf()** statement is not executed. Otherwise, the **printf()** statement will print the string **"You number is larger than 5."** on the screen. Finally, the last **printf()** statement will be executed to print the number entered by the user on the screen.



The if-else Statement

```
if (expression)
    statement1;
else
    statement2;
```

```
/* This program determines the maximum
value of num1 and num2 */
#include <stdio.h>
int main()
{
      int  num1, num2, max;
      printf(“Please enter two integers:”);
      scanf(“%d  %d”, &num1, &num2);
      if (num1 > num2)
            max = num1;
      else
            max = num2;
      printf(“The maximum of the \
            two is %d\n”,max);
      return 0;
}
```

Output

```
Please enter two integers:  9  4
The maximum of the two is 9

Please enter two integers:  -2  0
The maximum of the two is 0
```

<u>The if-else Statement</u>

1. The **if-else** statement implements a two-way selection. The format of the **if-else** statement is

   if (expression)

   statement1;

   else

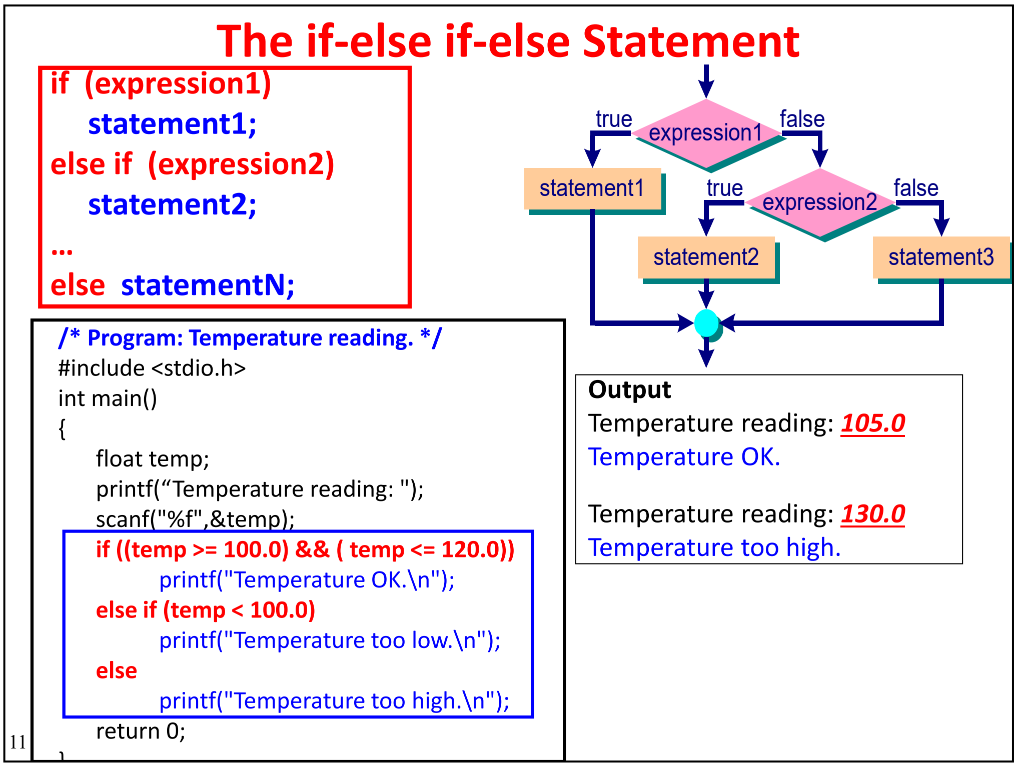
   statement2;

   if and **else** are reserved keywords.

2. When the **if-else** statement is executed, the **expression** is evaluated. If **expression** is true, then **statement1** is executed and the control is passed to the program statement following the **if** statement. If **expression** is false, then **statement2** is executed.

3. Both **statement1** and **statement2** may be a single statement terminated by a semicolon or a compound statement enclosed by **{}**.

4. In the example program, it computes the maximum number of two input integers. The two input integers are read in and stored in the variables **num1** and **num2**. The **if** statement is then used to compare the two variables. If **num1** is greater than **num2**, then the variable **max** is assigned with the value of **num1**. Otherwise, **max** is assigned with the value of **num2**. The program then prints the maximum number through the variable **max**.



The if-else if-else Statement

```
if  (expression1)
    statement1;
else if  (expression2)
    statement2;
…
else  statementN;
```

expression1
true
false
statement1
expression2
true
false
statement2
statement3

```
/* Program: Temperature reading. */
#include <stdio.h>
int main()
{
     float temp;
     printf(“Temperature reading: ");
     scanf("%f",&temp);
     if ((temp >= 100.0) && ( temp <= 120.0))
              printf("Temperature OK.\n");
     else if (temp < 100.0)
              printf("Temperature too low.\n");
     else
              printf("Temperature too high.\n");
     return 0;
}
```

Output

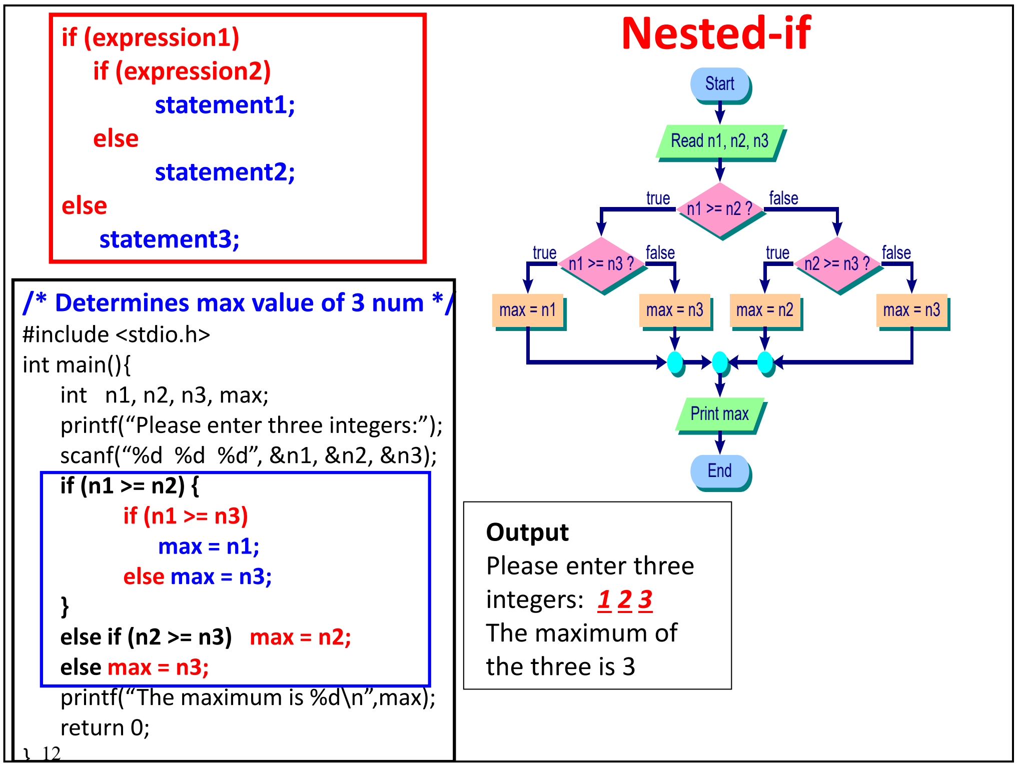
```
Temperature reading: 105.0
Temperature OK.

Temperature reading: 130.0
Temperature too high.
```

The if-else if-else Statement

1. The format for **if-else if-else** statement is

```
if (expression1)
     statement1;
else if (expression2)
     statement2;
....
else
     statementN;
```

2. Each of the **statement1**, **statement2**, **statement3**, etc. can either be a single statement terminated by a semicolon or a compound statement enclosed by **{}**.
3. If all the statements are false, then the last **statement** will be executed. In any case, only one statement will be executed, and the rest will be skipped. The last **else** part is optional and can be omitted. If the last **else** part is omitted, then no statement will be executed if all the expressions are evaluated to be false.
4. In the example program, it first reads in the temperature input. If the input value is between 100 and 120, the string **“Temperature OK.”** will be printed the screen, else if the input value is less than 100, the string **“Temperature too low”** will be displayed, else the string **“Temperature too high”** will be printed.



Nested-if

```
if (expression1)
    if (expression2)
            statement1;
    else
            statement2;
else
    statement3;
```

```
/* Determines max value of 3 num */
#include <stdio.h>
int main(){
     int   n1, n2, n3, max;
     printf("Please enter three integers:");
     scanf("%d  %d  %d", &n1, &n2, &n3);
     if (n1 >= n2) {
             if (n1 >= n3)
                  max = n1;
             else max = n3;
     }
     else if (n2 >= n3)   max = n2;
     else max = n3;
     printf("The maximum is %d\n",max);
     return 0;
}  12
```

Output
Please enter three integers: *1 2 3*
The maximum of the three is 3

Nested-if

1. The nested-**if** statement allows us to perform a multi-way selection. In a nested-**if** statement, both the **if** branch and the **else** branch may contain one or more **if** statements. The level of nested-**if** statements can be as many as the limit the compiler allows.
2. An example of a nested-**if** statement is given as shown.
3. If **expression1** and **expression2** are true, then **statement1** is executed. If **expression1** is true and **expression2** is false, then **statement2** is executed. If **expression1** is false, then **statement3** is executed. C compiler associates an **else** part with the nearest unresolved **if**, i.e. the **if** statement that does not have an **else** statement.
4. We can also use braces to enclose statements:

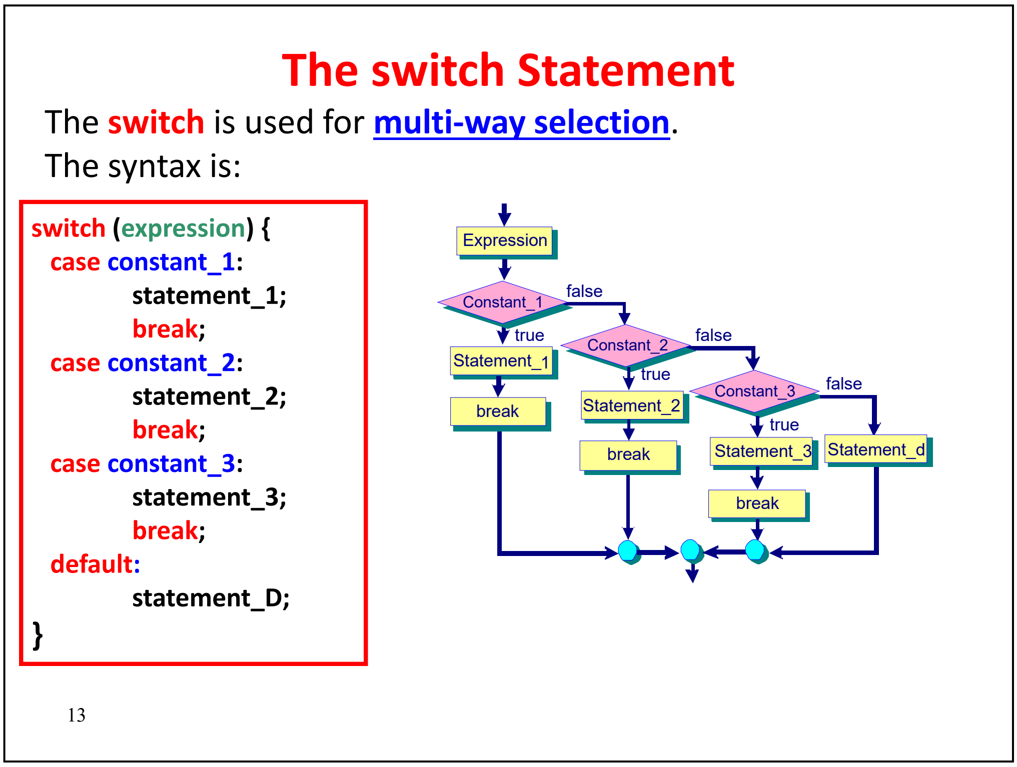
```
if (expression1) {
     if (expression2)
          statement1;
     else
          statement2;
} else
     statement3;
```

5. In the example program, it reads in three integers from the user and stores the values

in the variables **n1**, **n2** and **n3**. The values stored in **n1** and **n2** are then compared. If **n1** is greater than **n2**, then **n1** is compared with **n3**. If **n1** is greater than **n3**, then **max** is assigned with the value of **n1**. Otherwise, **max** is assigned with the value of **n3**. On the other hand, if **n1** is less than **n2**, then **n2** is compared with **n3** in a similar manner. The variable **max** is assigned with the value based on the comparison between **n2** and **n3**. Finally, the program will print the maximum value of the three integers via the variable **max**.

6. Note that in this example braces are used to enclose the statements for the inner **if** condition:

```
if (n1 >= n2) {
    if (n1 >= n3)
        max = n1;
    else max = n3;
}
```



The switch Statement

The switch is used for multi-way selection.

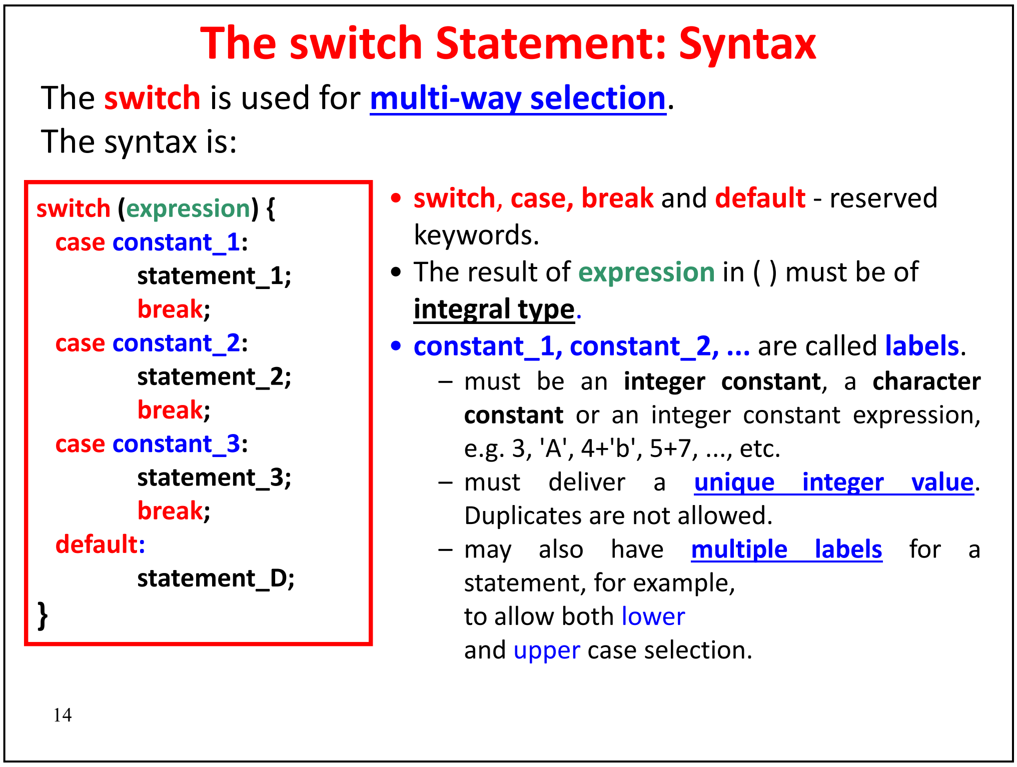
The syntax is:

```
switch (expression) {
  case constant_1:
          statement_1;
          break;
  case constant_2:
          statement_2;
          break;
  case constant_3:
          statement_3;
          break;
  default:
          statement_D;
}
```

13

The switch Statement

1. In the **switch** statement, **statement** is executed only when the **expression** has the corresponding value of **constant**.
2. The **default** part is optional. If it is there, **statement_D** is executed when **expression** has a value different from all the values specified by all the cases.
3. The **break** statement signals the end of a particular case and causes the execution of the **switch** statement to be terminated.
4. Each of the **statement** may be a single statement terminated by a semicolon or a compound statement enclosed by braces **{ }**.
5. Note that there are several restrictions in the use of the **switch** statement. The **expression** of the **switch** statement must return a result of *integer* or *character* data type. The value in the **constant** label part must be an integer constant or character constant. Moreover, it does not support a range of values to be specified.



The switch Statement: Syntax

The **switch** is used for **multi-way selection**.

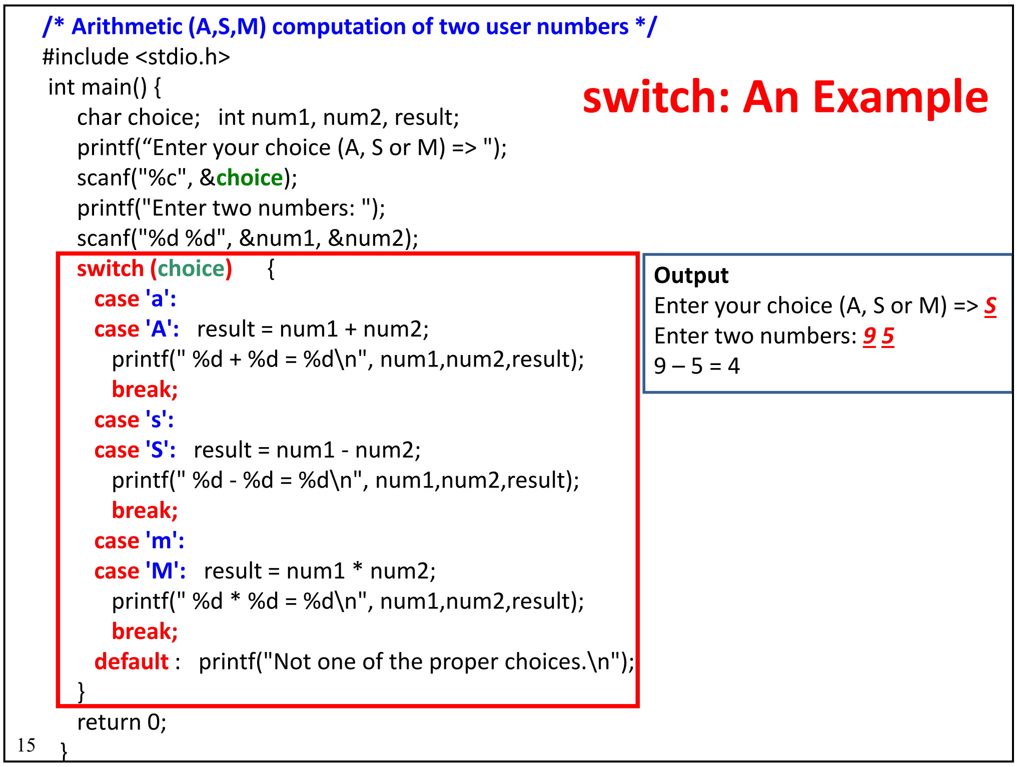
The syntax is:

```
switch (expression) {
  case constant_1:
          statement_1;
          break;
  case constant_2:
          statement_2;
          break;
  case constant_3:
          statement_3;
          break;
  default:
          statement_D;
}
```

- **switch, case, break** and **default** - reserved keywords.
- The result of **expression** in ( ) must be of **integral type**.
- **constant_1, constant_2, ...** are called **labels**.
  - must be an **integer constant**, a **character constant** or an integer constant expression, e.g. 3, 'A', 4+'b', 5+7, ..., etc.
  - must deliver a **unique integer value**. Duplicates are not allowed.
  - may also have **multiple labels** for a statement, for example, to allow both lower and upper case selection.

The switch Statement: Syntax

1. The **switch** statement provides a multi-way decision structure in which one of the several statements is executed depending on the value of an expression.
2. The syntax of a **switch** statement is shown.
3. **switch**, **case**, **break** and **default** are reserved keywords. **constant_1**, **constant_2**, etc. are called *labels*.
4. Each must be an integer constant, a character constant or an integer constant expression, e.g. 3, 'A', 4+'b', 5+7, etc.
5. Each of the labels must deliver a unique integer value. Duplicates are not allowed.
6. Multiple labels are allowed, for example, to support both lower and upper case selection.



switch: An Example

```
/* Arithmetic (A,S,M) computation of two user numbers */
#include <stdio.h>
 int main() {
     char choice;   int num1, num2, result;
     printf(“Enter your choice (A, S or M) => ");
     scanf("%c", &choice);
     printf("Enter two numbers: ");
     scanf("%d %d", &num1, &num2);
     switch (choice)     {
       case 'a':
       case 'A':   result = num1 + num2;
         printf(" %d + %d = %d\n", num1,num2,result);
         break;
       case 's':
       case 'S':   result = num1 - num2;
         printf(" %d - %d = %d\n", num1,num2,result);
         break;
       case 'm':
       case 'M':   result = num1 * num2;
         printf(" %d * %d = %d\n", num1,num2,result);
         break;
       default :   printf("Not one of the proper choices.\n");
     }
     return 0;
 }
```

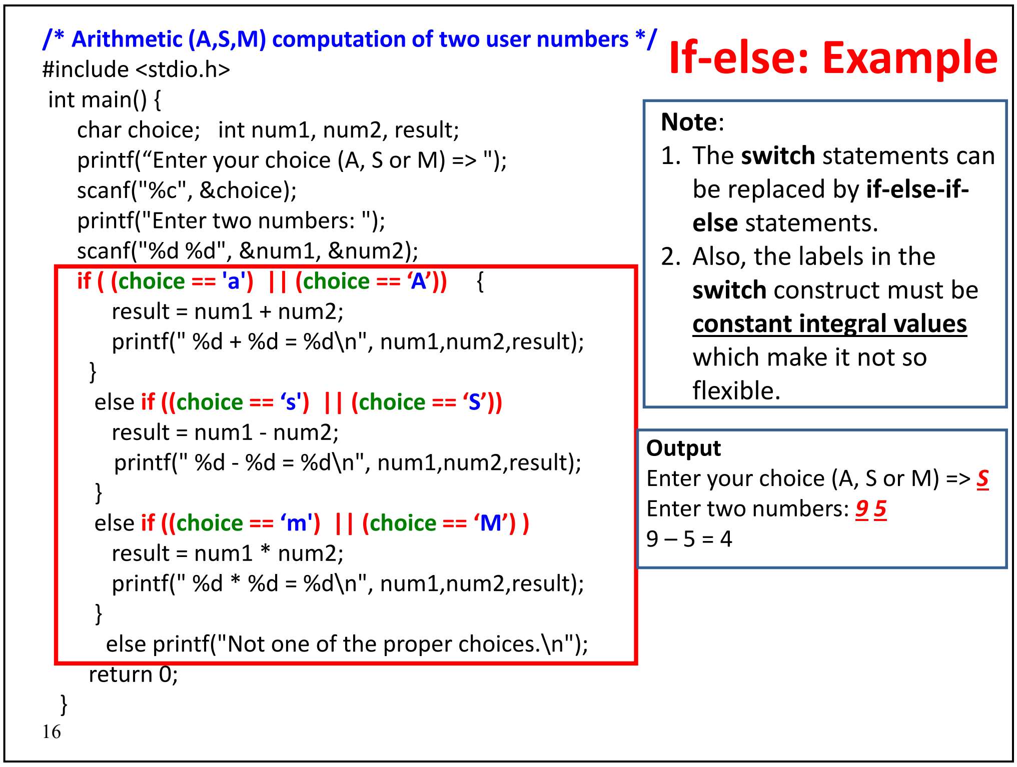
Output

```
Enter your choice (A, S or M) => S
Enter two numbers: 9 5
9 – 5 = 4
```

15

The switch Statement: Example

1. The **switch** statement is quite commonly used in menu-driven applications.
2. In this example program, it uses the **switch** statement for menu-driven selection. The program displays a list of arithmetic operations (i.e. addition, subtraction and multiplication) that the user can enter. Then, the user selects the operation command and enters two operands.
3. The **switch** statement is then used to control which operation is to be executed based on user selection. The control is transferred to the appropriate branch of the **case** condition based on the variable **choice**.
4. The statements under the **case** condition are executed and the result of the arithmetic operation will be printed.
5. For example, if user selects ‘S’, then the subtraction operation of the two numbers 9 and 5 will be computed.
6. In addition, we may also have *multiple labels* for a statement. As such, we can allow the choice to be specified in both lowercase and uppercase letters entered by the user.



If-else: Example

```
/* Arithmetic (A,S,M) computation of two user numbers */
#include <stdio.h>
 int main() {
     char choice;   int num1, num2, result;
     printf(“Enter your choice (A, S or M) => ");
     scanf("%c", &choice);
     printf("Enter two numbers: ");
     scanf("%d %d", &num1, &num2);
     if ( (choice == 'a')  || (choice == ‘A’))    {
         result = num1 + num2;
         printf(" %d + %d = %d\n", num1,num2,result);
      }
       else if ((choice == ‘s')  || (choice == ‘S’))
         result = num1 - num2;
         printf(" %d - %d = %d\n", num1,num2,result);
       }
       else if ((choice == ‘m')  || (choice == ‘M’) )
         result = num1 * num2;
         printf(" %d * %d = %d\n", num1,num2,result);
       }
        else printf("Not one of the proper choices.\n");
      return 0;
  }
```

Note:

1. The **switch** statements can be replaced by **if-else-if-else** statements.
2. Also, the labels in the **switch** construct must be **constant integral values** which make it not so flexible.

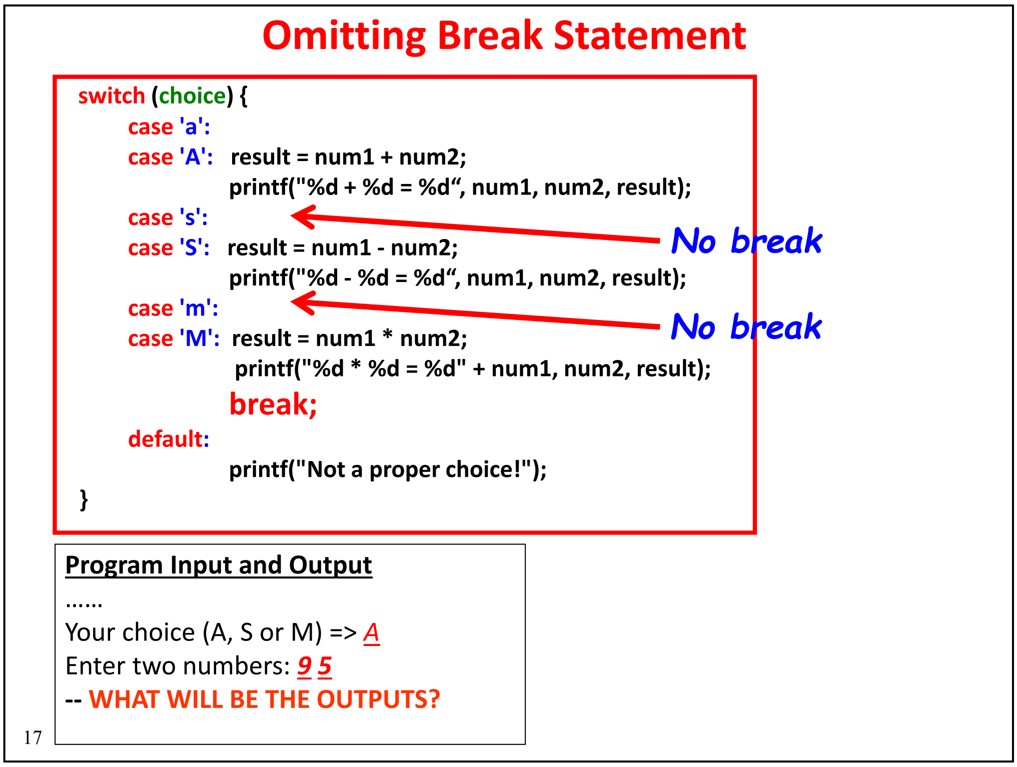
Output

```
Enter your choice (A, S or M) => S
Enter two numbers: 9 5
9 – 5 = 4
```

16

The if-else if-else statement: Example

1. The same program on supporting arithmetic operation can also be implemented using the **if-else-if-else** statements.
2. For example, if user selects ‘S’, then the subtraction operation of the two numbers 9 and 5 will be computed according to the if-else condition.
3. Generally, the **switch** statements can be replaced by **if-else-if-else** statements. However, as labels in the **switch** construct must be constant values (i.e. integer, character or integer/character expression), we will not be able to convert certain **if-else** statements into **switch** statements.



Omitting Break Statement

```
switch (choice) {
      case 'a':
      case 'A':   result = num1 + num2;
                  printf("%d + %d = %d“, num1, num2, result);
      case 's':
      case 'S':   result = num1 - num2;
                  printf("%d - %d = %d“, num1, num2, result);
      case 'm':
      case 'M':  result = num1 * num2;
                   printf("%d * %d = %d" + num1, num2, result);
                  break;
      default:
                  printf("Not a proper choice!");
}
```

No break

No break

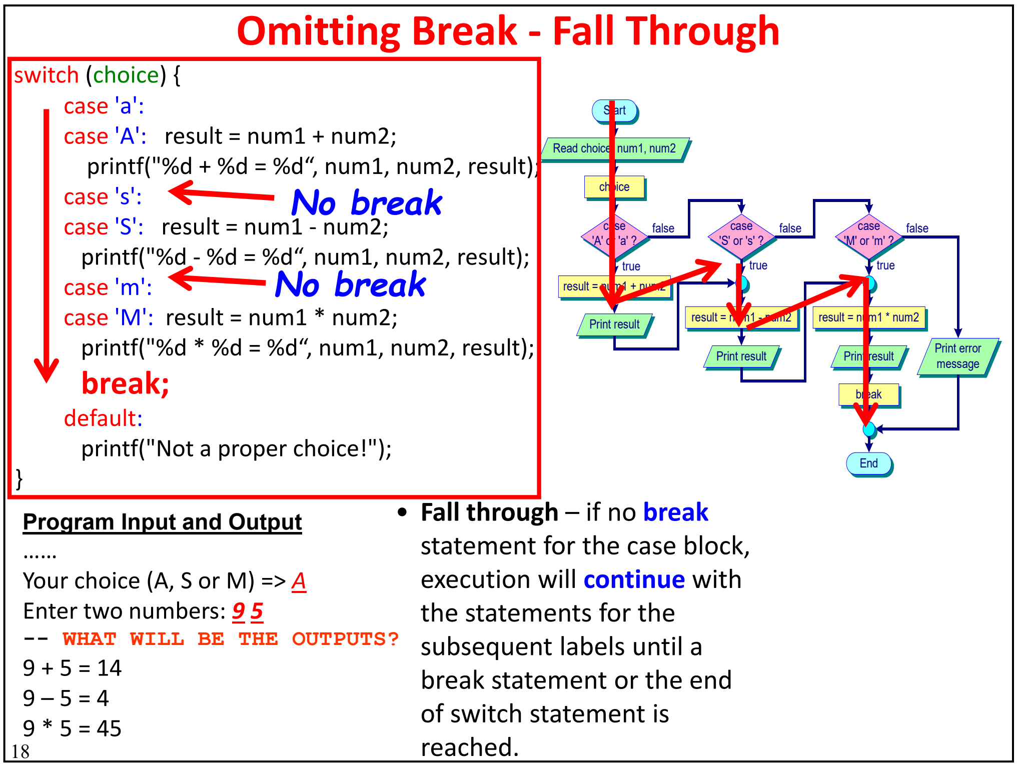
Program Input and Output

```
……
Your choice (A, S or M) => A
Enter two numbers: 9 5
-- WHAT WILL BE THE OUTPUTS?
```

17

The switch Statement – Omitting The break Statement

1. In the **switch** statement, the **break** statement is placed at the end of each **case**.
2. What will happen if we omit the **break** statement as shown in this example?
3. The **break** statement was omitted for the case **‘A’** and case **‘S’**.
4. If the user enters the choice **'A’** with the two numbers 9 and 5, what will be the outputs of the program?



Omitting Break - Fall Through

```
switch (choice) {
    case 'a':
    case 'A':   result = num1 + num2;
        printf("%d + %d = %d“, num1, num2, result);
    case 's':       No break
    case 'S':   result = num1 - num2;
        printf("%d - %d = %d“, num1, num2, result);
    case 'm':       No break
    case 'M':  result = num1 * num2;
        printf("%d * %d = %d“, num1, num2, result);
        break;
    default:
        printf("Not a proper choice!");
}
```

Program Input and Output

```
……
Your choice (A, S or M) => A
Enter two numbers: 9 5
-- WHAT WILL BE THE OUTPUTS?
9 + 5 = 14
9 – 5 = 4
9 * 5 = 45
```

- **Fall through** – if no **break** statement for the case block, execution will **continue** with the statements for the subsequent labels until a break statement or the end of switch statement is reached.

18

The switch Statement – Omitting break

1. The **break** statement is used to end each **case** constant block.
2. If we do not put the **break** statement for the case block, execution will continue with the statements for the subsequent **case** labels until a **break** statement or the end of the **switch** statement is reached. This is called the *fall through* situation.
3. Therefore, if the **break** statement is omitted, the input and output of the program will become:

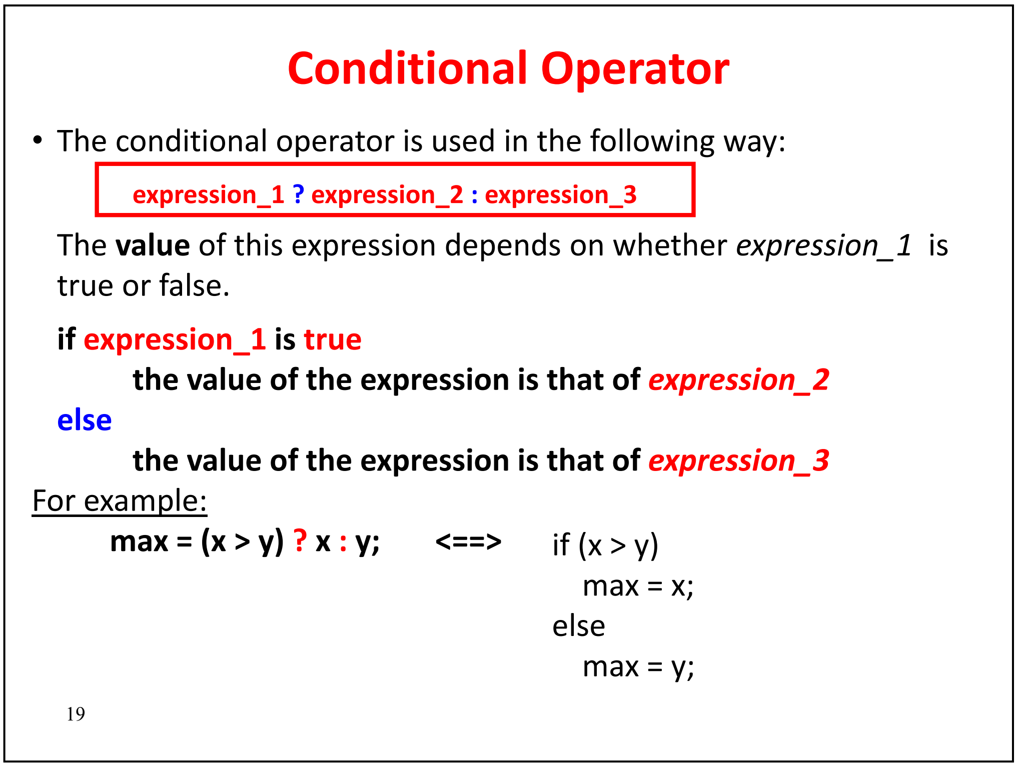
    Your choice (A, S or M) => A
    Enter two numbers: 9 5

    9 + 5 = 14

    9 - 5 = 4

    9 * 5 = 45

3. That is, the statements for the subtraction and multiplication operations are also executed.



Conditional Operator

- The conditional operator is used in the following way:

  expression_1 ? expression_2 : expression_3

  The **value** of this expression depends on whether *expression_1* is true or false.

  if expression_1 is true

  the value of the expression is that of *expression_2*

  else

  the value of the expression is that of *expression_3*

For example:

```
max = (x > y) ? x : y;     <==>   if (x > y)
                                     max = x;
                                  else
                                     max = y;
```

19

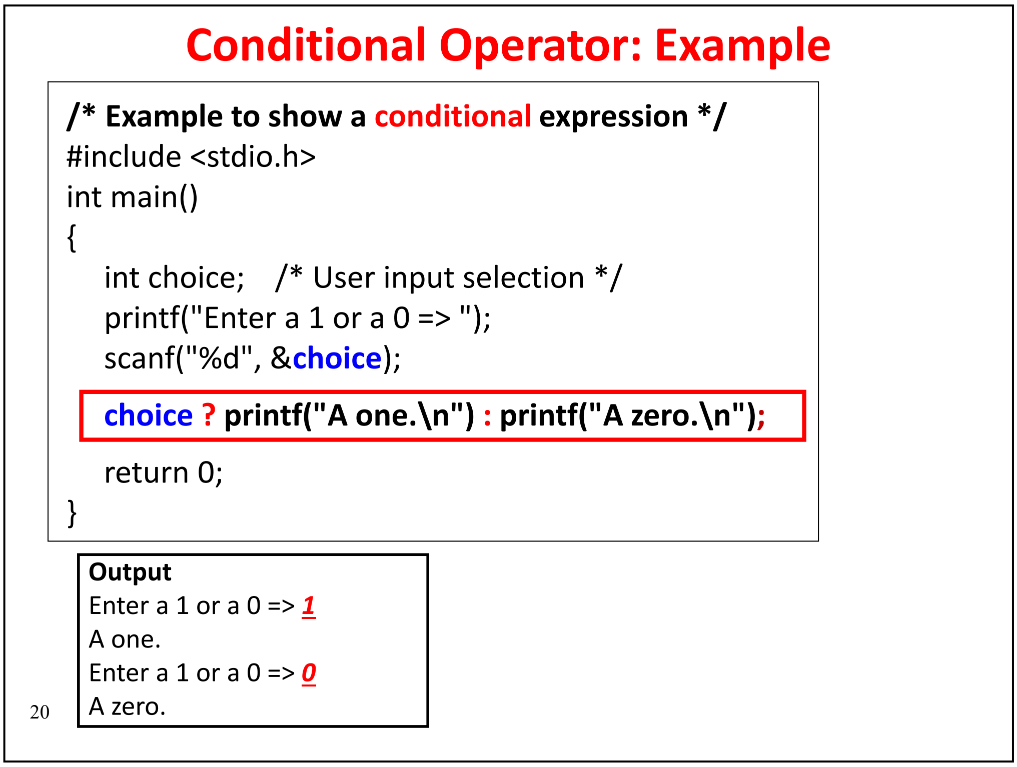
Conditional Operator

1. The conditional operator is a ternary operator, which takes three expressions, with the first two expressions separated by a '**?**' and the second and third expressions separated by a '**:** '.
2. The conditional operator is specified in the following way:

   expression_1 ? expression_2 : expression_3

3. The value of this expression depends on whether **expression_1** is true or false. If **expression_1** is true, the value of the expression becomes the value of **expression_2**, otherwise it is **expression_3**.
4. The conditional operator is commonly used in an assignment statement, which assigns one of the two values to a variable. For example, the maximum value of the two values **x** and **y** can be obtained using the following statement: **max = x > y ? x : y;** The assignment statement is equivalent to the following **if-else** statement:

```
if (x > y)
    max = x;
else
    max = y;
```



Conditional Operator: Example

```
/* Example to show a conditional expression */
#include <stdio.h>
int main()
{
   int choice;   /* User input selection */
   printf("Enter a 1 or a 0 => ");
   scanf("%d", &choice);

   choice ? printf("A one.\n") : printf("A zero.\n");

   return 0;
}
```

```
Output
Enter a 1 or a 0 => 1
A one.
Enter a 1 or a 0 => 0
A zero.
```

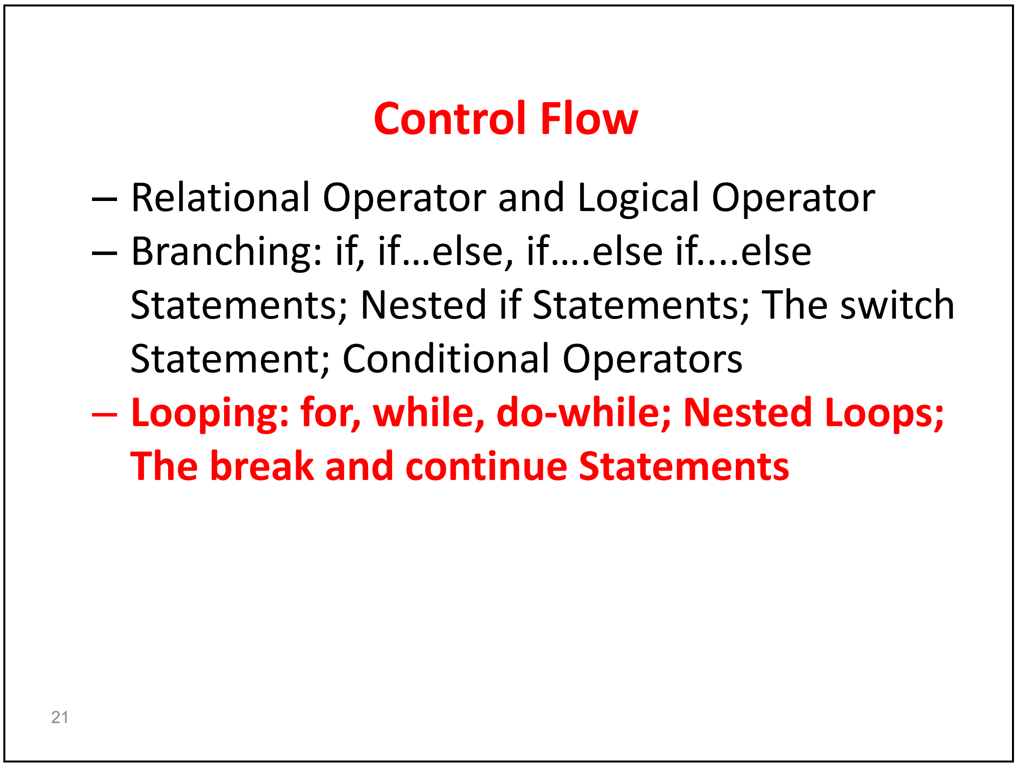
20

Conditional Operator: Example

1. This program gives an example on the use of the conditional operator.
2. The program statement that uses conditional operator is:

   choice ? printf("A one.\n") : printf("A zero.\n");

1. The program first reads a user input on the variable **choice**. If **choice** is 1 (i.e. true), then the string "**A one.**" will be printed. Otherwise if **choice** is 0 (i.e. false), the string "**A zero.**" will be printed.
4. However, this can also be implemented quite easily using the **if-else** statement.



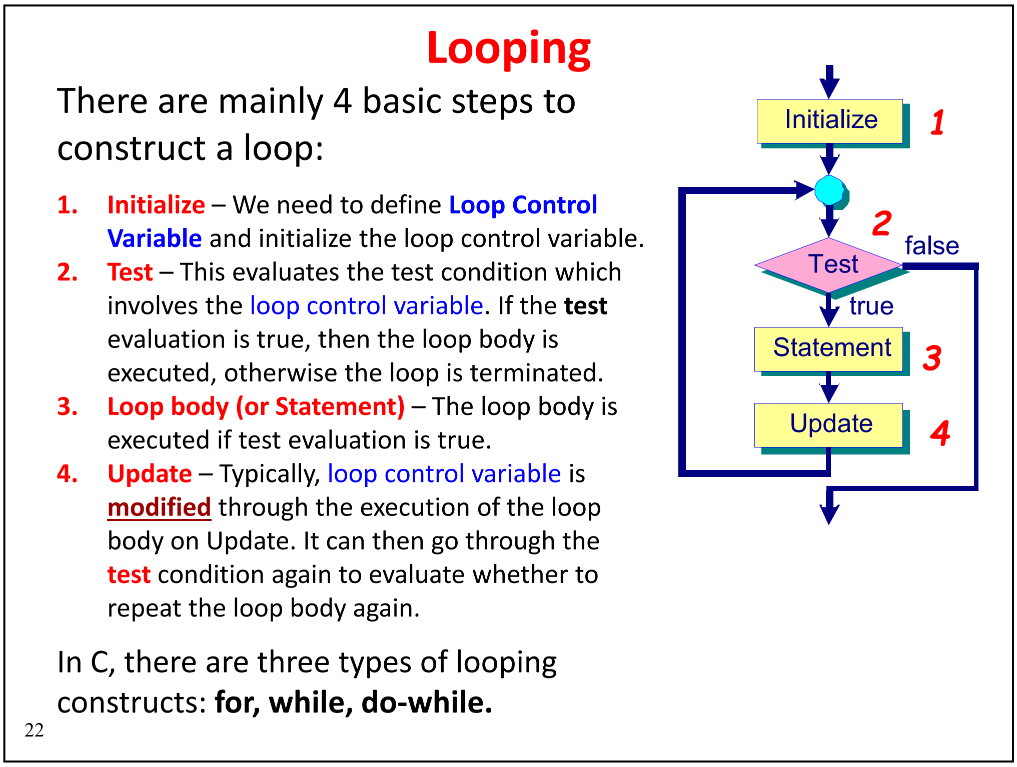
Control Flow

- Relational Operator and Logical Operator
- Branching: if, if…else, if….else if….else Statements; Nested if Statements; The switch Statement; Conditional Operators
- **Looping: for, while, do-while; Nested Loops; The break and continue Statements**

21

Control Flow

1. Here, we discuss the looping statements.



Looping

There are mainly 4 basic steps to construct a loop:

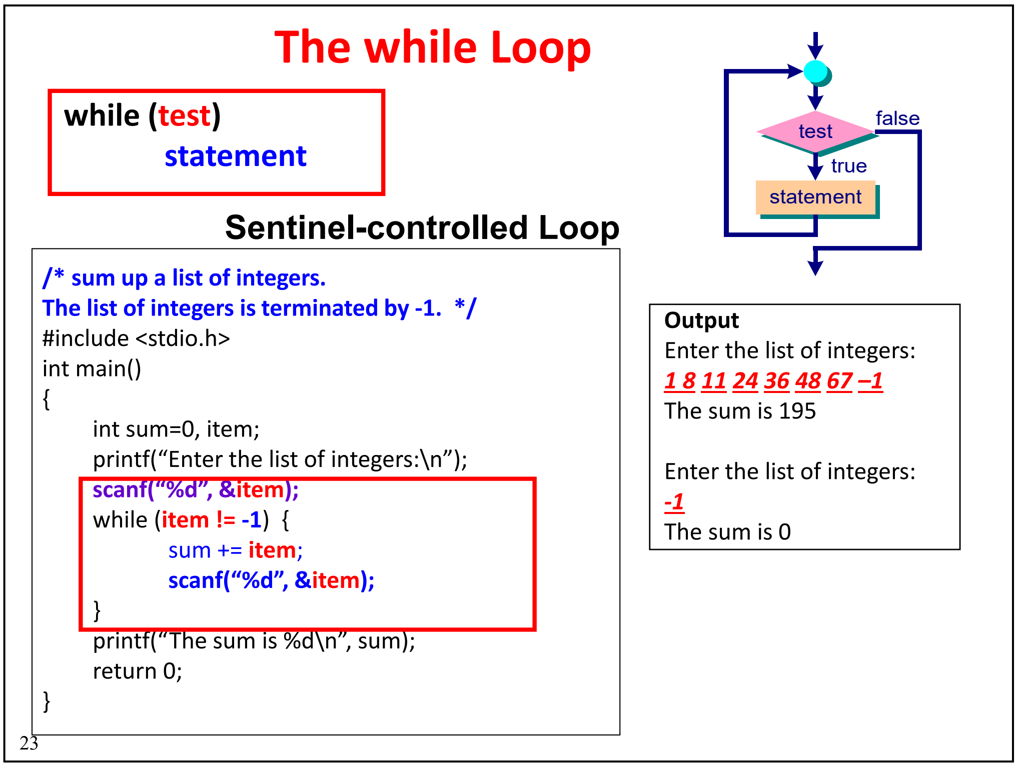
1. **Initialize** – We need to define **Loop Control Variable** and initialize the loop control variable.
2. **Test** – This evaluates the test condition which involves the loop control variable. If the **test** evaluation is true, then the loop body is executed, otherwise the loop is terminated.
3. **Loop body (or Statement)** – The loop body is executed if test evaluation is true.
4. **Update** – Typically, loop control variable is **modified** through the execution of the loop body on Update. It can then go through the **test** condition again to evaluate whether to repeat the loop body again.

In C, there are three types of looping constructs: **for, while, do-while.**

22

Looping

1. To construct loops, we need the following four basic steps:
   1) **Initialize** - This defines and initializes the loop control variable, which is used to control the number of repetitions of the loop.
   2) **Test** - This evaluates the **test** condition. If the **test** condition is true, then the loop body is executed, otherwise the loop is terminated. The **while** loop and **for** loop evaluate the **test** condition at the beginning of the loop, while the **do-while** loop evaluates the **test** condition at the end of the loop.
   3) **Loop body** - The **loop body** is executed if the **test** condition is evaluated to be true. It contains the actual actions to be carried out.
   4) **Update** - The **test** condition typically involves the loop control variable that should be modified each time through the execution of the loop body. The loop control variable will go through the **test** condition again to determine whether to repeat the loop body again.
2. We can use one of the three looping constructs, namely the **while** loop, the **for** loop and the **do-while** loop, for constructing loops.



The while Loop

while (test)
statement

Sentinel-controlled Loop

```
/* sum up a list of integers.
The list of integers is terminated by -1.  */
#include <stdio.h>
int main()
{
        int sum=0, item;
        printf("Enter the list of integers:\n");
        scanf("%d", &item);
        while (item != -1)  {
                sum += item;
                scanf("%d", &item);
        }
        printf("The sum is %d\n", sum);
        return 0;
}
```

```
Output
Enter the list of integers:
1 8 11 24 36 48 67 –1
The sum is 195

Enter the list of integers:
-1
The sum is 0
```

23

The while Loop

1. The format of the **while** statement is

   while (test)

   statement;

2. **while** is a reserved keyword. **statement** can be a simple statement or a compound statement.

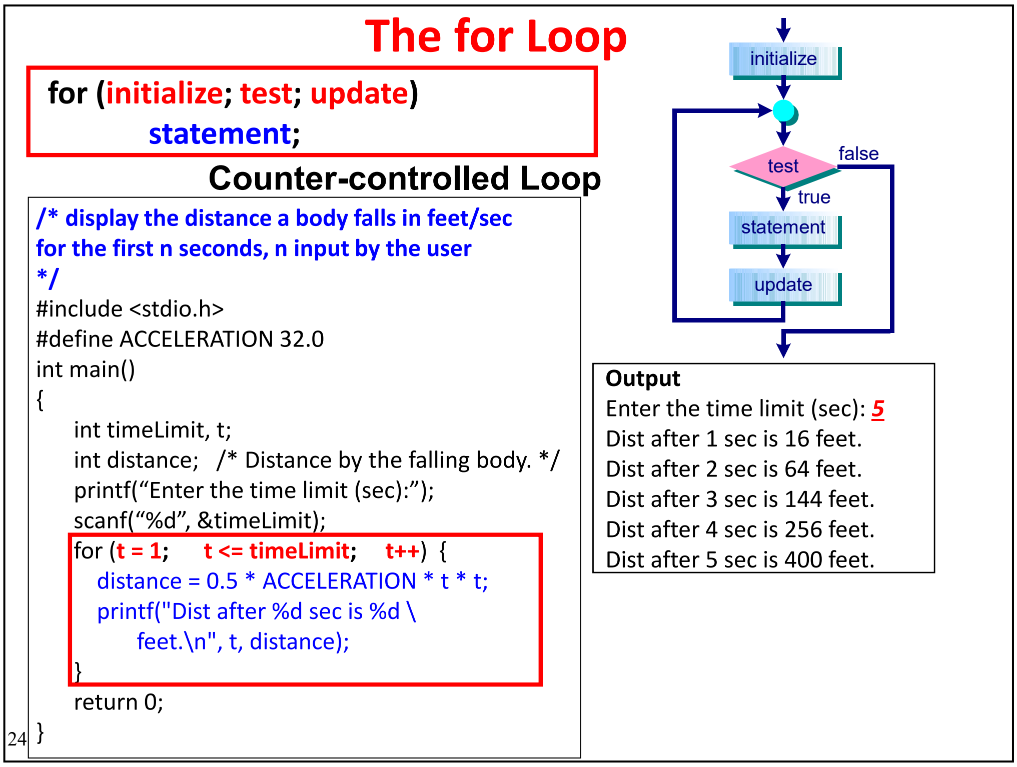
3. The **while** statement is executed by evaluating the **test** condition. If the result is true, then **statement** is executed. Control is then transferred back to the beginning of the **while** statement, and the process repeats again. This looping continues until the **test** condition finally becomes false. When the **test** condition is false, the loop terminates and the program continues execute the next sequential statement.

4. The **while** loop is best to be used as a **sentinel-controlled** loop in situations where the number of times the loop to be repeated is not known in advance.

5. In the example program, it aims to sum up a list of integers which is terminated by -1. It does not know how many data items are to be read at the beginning of the program. It will keep on reading the data until the user input is –1 which is the sentinel value.

6. In the program, the **scanf()** statement reads in the first number and stores it in the variable **item**. Then, the execution of the **while** loop begins. If the initial **item** value is not –1, then the statements in the braces are executed. The **item** value is first added to another variable **sum**. Another **item** value is then read in from the user, and the control

is transferred back to the **test** expression **(item != -1)** for evaluation. This process repeats until the **item** value becomes –1.



The for Loop

for (initialize; test; update)
statement;

Counter-controlled Loop

```
/* display the distance a body falls in feet/sec
for the first n seconds, n input by the user
*/
#include <stdio.h>
#define ACCELERATION 32.0
int main()
{
     int timeLimit, t;
     int distance;   /* Distance by the falling body. */
     printf(“Enter the time limit (sec):”);
     scanf(“%d”, &timeLimit);
     for (t = 1;     t <= timeLimit;     t++)  {
        distance = 0.5 * ACCELERATION * t * t;
        printf("Dist after %d sec is %d \
             feet.\n", t, distance);
     }
     return 0;
}
```

```
Output
Enter the time limit (sec): 5
Dist after 1 sec is 16 feet.
Dist after 2 sec is 64 feet.
Dist after 3 sec is 144 feet.
Dist after 4 sec is 256 feet.
Dist after 5 sec is 400 feet.
```

24

The for Loop

1. The **for** statement allows us to repeat a sequence of statements for a specified number of times which is known in advance. The format of the **for** statement is given as follows:

   for (initialize; test; update)

   statement;

2. **for** is a reserved keyword. **statement** can be a simple statement or a compound statement.

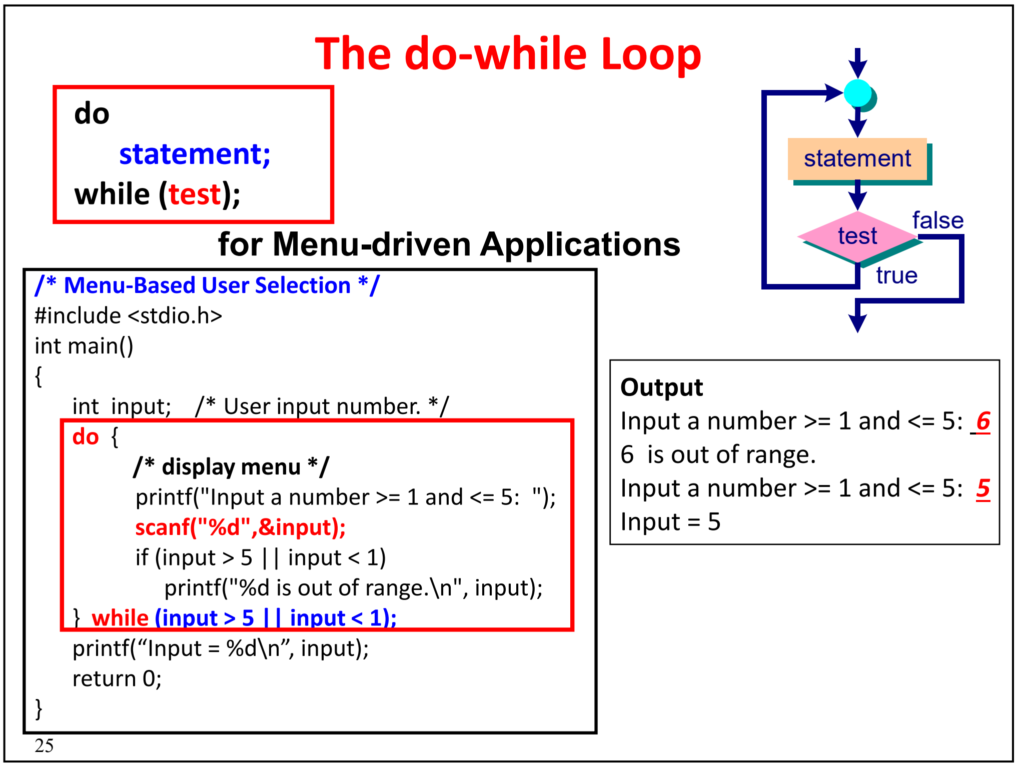
3. **initialize** is usually used to set the initial value of one or more loop control variables. Generally, **test** is a relational expression to control iterations. **update** is used to update some loop control variables before repeating the loop.

4. In the **for** loop, **initialize** is first evaluated. The **test** condition is then evaluated. If the **test** condition is true, then the **statement** and **update** expression are executed. Control is then transferred to the **test** condition, and the loop is repeated again if the **test** condition is true. If the **test** condition is false, then the loop is terminated. The control is then transferred out of the **for** loop to the next sequential statement.

5. The **for** loop is mainly used as a counter-controlled loop.

6. In the example program, it reads in the time limit and stores it in the variable **timeLimit**. It then uses the **for** loop as a counter-controlled loop. The loop control variable **t** is used to control the number of repetitions in the loop. The variable **t** is initialized to 1. In the loop body, the distance calculation formula is used to compute

the distance. The loop control variable **t** is also incremented by 1 every time the loop body finishes executing. The loop will stop when **t** equals to **timeLimit**.



The do-while Loop

```
do
    statement;
while (test);
```

statement
test
false
true

for Menu-driven Applications

```
/* Menu-Based User Selection */
#include <stdio.h>
int main()
{
    int  input;   /* User input number. */
    do  {
          /* display menu */
          printf("Input a number >= 1 and <= 5:  ");
          scanf("%d",&input);
          if (input > 5 || input < 1)
              printf("%d is out of range.\n", input);
    } while (input > 5 || input < 1);
    printf("Input = %d\n", input);
    return 0;
}
```

Output

```
Input a number >= 1 and <= 5: 6
6  is out of range.
Input a number >= 1 and <= 5:  5
Input = 5
```

25

The do-while Loop

1. Both the **while** and the **for** loops test the condition prior to executing the loop body. C also provides the **do-while** statement which implements a control pattern different from the **while** and **for** loops. The body of a **do-while** loop is executed at least once.

2. The general format is

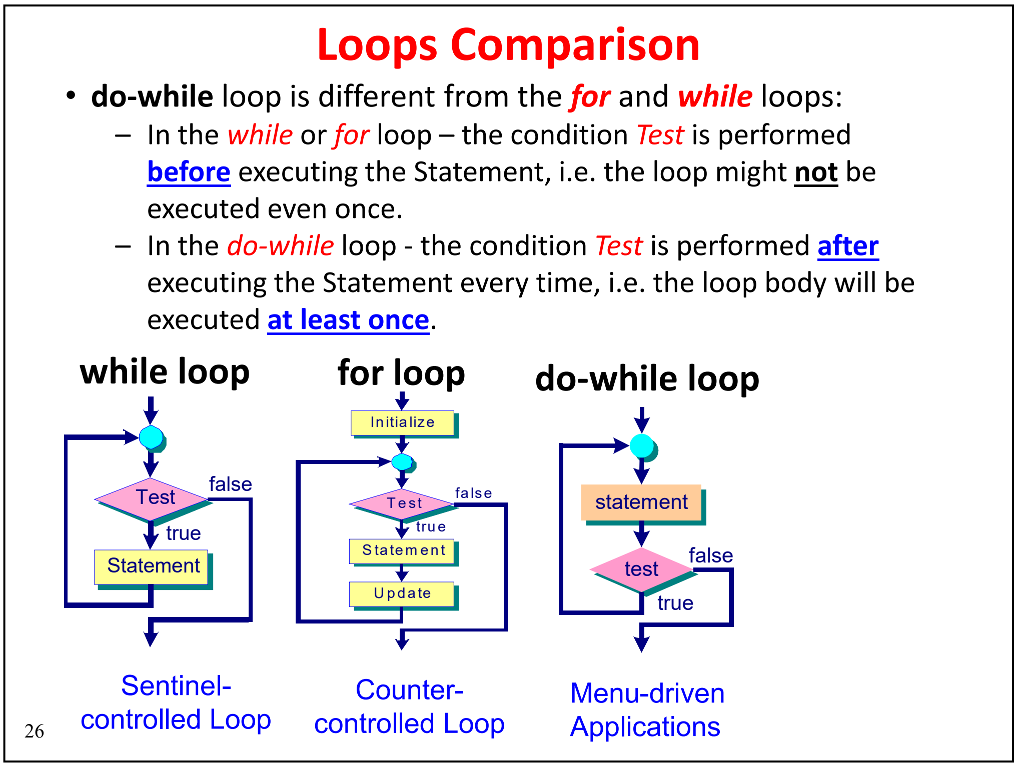
   do

   statement;

   while (test);

3. The **do-while** loop differs from the **while** and **for** statements in that the condition **test** is only performed after the **statement** has finished execution. This means the loop will be executed at least once. On the other hand, the body of the **while** or **for** loop might not be executed even once.

4. In the example program, it aims to implement a menu-driven application. The program reads in a number between 1 and 5. If the number entered is not within the range, an error message is printed on the display to prompt the user to input the number again. The program will read the user input at least once.



Loops Comparison

- **do-while** loop is different from the ***for*** and ***while*** loops:
  - In the *while* or *for* loop – the condition *Test* is performed **before** executing the Statement, i.e. the loop might **not** be executed even once.
  - In the *do-while* loop - the condition *Test* is performed **after** executing the Statement every time, i.e. the loop body will be executed **at least once**.

while loop **for loop** **do-while loop**

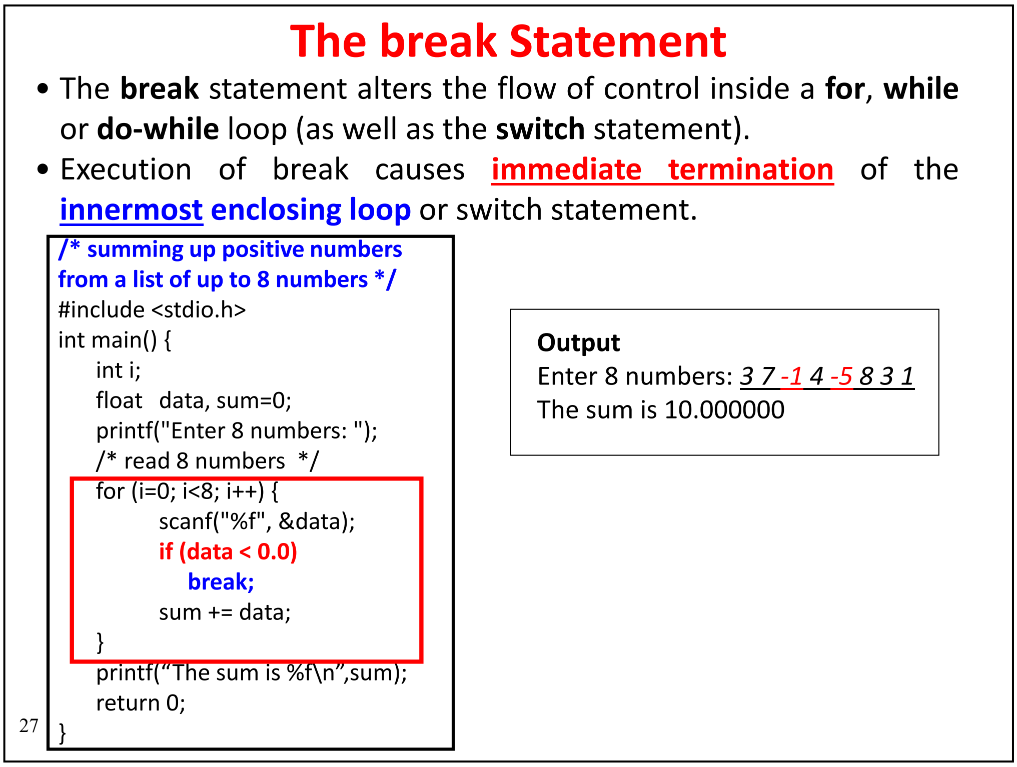
Sentinel-controlled Loop

Counter-controlled Loop

Menu-driven Applications

Loops Comparison

1. The **while** loop is mainly used for sentinel-controlled loops.
2. The **for** loop is mainly used for counter-controlled loops.
3. The **do-while** loop differs from the **while** loop in that the **while** loop evaluates the **test** condition at the beginning of the loop, whereas the **do-while** loop evaluates the **test** condition at the end of the loop. If the initial **test** condition is true, the two loops will have the same number of iterations. The number of iterations between the two loops will differ only when the initial **test** condition is false. In this case, the **while** loop will exit without executing any statements in the loop body. But the **do-while** loop will execute the loop body at least once before exiting from the loop.
4. Therefore, the **do-while** statement is useful for situations such as **menu-driven applications** which may require executing the loop body at least once.



The break Statement

- The **break** statement alters the flow of control inside a **for**, **while** or **do-while** loop (as well as the **switch** statement).
- Execution of break causes immediate termination of the innermost enclosing loop or switch statement.

```
/* summing up positive numbers
from a list of up to 8 numbers */
#include <stdio.h>
int main() {
     int i;
     float   data, sum=0;
     printf("Enter 8 numbers: ");
     /* read 8 numbers  */
     for (i=0; i<8; i++) {
             scanf("%f", &data);
             if (data < 0.0)
                 break;
             sum += data;
     }
     printf("The sum is %f\n",sum);
     return 0;
}
```

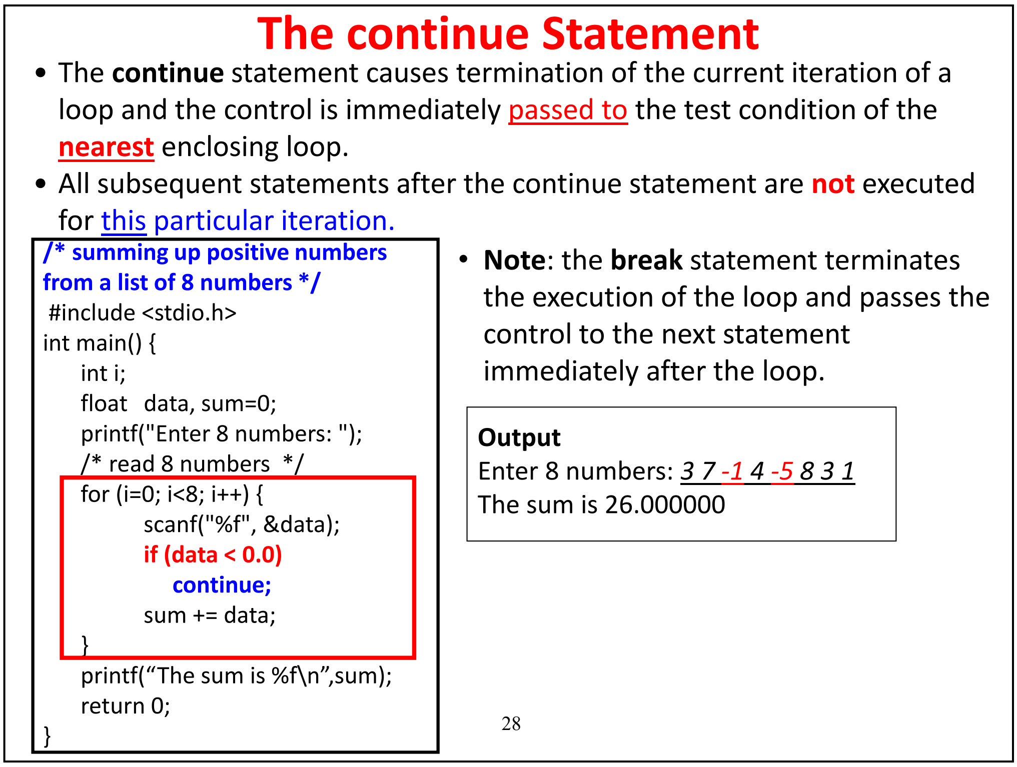
Output

```
Enter 8 numbers: 3 7 -1 4 -5 8 3 1
The sum is 10.000000
```

27

The break statement

1. The **break** statement alters flow of control inside a **for**, **while** or **do-while** loop, as well as the **switch** statement.
2. The execution of **break** causes immediate termination of the innermost enclosing loop or the **switch** statement. The control is then transferred to the next sequential statement following the loop.
3. In the example program, it aims to sum up positive numbers from a list of numbers until a negative number is encountered. The program reads the input numbers of data type **float** for at most 8 numbers. A **for** loop is used to process the input number one by one. If the number is not less than zero, then the value is added to **sum.** Otherwise the **break** statement is used to terminate the loop. The control is then transferred to the next statement following the loop construct.



The continue Statement

- The **continue** statement causes termination of the current iteration of a loop and the control is immediately passed to the test condition of the **nearest** enclosing loop.
- All subsequent statements after the continue statement are **not** executed for this particular iteration.

```
/* summing up positive numbers
from a list of 8 numbers */
 #include <stdio.h>
int main() {
     int i;
     float   data, sum=0;
     printf("Enter 8 numbers: ");
     /* read 8 numbers  */
     for (i=0; i<8; i++) {
              scanf("%f", &data);
              if (data < 0.0)
                  continue;
              sum += data;
     }
     printf(“The sum is %f\n”,sum);
     return 0;
}
```

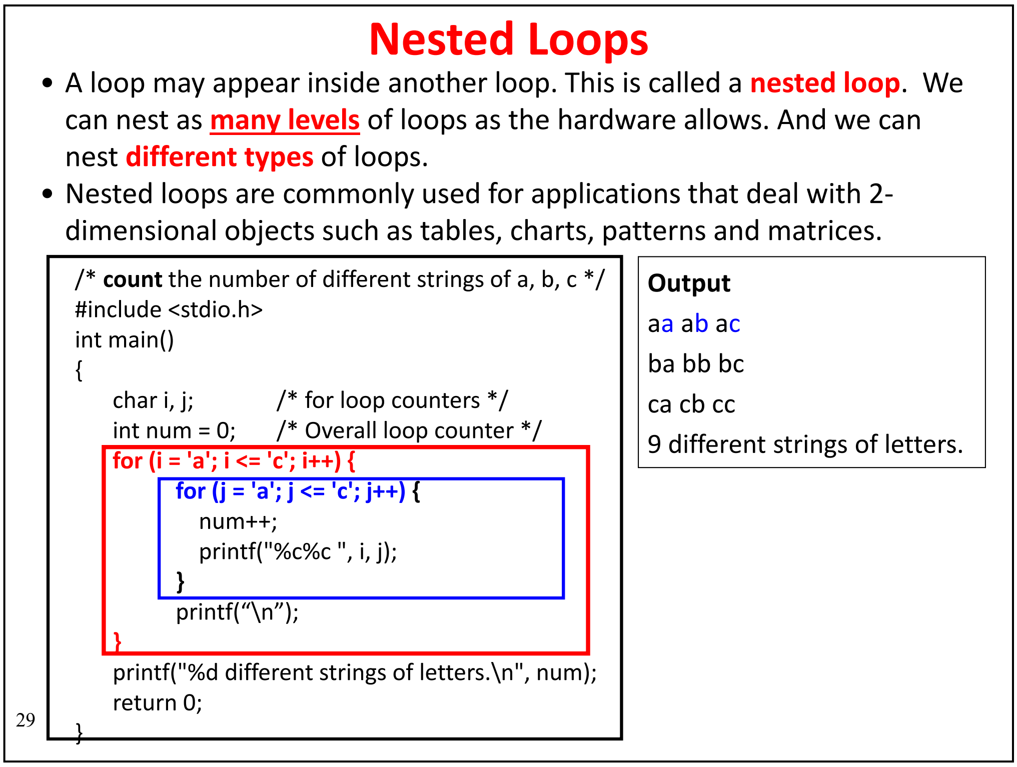
- **Note**: the **break** statement terminates the execution of the loop and passes the control to the next statement immediately after the loop.

Output

```
Enter 8 numbers: 3 7 -1 4 -5 8 3 1
The sum is 26.000000
```

The continue Statement

1. The **continue** statement causes termination of the current iteration of a loop and the control is immediately passed to the **test** condition of the nearest enclosing loop. All subsequent statements after the **continue** statement are not executed for this particular iteration.
2. The **continue** statement differs from the **break** statement in that the **continue** statement terminates the execution of the current iteration, and the loop still carries on with the next iteration if the **test** condition is fulfilled, while the **break** statement terminates the execution of the loop and passes the control to the next statement immediately after the loop.
3. In the example program, it aims to sum up only positive numbers from a list of 8 numbers. The program reads eight numbers of data type **float**. A **for** loop is used to process the input number one by one. If the number is not less than zero, then the value is added to **sum.** Otherwise the **continue** statement is used to terminate the current iteration of the loop, and the control is transferred to the next iteration of the loop. Notice that the **for** loop will process all the eight numbers when they are read in.



Nested Loops

- A loop may appear inside another loop. This is called a **nested loop**. We can nest as **many levels** of loops as the hardware allows. And we can nest **different types** of loops.
- Nested loops are commonly used for applications that deal with 2-dimensional objects such as tables, charts, patterns and matrices.

```
/* count the number of different strings of a, b, c */
#include <stdio.h>
int main()
{
     char i, j;            /* for loop counters */
     int num = 0;       /* Overall loop counter */
     for (i = 'a'; i <= 'c'; i++) {
            for (j = 'a'; j <= 'c'; j++) {
               num++;
               printf("%c%c ", i, j);
            }
            printf(“\n”);
     }
     printf("%d different strings of letters.\n", num);
     return 0;
}
```

Output

```
aa ab ac
ba bb bc
ca cb cc
9 different strings of letters.
```

Nested Loops

1. A loop may appear inside another loop. This is called a nested loop. We can nest as many levels of loops as the system allows. We can also nest different types of loops.
2. In the program, it generates the different strings of letters from the characters '**a**', '**b**' and '**c**'. The program contains a nested loop, in which one **for** loop is nested inside another **for** loop. The counter variables **i** and **j** are used to control the **for** loops. Another variable **num** is used as an overall counter to record the number of strings that have been generated by the different combinations of the characters.
3. The nested loop is executed as follows. In the outer **for** loop, when the counter variable **i='a'**, the inner **for** loop is executed, and generates the different strings **aa**, **ab** and **ac**. When **i='b'** in the outer **for** loop, the inner **for** loop is executed and generates **ba**, **bb** and **bc**. Similarly, when **i='c'** in the outer **for** loop, the inner **for** loop generates **ca**, **cb** and **cc**. The nested loop is then terminated. The total number of strings generated is also printed on the screen.
4. Nested loops are commonly used for applications that deal with 2-dimensional objects such as tables, charts, patterns and matrices.

Thank You!

<u>Thank You</u>

1. Thanks for watching the lecture video.