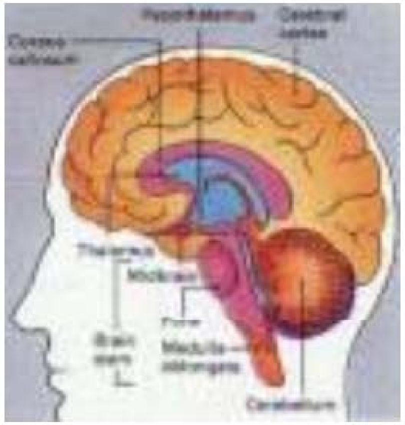
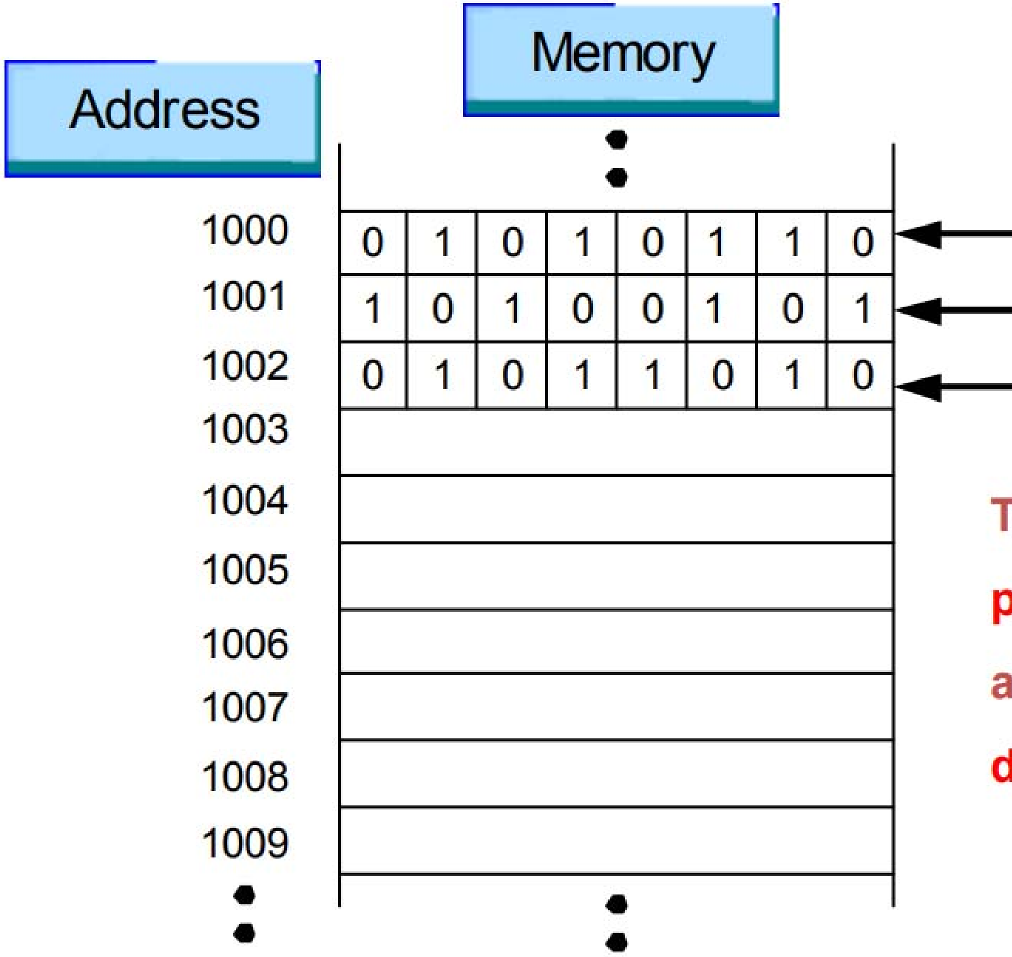
CS100
Introduction to Programming

Lecture 2. Data Types, Memory, Simple Input/Output & Control Flow

I. Memory & Data Types

Computer Memory





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	
1004	
1005	
1006	
1007	
1008	
1009	
⋮	⋮

Memory Cells

Content of memory

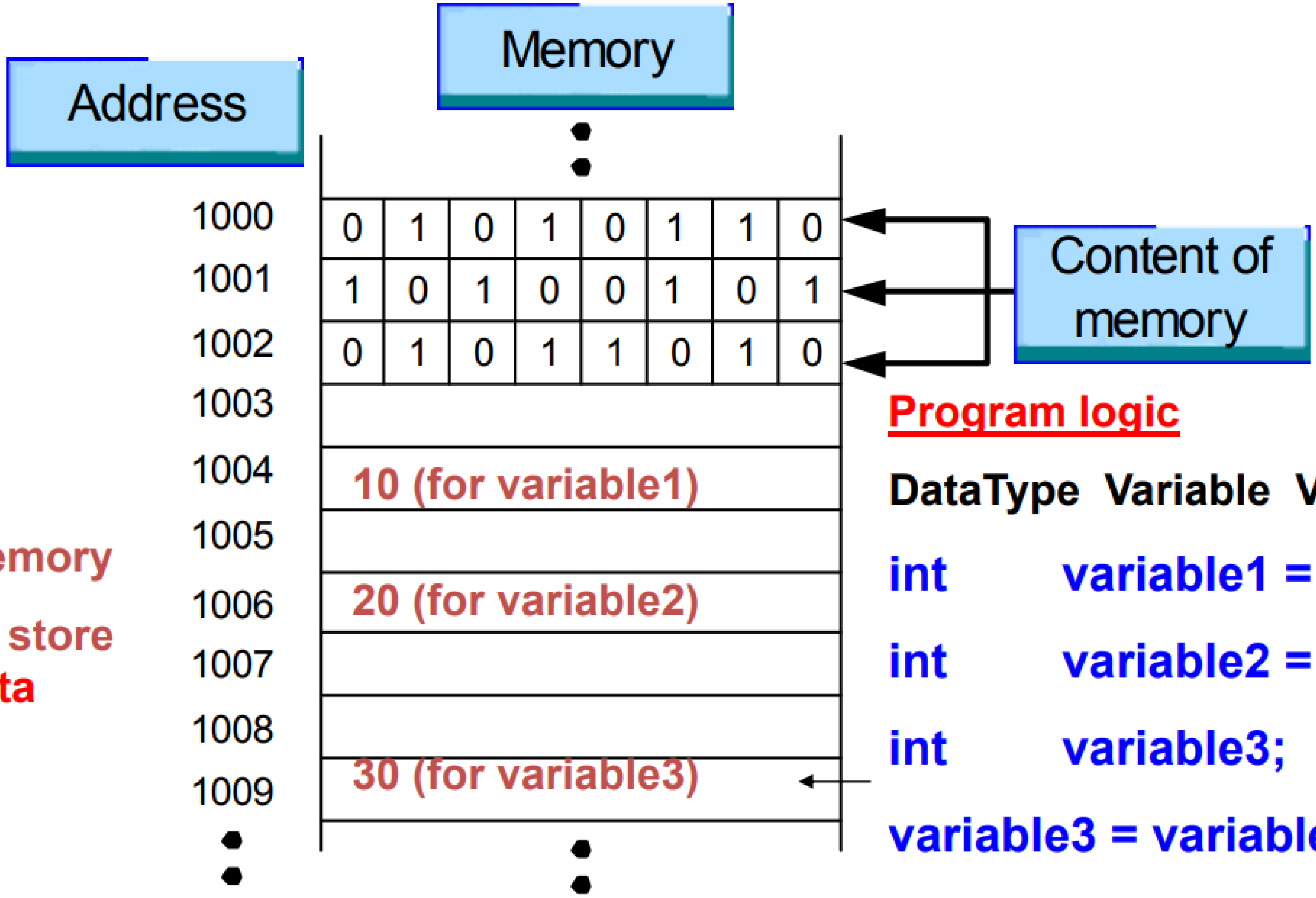
To store

program instructions

and

data

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	
1004	10 (for variable1)
1005	
1006	20 (for variable2)
1007	
1008	
1009	30 (for variable3)

Memory

To store data

Program logic

DataType Variable Value

```
int        variable1 = 10;
int        variable2 = 20;
int        variable3;
variable3 = variable1 + variable2;
....
```

Data Types

- It determines the kind of data that a variable can hold, how many **memory** cells are reserved for it and the operations that can be performed on it.

- **Integers**
  - short (2 bytes – 16 bits)
  - **int** (2 bytes)
  - long 32 bits (4 bytes)
  - unsigned (2 bytes)
  - unsigned short (2 bytes)
  - unsigned long 32 bits (4 bytes)
- **Floating Points**
  - **float** (4 byte, or 32 bits)
  - **double** (8 bytes, or 64 bits)

- **Characters**
  - 128 distinct characters in the ASCII character set.
  - Two C character types:
    - **char** (1 byte or 8 bits, range: [–128, 127]
    - unsigned char (1 byte or 8 bits, range: [0, 255]

Data Types

- The amount of **memory** used for objects of these types is machine dependent.

- The **range** of the values allowed for each type depends on the number of bits used

- Choose the type whose range is **just enough** to cover all the possible values of the object, for **space efficiency**.

Literals

- Literals (constant values) are fixed **values** (associated with data type) used in the program.

- Four types of literals:
  - **Integer** literals, e.g. 100, −256
  - **Floating-point** literals, e.g. 2.4, −3.0
  - **Character** literals, e.g. ‘a’, ‘+’
  - **String** literals, e.g. “Hello World”

Variables

- A **variable** is a name given to the memory cell(s) where the computer uses to store data.
- A variable’s **name** allows the program to refer to the variable.
- It is a good practice to follow the naming convention.
- The following 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		

Variable Declaration

- To use a variable, you must first declare the variable.
- A variable declaration always contains 2 components:
  - its **data type** (e.g. short, int, long, etc.)
  - its **name** (e.g. count, numOfSeats, etc.)
- Syntax for variable declaration:

  < **data type** > < **name** >

- Below are some examples of variable declarations:

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

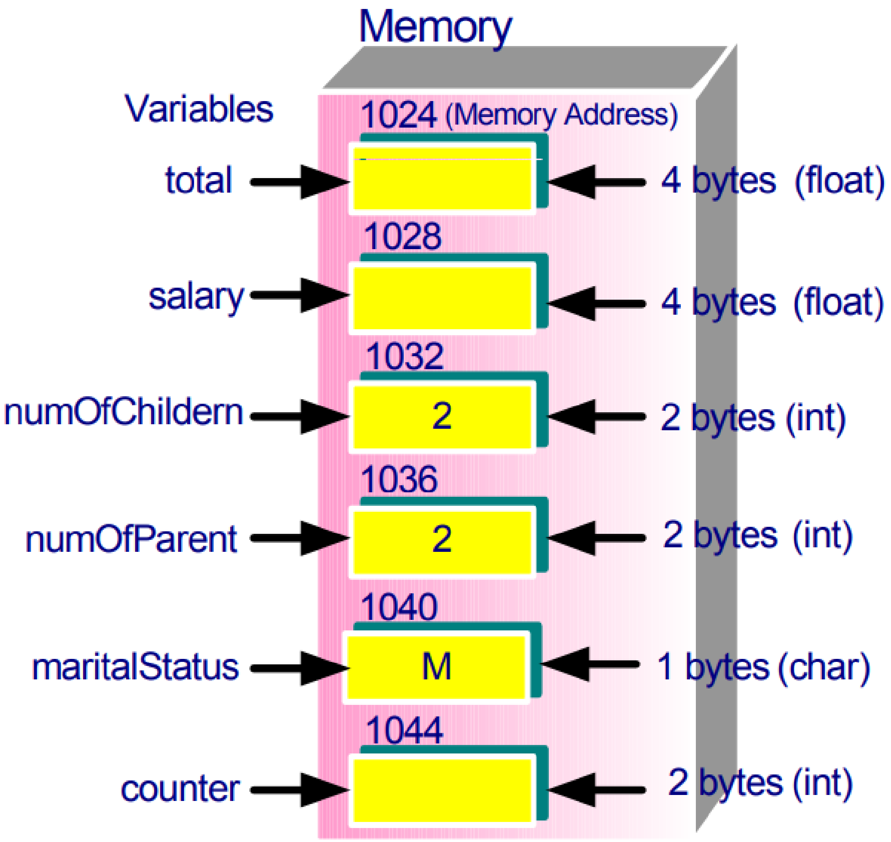
- Below are some examples of variable initializations:

```
int count = 20;
float temperature, result;
temperature = 36.9;
```

Declaring Variables with Initialization

- **Example**

```
int main()
{
    float total, salary;
    int numOfChildren = 2;
        numOfParents = 2;
    char maritalStatus = ‘M’;
    int counter;


    ......
    return 0;
}
```



- In this example, total and salary are declared without initial values and the other variables are declared with initial values.

II. Expressions

Operators

- Arithmetic operators: +, −, *, /, %
  - E.g. 7/3 (= 2); 7%3 (= 1); 6.6/2.0 (=3.3); etc.
- Assignment operators:
  - E.g. float amount = 25.50;
- Chained assignment:
  - E.g. a = b = c = 3;
- Arithmetic assignment operators: +=, −=, *=, /=, %=
  - E.g. a += 5 (meaning a = a + 5).
- Relational operators: ==, !=, <, <=, >, >=
  - E.g. 7 >= 5 (this returns TRUE).
- Incremental / decremental operators: ++, −−
  - E.g. a++ (means a = a + 1); b−− (means b = b – 1).

Increment/decrement Operators

- **increment operator**: ++ can be used in two ways, prefix and postfix modes. In both forms, the variable will be incremented by 1.
- In prefix mode: ++varName

  (1) varName is incremented by 1 and

  (2) the value of the expression is the updated value of varName.
- In postfix mode: varName++

  (1) The value of the expression is the current value of varName and

  (2) then varName is incremented by 1.
- The way the **decrement operator** '––' works is the same as the "++", except that the variable is decremented by 1.

Increment/decrement Operators

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

   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);
   printf(“value of num is %d\n”, num);
   return 0;
}
```

Output:

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


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

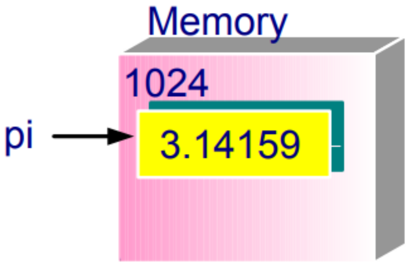
Constants

- A constant is an object whose value is **unchanged** throughout the life of the program.
- There are **three** ways to define a constant:

1) directly give the value

```
print(“p = %f.\n”, 3.14159);
/* 3.14159 is a floating point constant */
```

2) define a constant variable



format: **const type varName = value**

where

type: int, float, char, etc.

varName: name of the constant variable

```
const float pi = 3.14159;
/* declare a float constant variable pi with value
3.14159 */
printf(“p = %f.\n”, pi);
```

Constants

3) use the preprocessor directive #define

Format: **#define** **constantName** **value**

where constantName is name of the constant.

(constantName should use ***upper*** case).

```
#include <studio.h>
#define TAX_RATE 0.12 //define a constant TAXRATE with value 0.12
int main()
{
    float income1, income2, tax;
    tax = income1 * TAX_RATE; //substituted by 0.12
    tax = tax + income2 * TAX_RATE; //substituted by 0.12
    return 0;
}
```

- During compilation, the value of the constant will be **substituted** whenever the name of the constant appears in the program
- By giving a name to a constant,
  - it improves the readability of the program
  - it makes programs easier to be modified

Expressions

- An **expression** is any combination of variables, constants and operators that can be evaluated to yield a result.
  – Examples: a+b; count++; (item1 + item2) * tax_rate; speed = distance/time;

- You can tell the compiler explicitly how you want an expression to be evaluated by using **parentheses** **(** and **)** .
  – Note: (1 + 2 * 3) is different from ((1 + 2) * 3)

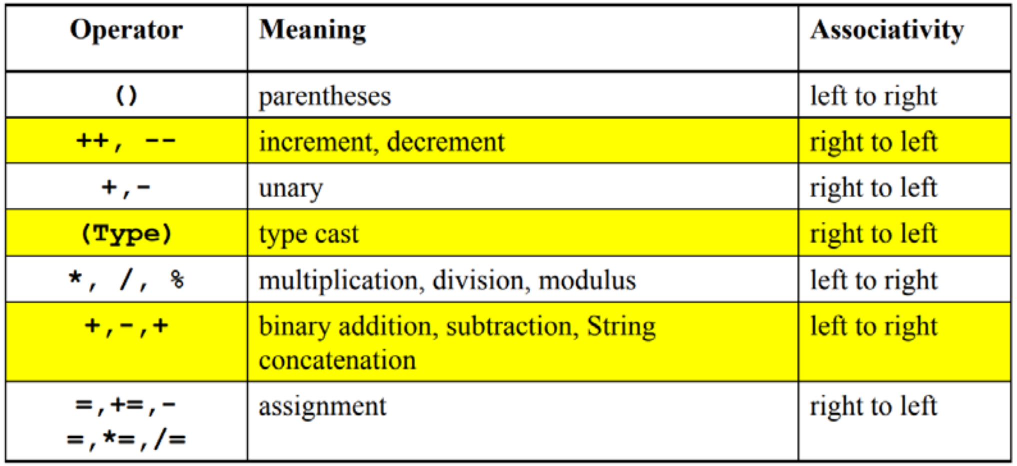
- To make your code easier to read and maintain, you should be explicit and indicate with parentheses whenever possible.

Operator Precedence

- The expression is evaluated according to the priority of the operator

Higher priority

Lower priority

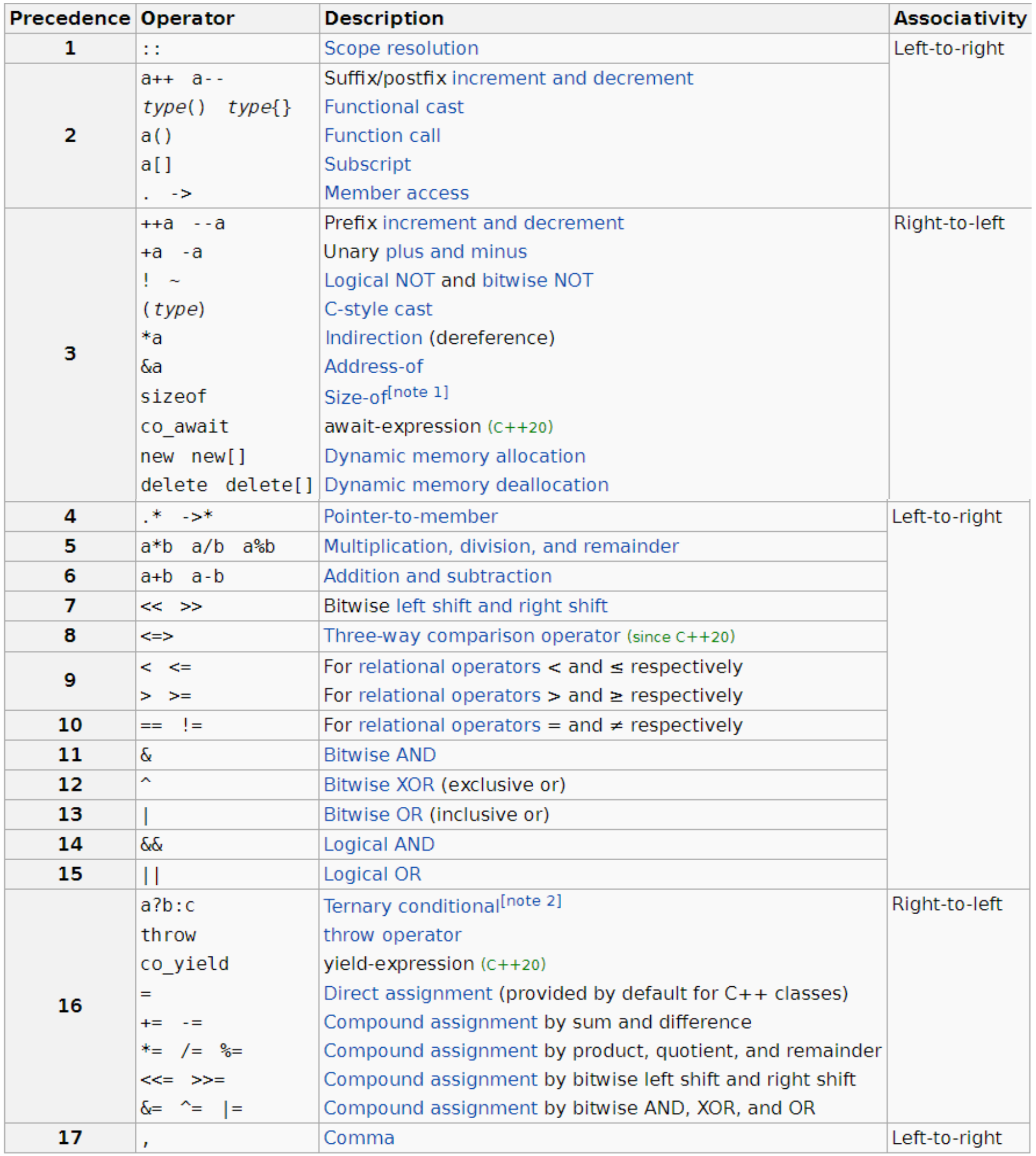


Operator	Meaning	Associativity
()	parentheses	left to right
++, --	increment, decrement	right to left
+,-	unary	right to left
(Type)	type cast	right to left
*, /, %	multiplication, division, modulus	left to right
+,-,+	binary addition, subtraction, String concatenation	left to right
=,+=,- =,*=,/=	assignment	right to left

- Higher priority should be evaluated first

X = a+(a-b*b++)/c

Full List of Operators with Precedence



Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a-- *type*() *type*{} a() a[] . ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access	
3	++a --a +a -a ! ~ (*type*) *a &a sizeof co_await new new[] delete delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of[note 1] await-expression (C++20) Dynamic memory allocation Dynamic memory deallocation	Right-to-left
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	<=>	Three-way comparison operator (since C++20)	
9	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
10	== !=	For relational operators = and ≠ respectively	
11	&	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13	|	Bitwise OR (inclusive or)	
14	&&	Logical AND	
15	||	Logical OR	
16	a?b:c throw co_yield = += -= *= /= %= <<= >>= &= ^= |=	Ternary conditional[note 2] throw operator yield-expression (C++20) Direct assignment (provided by default for C++ classes) Compound assignment by sum and difference Compound assignment by product, quotient, and remainder Compound assignment by bitwise left shift and right shift Compound assignment by bitwise AND, XOR, and OR	Right-to-left
17	,	Comma	Left-to-right

Data Type Conversion

Arithmetic operations require two numbers in an expression/assignment are of the same type.

There are three kinds of conversions :

1. Explicit conversion: uses the type casting operators, i.e. (int), (float), …, etc.

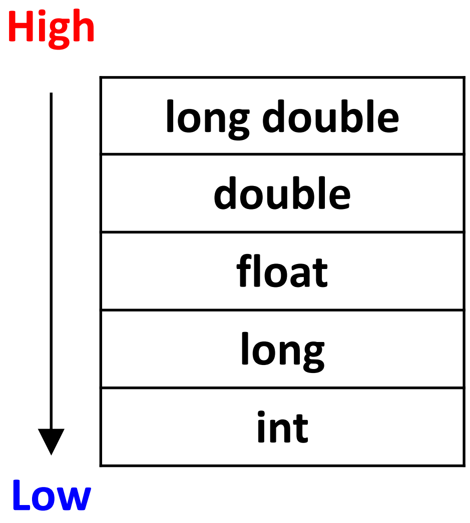
- e.g. `(int)2.7 + (int)3.5`

2. Arithmetic conversion: in mix operation it converts the operands to be type of the **higher** ranking of the two

- e.g. `2 + 3.5; // convert to float`

3. Assignment conversion: converts the type of result of computing the expression to that of the type of the left hand side if they are different:

- e.g. `num = 2.7 + 3.5; // num is int`


High

long double

double

float

long

int

Low

Data Type Conversion

```
#include <stdio.h>
int main(){
   int num;
   /* Explicit Conversion */
   num = (int)2.7 + (int)3.5;
   /* convert 2.7 to 2 and 3.5 to 3
   then do addition */
   printf(“num = %d\n”, num);

   /* Assignment Conversion */
   num = 2.7 + 3.5;
   /* add 2.7 and 3.5 to get 6.2, then
   convert it to 6 */
   printf(“num = %d\n”, num);

   /* Arithmetic Conversion */
   /* converts 2 to 2.0 then do
   addition */
   printf(“num = %f\n”, 2 + 3.5);
   return 0;
}
```

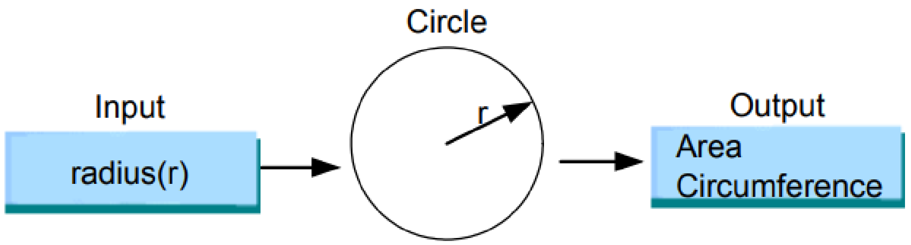
Output

```
num = 5
num = 6
num = 5.500000
```

Possible ***pitfalls*** of data type conversion -

Loss of precision: e.g. from **float** to **int**, the fractional part is lost.

A C Program Example



Area = π*r*r

Cirumference= 2*π*r

```
#include <stdio.h>
int main()
{
   const float PI = 3.14;
   float radius, area, circumference;
   // Read the radius of the circle
   printf("Enter the radius: ");
   scanf("%f", &radius);
   // Calculate the area
   area = PI * radius * radius;
   // Calculate the circumference
   circumference = 2 * PI * radius;
   // Print the area and circumference of the circle
   printf("The area is %0.1f\n", area);
   printf("The circumference is %0.1f", circumference);
   return 0;
}
```

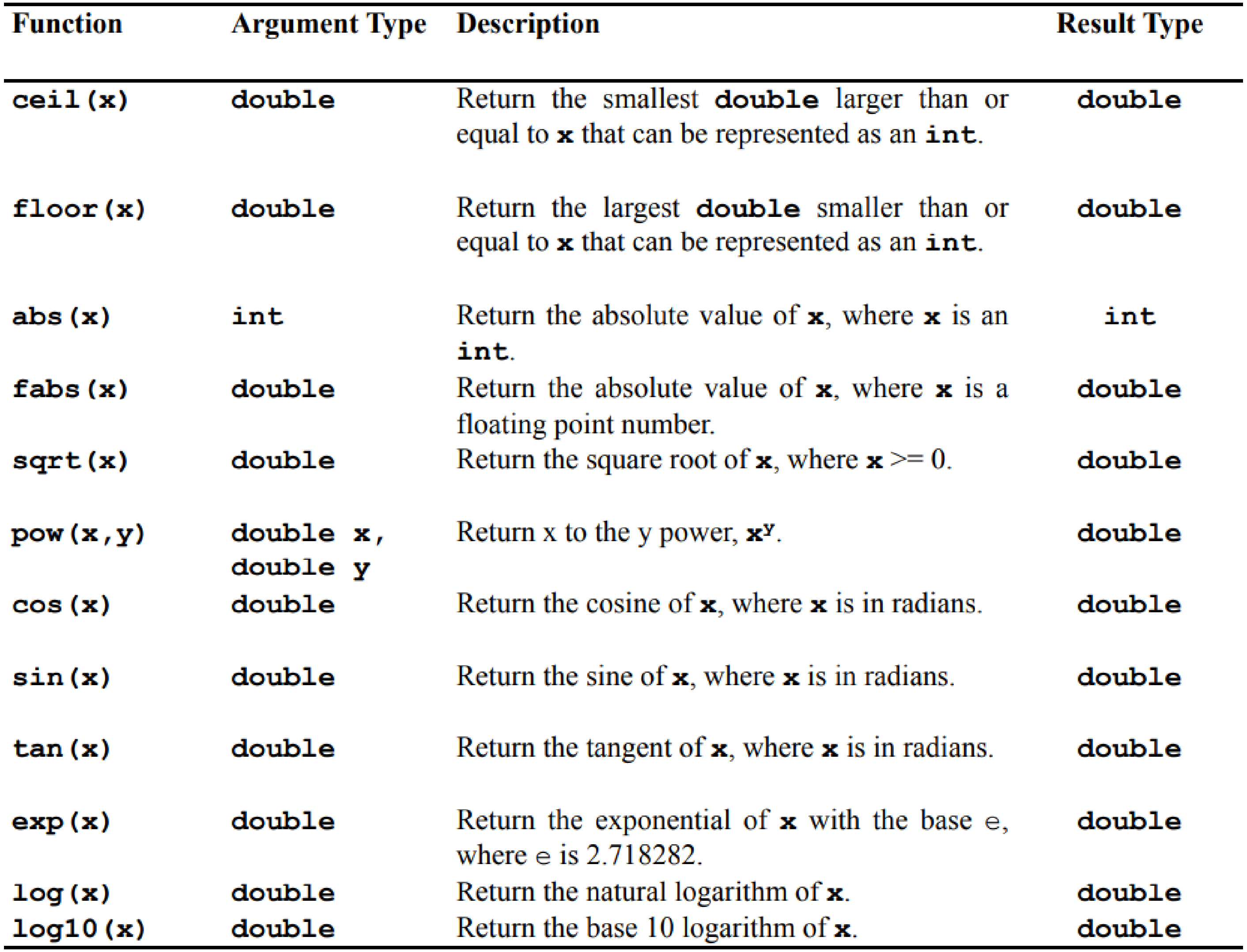
In C:

Output function: printf()

Input function: scanf()

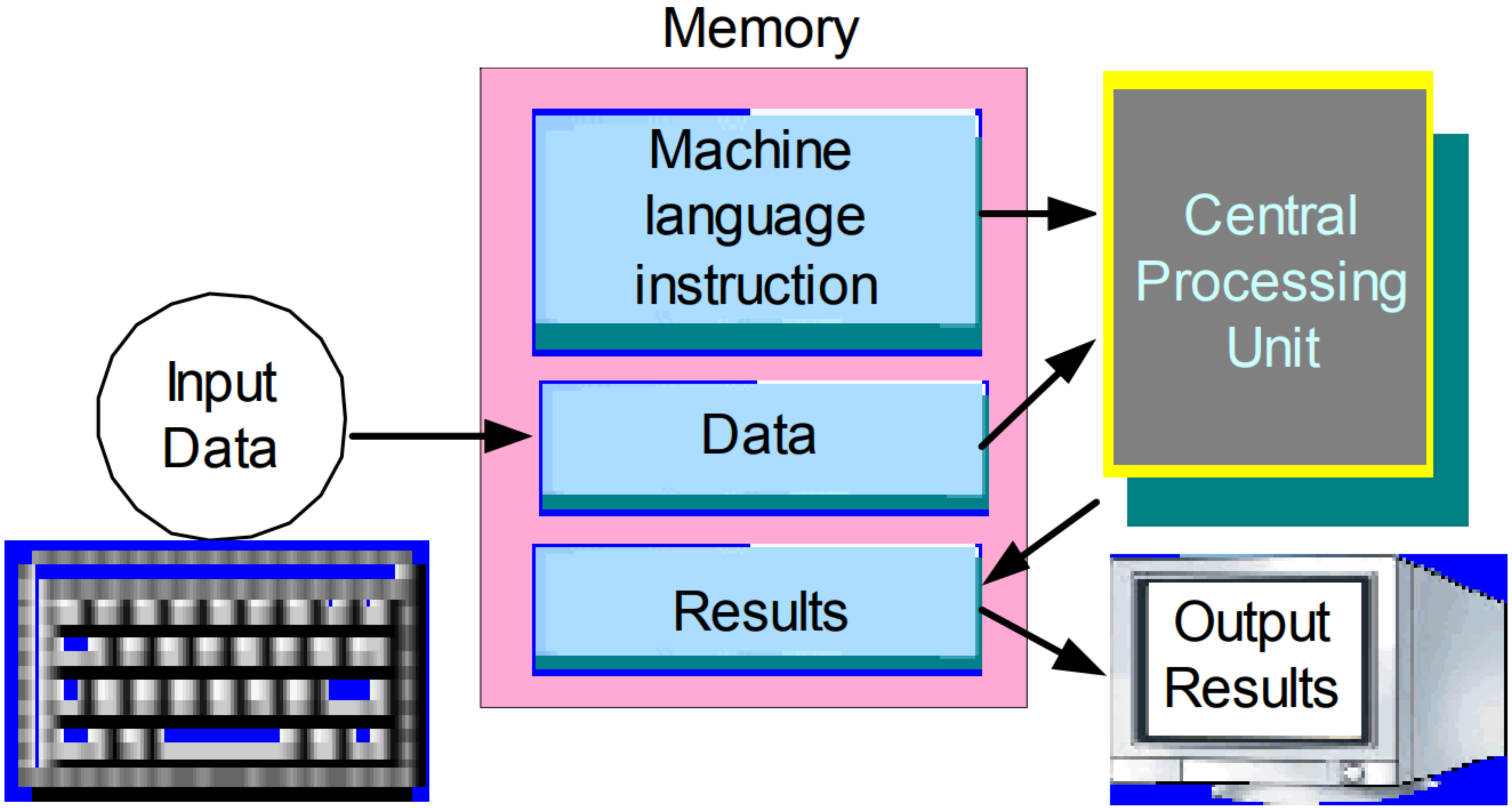
Mathematical Library Functions

#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**

Executing Programs

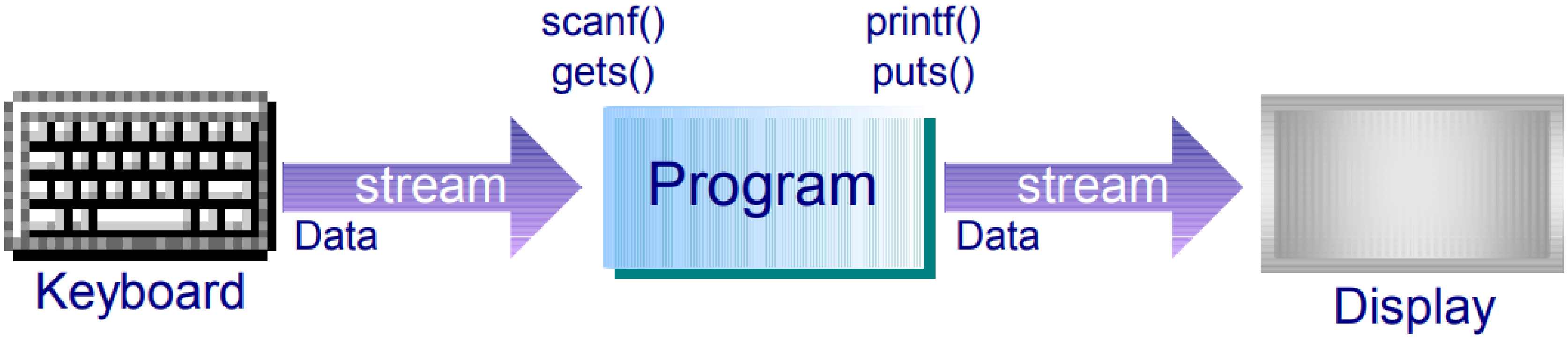


III. Formatted Input/Output

Formatted Input/Output

- **Input/output (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 I/O libraries.

- Input from the keyboard / output to the monitor screen is referred to as **standard input/output**.



I/O Functions

- A function is a piece of code to perform a specific task.
- A library contains a group of functions, usually for related tasks, e.g. standard I/O functions are in the library <stdio.h>, maths functions in the library <math.h>

- To use the I/O functions in <stdio>, the line

  #include <stdio.h>

  need be included as the preprocessor instructions in a program

- Two I/O functions are used most frequently:
  - **printf(…)** : output function
  - **scanf(…)** : input function

Simple Output: printf()

- The printf() statement has the form:

  printf(control-string, argument-list);

- The **control-string** is a string constant. It is printed on the screen.
  - %?? is a conversion specification. An item will be substituted for it in the printed output.

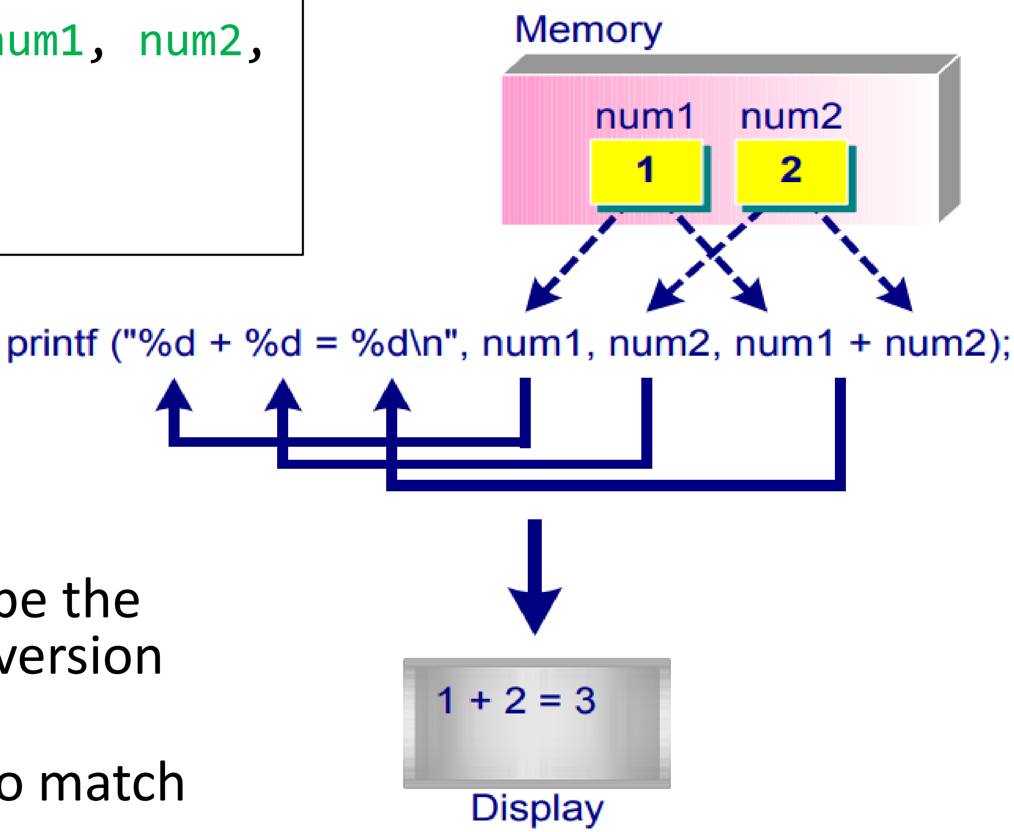
- The **argument-list** contains a list of items such as item1, item2, …, etc.
  - Values are to be substituted into places held by the conversion specification in the control string.
  - An item can be a constant, a variable or an expression like num1 + num2.

printf() – Example 1

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

Output:

```
1 + 2 = 3
```



Note:

- The number of items must be the same as the number of conversion specifiers.
- The types of items must also match the conversion specifiers.

printf() – Conversion Specification

Type of *Conversion Specifiers*

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

printf() – Example 2

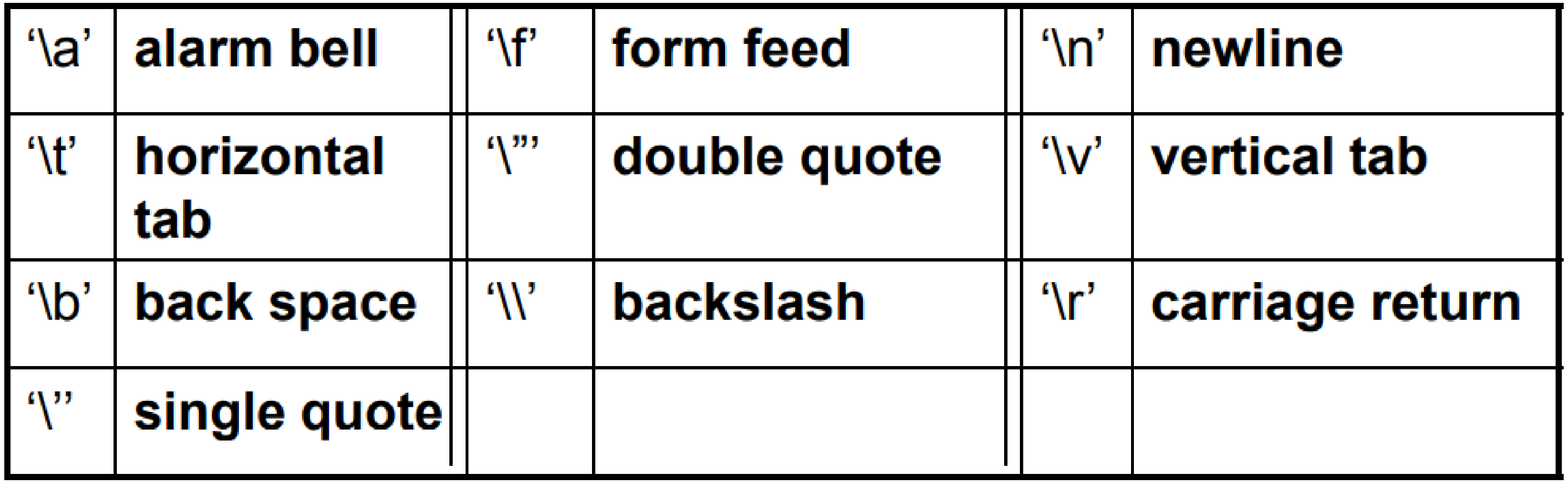
```
#include <stdio.h>
int main()
{
  int num = 10;
  float i = 10.3;
  double j = 100.0;

  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 */

  return 0;
}
```

Output:

```
int num = 10
float i = 10.300000
double j = 100.000000
```

Examples of Escape Sequence

- Some useful **non-printable control characters** are referred to by the ***escape sequence*** which is a better alternative, in terms of memorization, than numbers, e.g. **‘\n’** the newline (or linefeed) character instead of the number **10**.



‘\a’	**alarm bell**	‘\f’	**form feed**	‘\n’	**newline**
‘\t’	**horizontal tab**	‘\”’	**double quote**	‘\v’	**vertical tab**
‘\b’	**back space**	‘\\’	**backslash**	‘\r’	**carriage return**
‘\’’	**single quote**				

General Structure of Conversion Specification for Formatted Output

- A conversion specification is of the form

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

  - ***%*** and ***conversionSpecifier*** are compulsory. The others are optional.
  - We will focus on using **%** and **conversionSpecifier** for printing integers, floating point numbers and strings.
  - Students should refer to the reference book or web materials for other options of formatted output.

Printing Integer Values

	Conversion Specification	Flag	Field Width	Conversion Specifier	Output on Screen
(1)	**%d**	none	none	**d**	125
(2)	**%+6d**	+	6	**d**	□□+125
(3)	**%-6d**	–	6	**d**	125□□□

- A ***flag*** is used to control the display of plus or minus sign of a number, and left or right justification.
  - The **+ flag** is used to print values with a plus sign "+" if positive, and a minus sign "–" otherwise.
  - The **– flag** is used to print values left-justified.
- The ***minimum field width*** gives the lower bound of the field width to be used during printing (padding with blanks or zeros if the item is less wide than it)

Printing Floating-Point Values

	Conversion Specification	**Flag**	**Field Width**	**Precision**	**Conversion Specifier**	**Output on Screen**
(1)	**%f**	none	none	none	f	10.345689
(2)	**%+11.5f**	+	11	5	f	□□+10.34568
(3)	**%−11.5f**	−	11	5	f	10.34568□□□
(4)	**%+12.3e**	+	12	3	e	□□+1.034e+01
(5)	**%−12.3e**	−	12	3	e	1.034e+01□□□

- The ***precision*** field can be used for printing floating-point numbers. The precision field specifies **the number of digits after the decimal point** to be printed.

Simple Input: scanf()

- A scanf() statement has the form:

```
scanf(control-string, argument-list);
```

- The **control-string** is a string constant containing conversion specifications.
- The **argument-list** contains 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 & (“ampersand”) sign. This is to tell scanf() the address of the variable so that scanf() can read the input value and store it in the variable.
- 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 1

```
#include <stdio.h>

int main()
{
   int n1, n2;
   printf(“Please enter 2 integers:\n”);
   scanf(“%d %d”, &n1, &n2);
   printf(“The sum = %d\n”, n1 + n2);
   return 0;
}
```

Output:

```
Please enter 2 integers:
5 10
The sum = 15
```

scanf() – Example 2

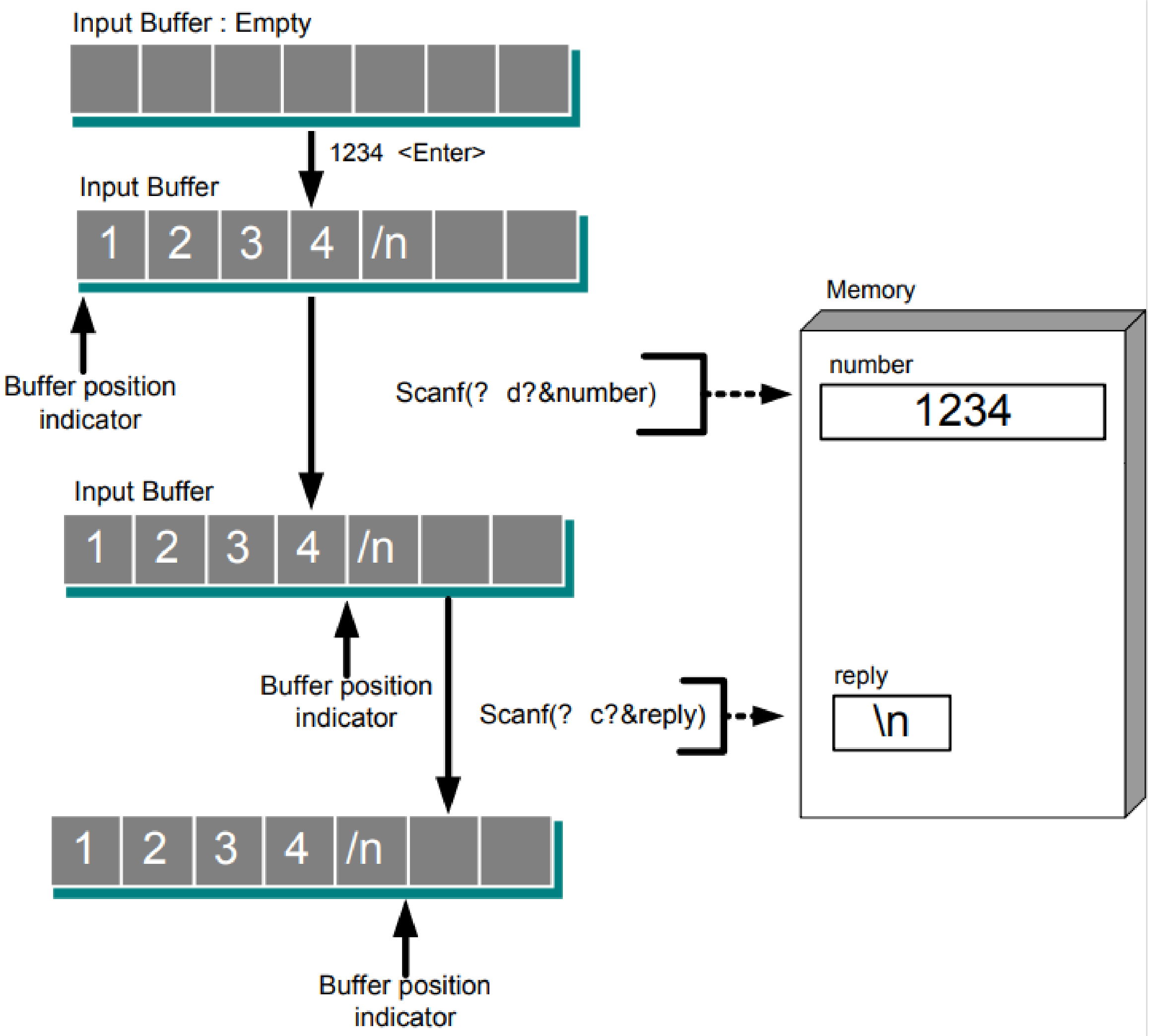
```
#include <stdio.h>
int main()
{
   int number;
   printf(“Please enter a number:”);
   scanf(“%d”, &number);
   printf(“The number read is %d\n”, number);
   // read in a char
   char reply;
   printf(“Correct(y/n)?”);
   scanf(“%c”, &reply);
   printf(“your reply: %c\n”, reply); // display char
   return 0;
}
```

Output:

```
Please enter a number: 1234<Enter>
The number read is 1234
Correct(y/n)? your reply:
```

an error here

scanf() – Example 2



Reason:

There is a hidden character ‘\n’ entered when you type *1234<Enter>*

scanf() – Example 2

- **Solution 1:**

```
     ...
  fflush(stdin); // flush the input buffer with newline
  printf(“Correct(y/n)?”);
  scanf(“%c”, &reply);
  printf(“your reply: %c\n”, reply);
     ...
```

- **Solution 2:**

```
     ...
  printf(“Correct(y/n)?”);
  scanf(“\n%c”, &reply); // read the newline
  printf(“your reply: %c\n”, reply);
     ...
```

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() need process the control string for formatting. Also, if an error occurs, **putchar()** returns either the integer value of the written character or EOF.

getchar()

- The syntax of calling getchar() is

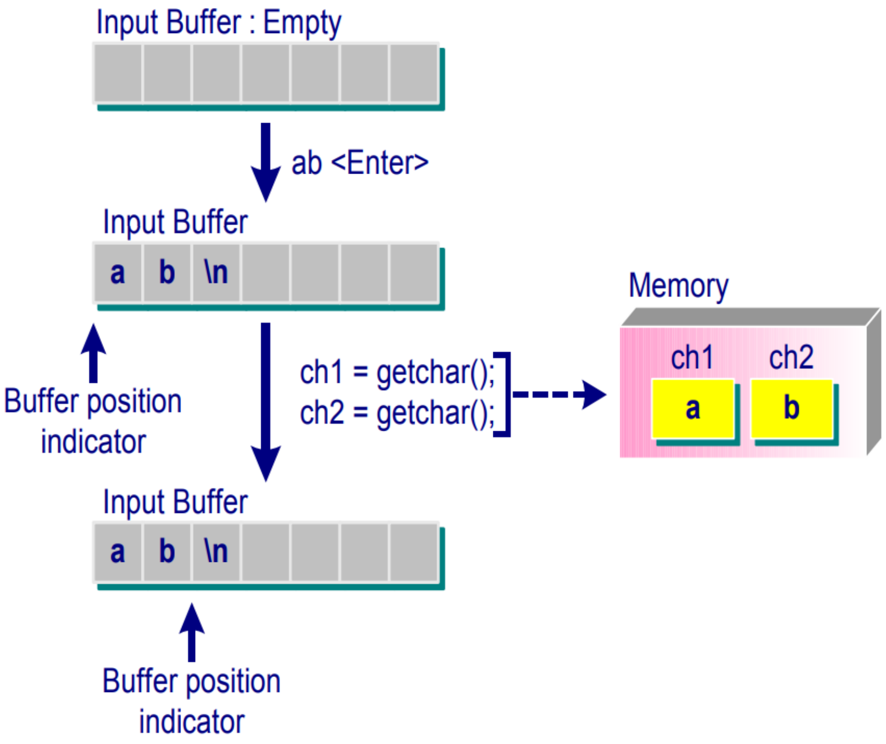
  ch = getchar(); // ch is a character variable.

- It is equivalent to

  scanf(“%c”, &ch);

Character Input/Output - Example

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



Output:

```
1a
1a
ab            (User Input)
ab
```

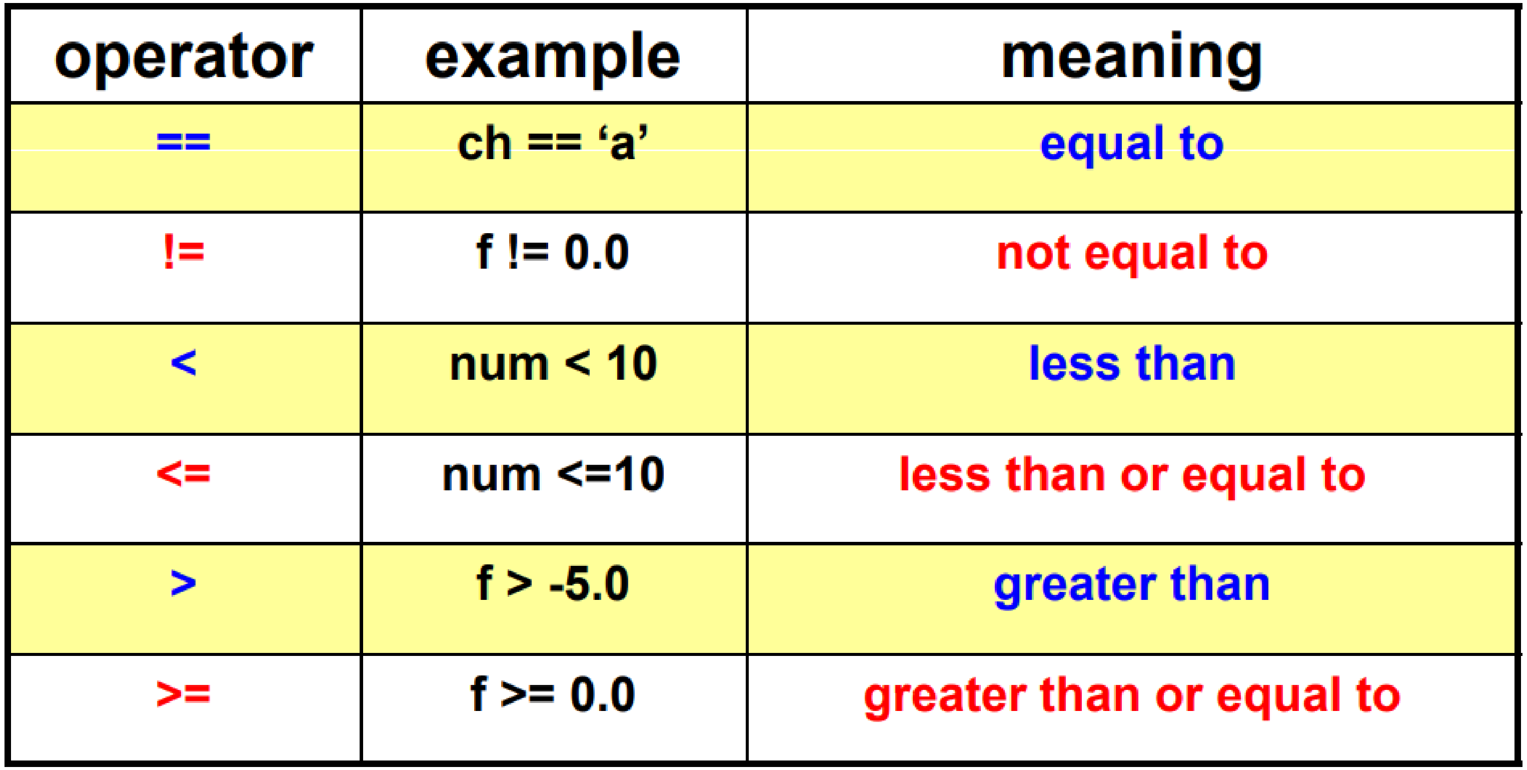
IV. Logical Operations

Relational Operators

Used for **comparison** between **two values**.

Return **Boolean** result: **true** or **false**.

Relational Operators:

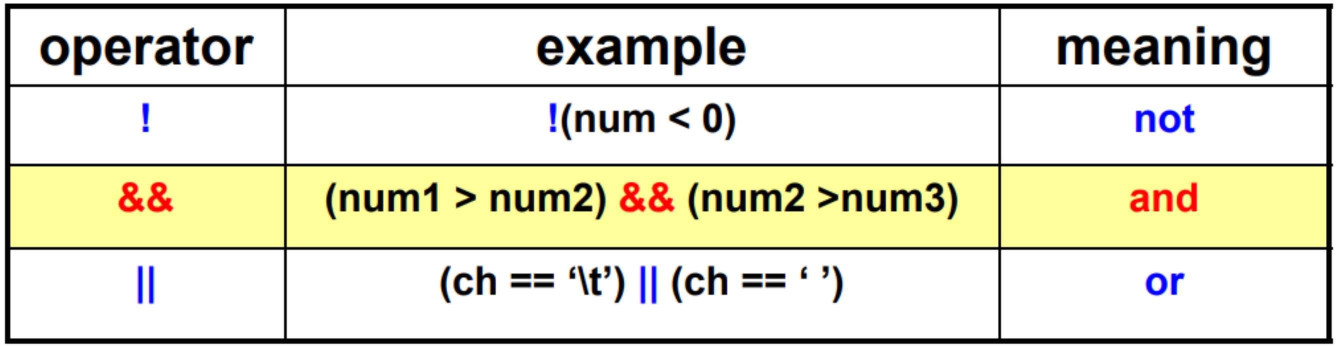


operator	example	meaning
==	ch == ‘a’	equal to
!=	f != 0.0	not equal to
<	num < 10	less than
<=	num <=10	less than or equal to
>	f > -5.0	greater than
>=	f >= 0.0	greater than or equal to

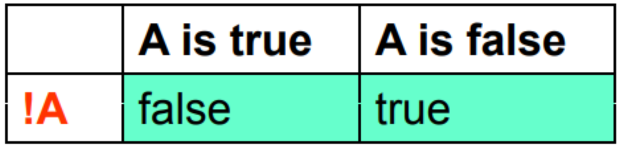
Logical Operators

- Work on one or more relational expressions to yield a logical value: **true** or **false**.
- Allow testing and combining the results of comparison expressions.

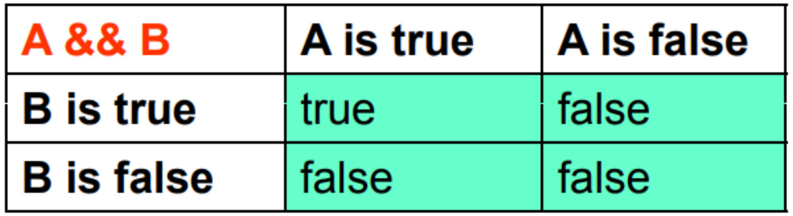
Logical Operators:



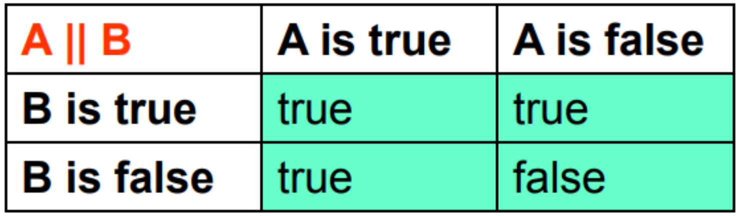
operator	example	meaning
!	!(num < 0)	not
&&	(num1 > num2) && (num2 >num3)	and
||	(ch == ‘\t’) || (ch == ‘ ’)	or



	A is true	A is false
!A	false	true



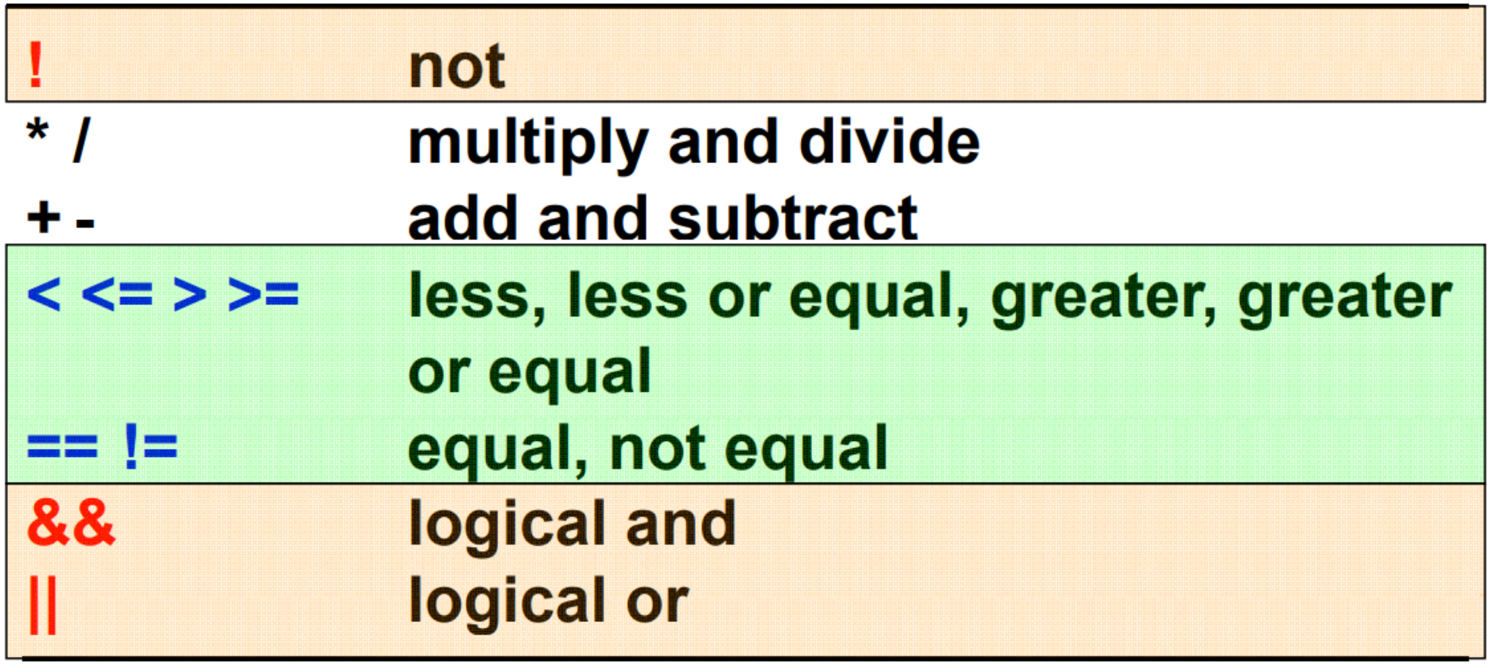
A && B	A is true	A is false
B is true	true	false
B is false	false	false



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

Precedence of operators

- 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

- **Example**: The expression `!(5 >= 3)||(7 > 3)` is true, where the **logical or operator ||** is executed in the end.

Boolean Result

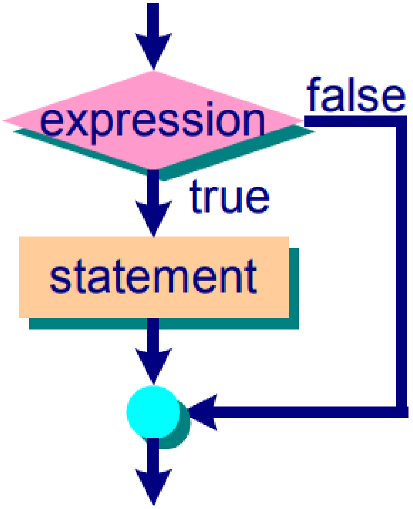
- The result of evaluating an expression involving relational and/or logical operators is
  - either **true** or **false**
  - either **1** or **0**
  - When the result is **true**, it is **1**. Otherwise, it is **0**. That is, the C language uses 0 to represent a false condition.
- In general, **any integer expression whose value is non-zero is considered true**; otherwise it is **false**.
- Examples:

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

The if Statement

if (expression)

> **statement;**
>
> /* simple or compound statement enclosed with brackets */



```
#include <stdio.h>
int main(void)
{
   int num;  /* value supplied by user. */
   printf(“Give an integer from 1 to 10: ”);
   scanf(“%d”, &num);
   if (num > 5)
      printf(“Your number is larger than 5.\n”);
   printf(“%d is the number you entered.\n”, num);
   return 0;
}
```

Output 1:

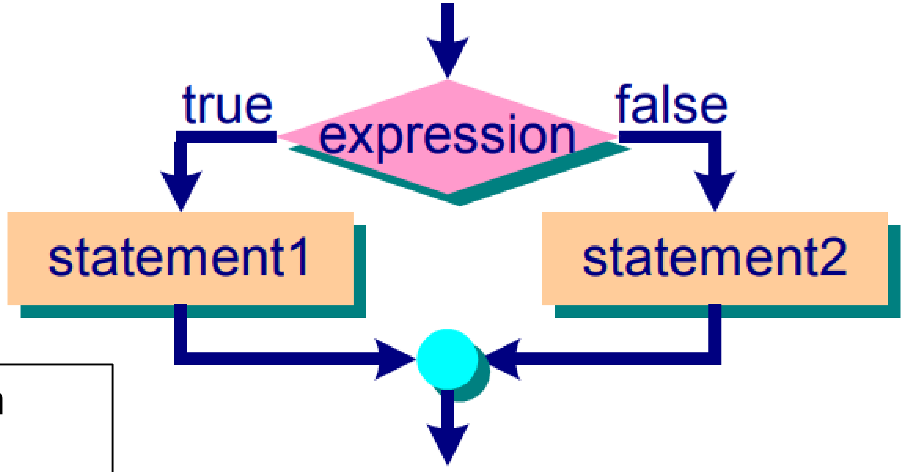
```
Give an integer from 1 to 10: 3
3 is the number you entered.
```

Output 2:

```
Give an integer from 1 to 10: 7
Your number is larger than  5.
7 is the number you entered.
```

The if-else Statement

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



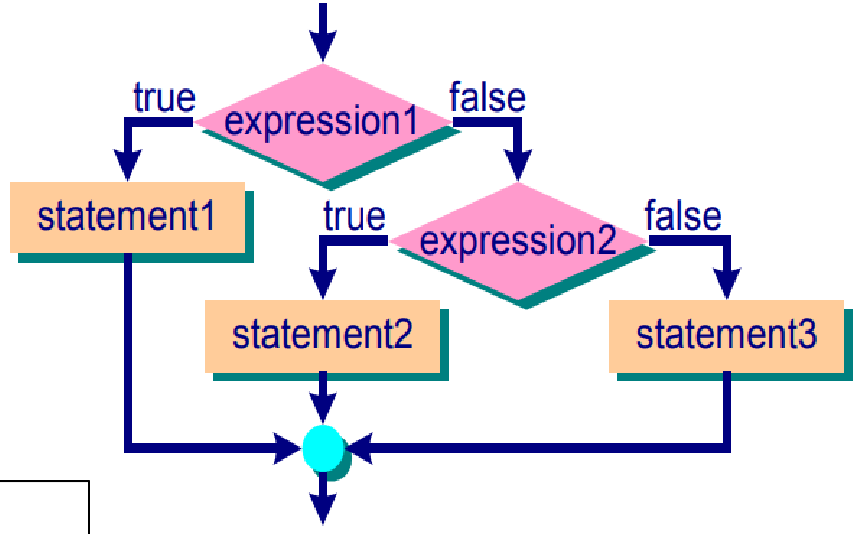
```
/* This program computes the maximum
value of num1 and num2 */
#include <stdio.h>
int main(void)
{
   int num1, num2, max;
   printf(“Please enter two integers:”);
   scanf(“%d %d”, &num1, &num2);
   if (num1 > num2)
      max = num1;
   else
      max = num2;
   printf(“The maximum of the two \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

The if…else if…else Statement

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



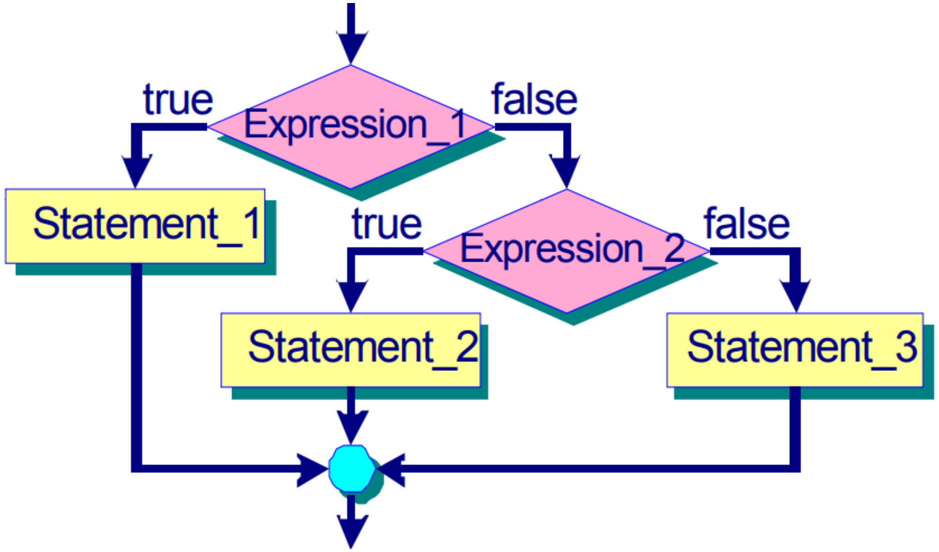
```
#include <stdio.h>
int main(void)
{
   float temp;  // temperature reading.
   printf(“Temperature reading:”);
   scanf(“%f”, &temp);
   if (temp >= 100.00 && temp <= 120.0)
      printf(“Temperature OK.\n”);
   else if (temp < 100.0)
      printf(“Temperature too low.\n”);
   else
      printf(“Temperature too high.\n”);
   return 0;
}
```

Output:

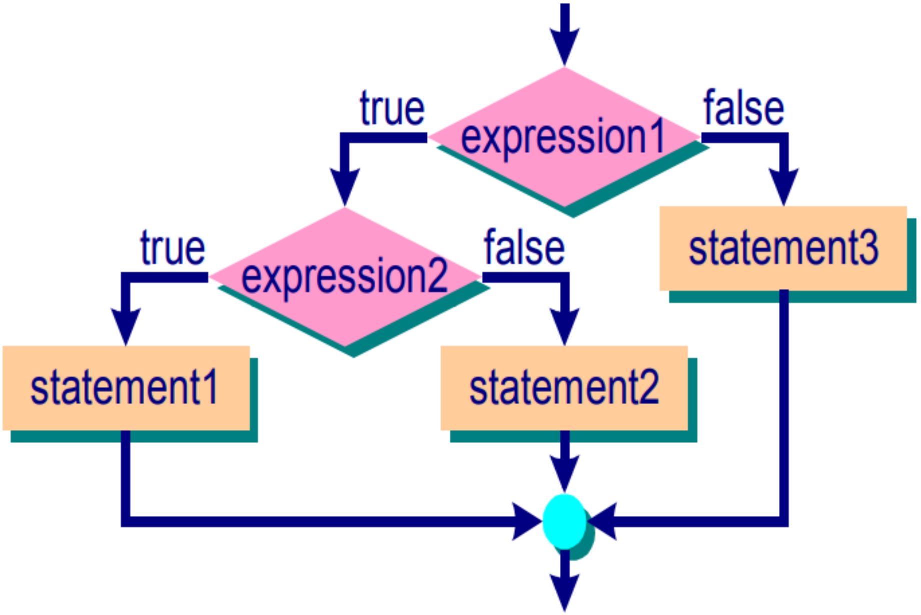
```
Temperature reading: 105.0
Temperature OK.

Temperature reading: 130.0
Temperature too high.
```

Nested-if

```
if (expression 1)
      statement1;
else
      if (expression2)
            statement2;
      else
            statement3;
```

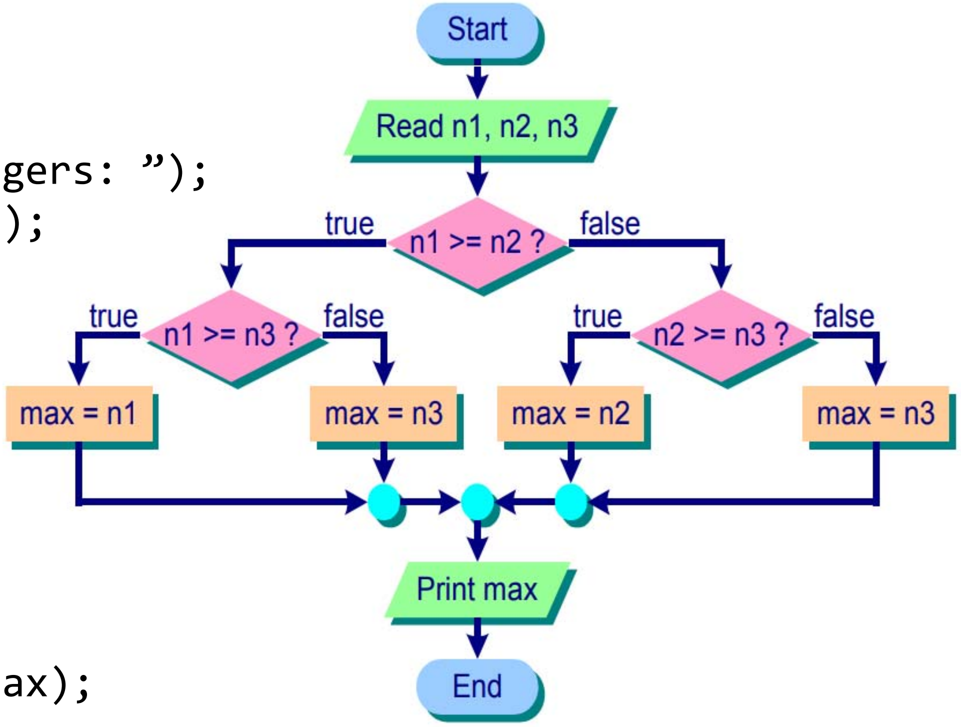


```
if (expression 1)
         if (expression2)
               statement1;
         else
               statement2;
else
      statement3;
```



Nested-if Example

```
/* This program computes the maximum value of
three numbers */
#include <stdio.h>
int main(void)
{
   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;
}
```



Output:

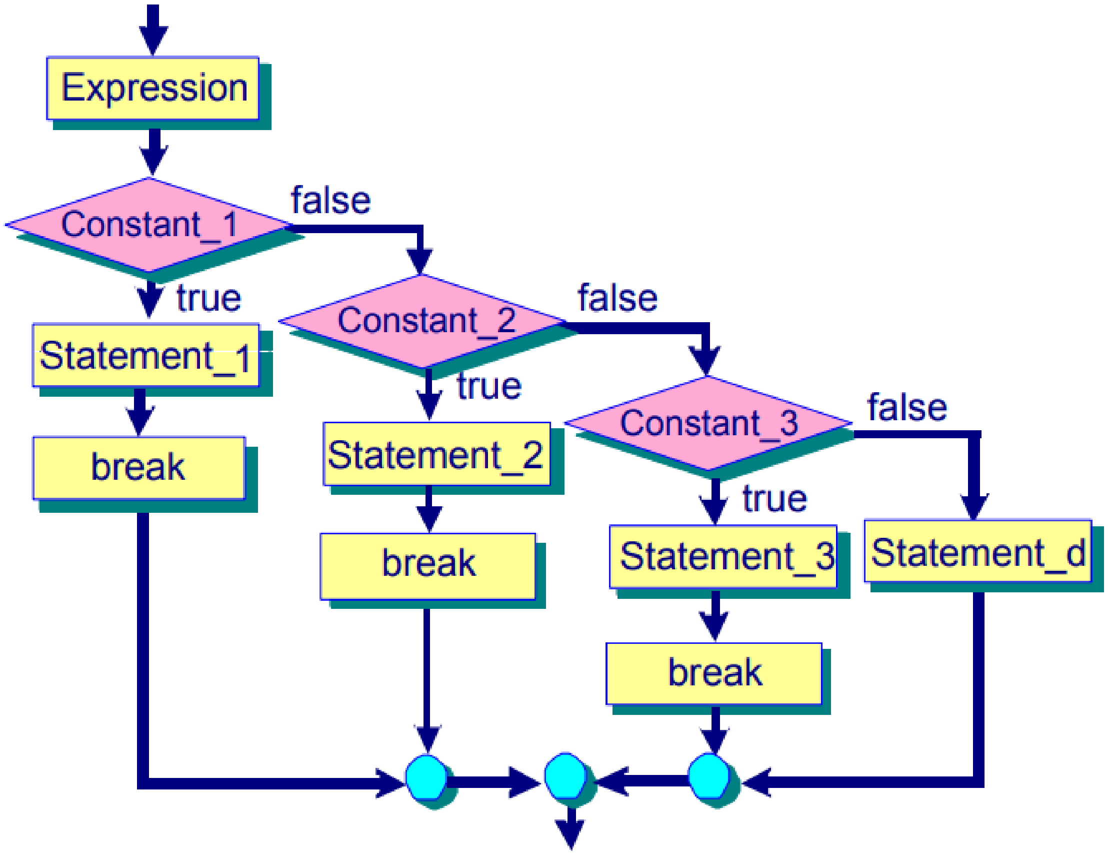
Please enter three integers: ***1 2 3***

The maximum of the three is 3

The switch Statement

The **switch** is 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;
}
```



The switch Statement

- ***switch***, ***case***, ***break*** and ***default*** are reserved words.
- The result of ***Expression*** in ( ) must be an **integral type**.
- *Constant_1*, *Constant_2*, … are called **labels**. Each must be an integer constant, a character constant or an integer constant expression, e.g. 3, ‘A’, 4+‘b’, 5+7, etc.
- Each of the labels *Constant_1*, *Constant_2*, … must deliver a **unique integer value**. Duplicates are not allowed.
- We may also have **multiple labels** for a statement, for example, to allow both the lower and upper case selection.
- If we do not use **break** after some statements in the switch statement, execution will continue with the statements for the subsequent labels until a break statement or the end of the switch statement. This is called the **fall through** situation.

switch: Example

```
#include <stdio.h>
main(void) {
   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(“num1 + num2 = %d\n”, result);
         break;
      case ‘s’:
      case ‘S’: result = num1 – num2;
         printf(“num1 – num2 = %d\n”, result);
         break;
      case ‘m’:
      case ‘M’: result = num1 * num2;
         printf(“num1 * num2 = %d\n”, result);
         break;
      default: printf(“Not a proper choice.\n”);
   }
   return 0;
}
```

Output:

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

A switch Example: Converting Score to Grade

Weighted Average Score S	Grade
90 <= S	A
80 <= S < 90	B
70 <= S < 80	C
60 <= S < 70	D
50 <= S < 60	E
S < 50	F

```
switch ((int)averageScore/10) {
  case 10: case 9:
    grade = ‘A’; break;
  case 8:
    grade = ‘B’; break;
  case 7:
    grade = ‘C’; break;
  case 6:
    grade = ‘D’; break;
  case 5:
    grade = ‘E’; break;
  default: grade = ‘F’;
}
```

The 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

=> value of the expression is that of Expression_2

else

=> value of the expression is that of Expression_3

- **Example**:

```
max = (x > y) ? x : y;
```

<==>

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

Conditional Operator: Example

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

   selection ? printf(“A one.\n”) : printf(“A zero.\n”);
   return 0;
}
```

Output:

```
Enter a 1 or a 0 => 1
A one.
Enter a1 or a0 => 0
A zero.
```

V. Looping

Repetition: Loops

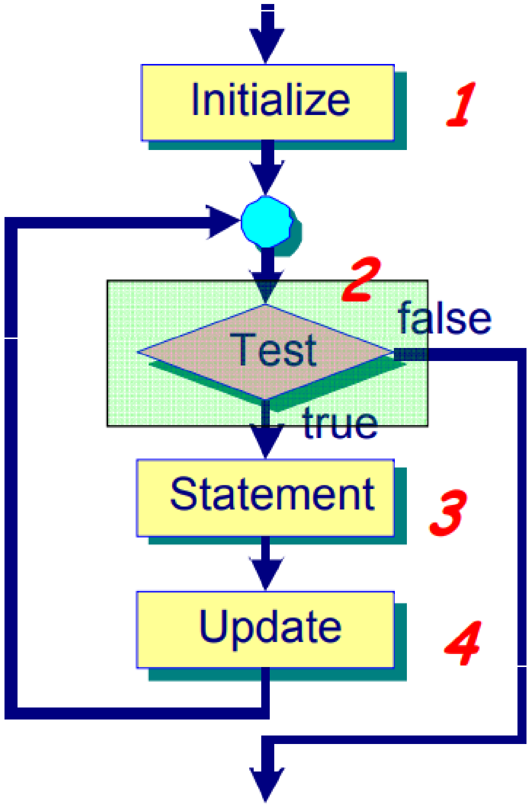
Some sections of statements are executed repeatedly. These repetitions are referred to as **loops**. There are two types of loops:

- **Counter-controlled loops**: The loop body is repeated for a number of times, and *the number of repetitions is known* before the loop starts execution.
- **Sentinel-controlled loops**: *The number of repetitions is not known* before the loop starts execution. Usually, a **sentinel value** (such as −1, different from regular data) is used to determine whether to execute the loop body.

Looping

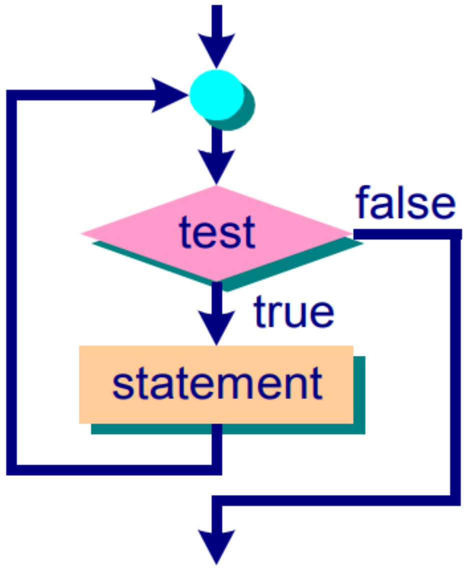
To construct loops, we need:

1. **Initialization** – initialize the ***loop control variable***.
2. **Test condition** – evaluate the test condition (involve ***loop control variable***).
3. **Loop body** – if test is true, the loop body is executed.
4. **Update** – typically, ***loop control variable*** is modified through the execution of the loop body. It can then go through the **test** condition again.



The while Loop

```
while (test)
        statement
```



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

Output:

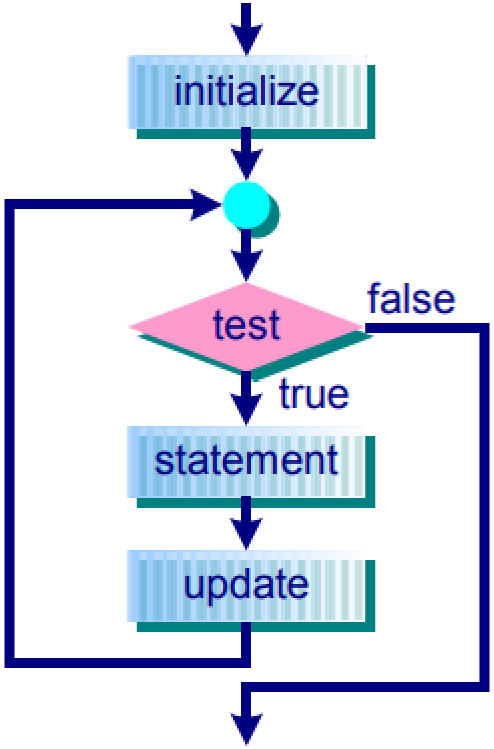
```
Enter the list of integers:
1 8 11 24 36 48 67 -1
The sum is 195

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

The for Loop

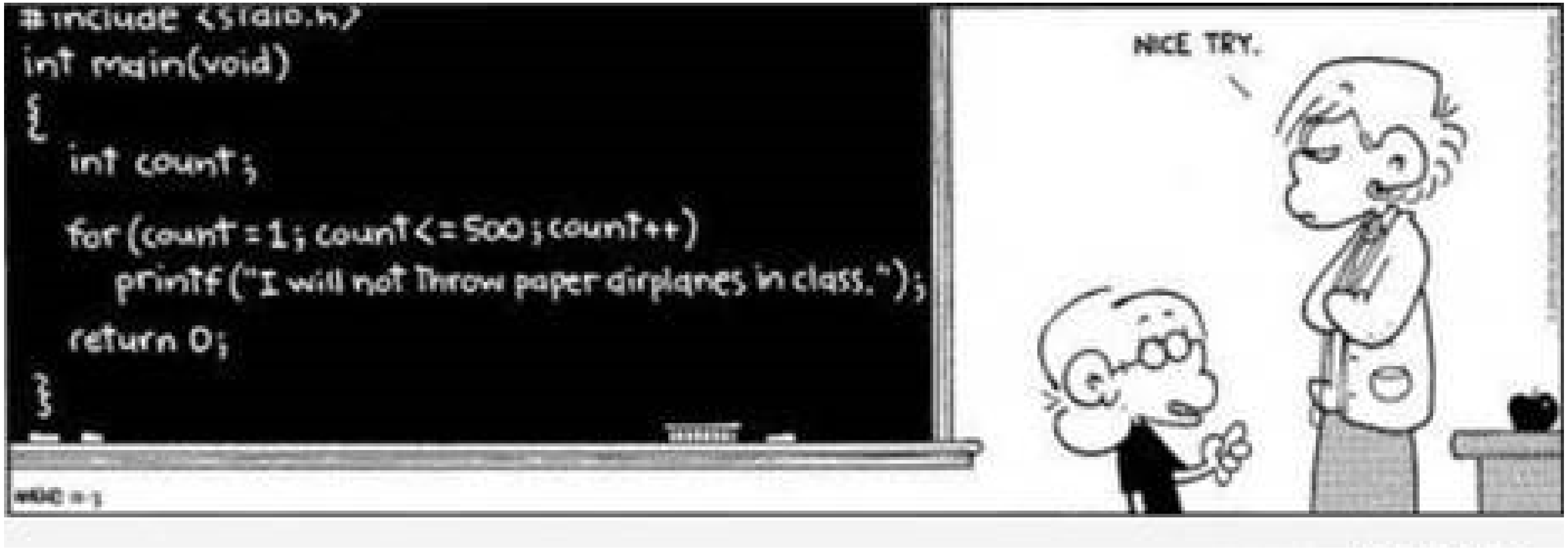
```
for (initialize; test; update)
    statement;
```

- Normally, *test* is a relational expression to control iterations.
- *update* is frequently used to update some loop control variable before repeating the loop.
- Any or all of the 3 expressions may be omitted. In case test is missing, it becomes an *infinite loop*, i.e. all statements inside the loop will be executed again and again. For example:

```
for (;;) {   /* an infinite loop */
    statement1;
    …
}
```



for Loop: Example 0



```
#include <stdio.h>
int main(void)
{
   int count;

   for (count = 1; count <= 500; count++)
       printf("I will not Throw paper airplanes in class.");

   return 0;
}
```

for Loop: Example 1

```
/* Display the distance a body falls in
feet/sec for the first n seconds; n is input
by user. */
#include <stdio.h>
#define ACCELERATION 32.0
main()
{
   int timeLimit, t;
   /* Distance by a falling body */
   int distance;

   printf(“Enter the time limit(seconds): ”);
   scanf(“%d”, &timeLimit);
   for (t = 1; t <= timeLimit; t++) {
      distance = 0.5 * ACCELERATION * t * t;
      printf(“Dist after %d seconds is %d\
         feet.\n”, t, distance);
   }
   return 0;
}
```

Output:

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

Enter the time limit(seconds): 0
```

for Loop: Example 2

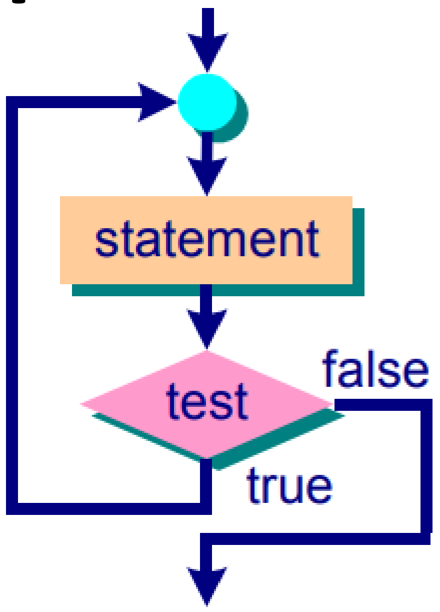
```
/* Display the distance a body falls every
5 seconds for the first n seconds; n is input
by user. */
#include <stdio.h>
#define ACCELERATION 32.0

main()
{
   int timeLimit, t;
   /* Distance by a falling body */
   int distance;

   printf("Enter the time limit(seconds): ");
   scanf("%d", &timeLimit);
   for (t = 5; t <= timeLimit; t += 5) {
      distance = 0.5 * ACCELERATION * t * t;
      printf("Dist after %d seconds is %d\
         feet.\n", t, distance);
   }
   return 0;
}
```

Output:

```
Enter the time limit(seconds): 15
Dist after 5 seconds is 400 feet.
Dist after 10 seconds is 1600 feet.
Dist after 15 seconds is 3600 feet.
```

The do-while Loop

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



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

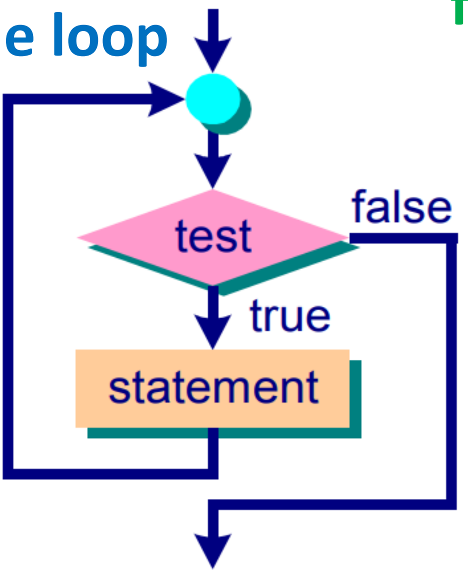
Output:

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

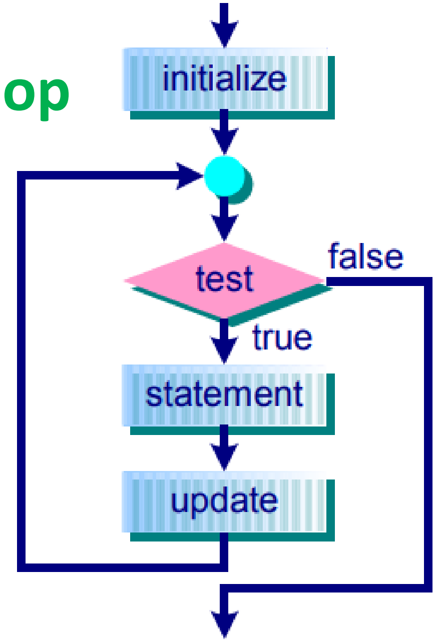
The do-while Loop

- Different from the ***while*** and ***for*** statements:
  - The condition ***test*** is performed **after** executing the statement every time.
  - This means the loop will be executed **at least once**.
  - Note: the *while* or *for* loop might **not** be executed **even once**.

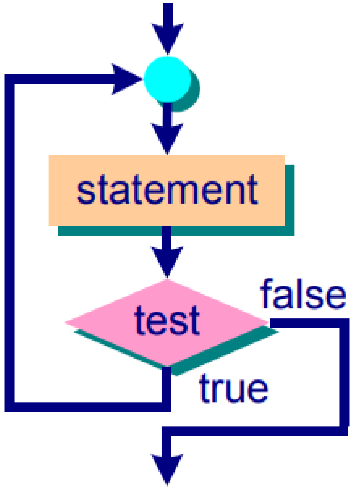
while loop



for loop



do-while loop



The break Statement

- To alter flow of control inside loop (and inside the switch statement). Execution of **break** causes <u>immediate termination</u> of the inner most enclosing loop or switch statement.

```
/* use break to exit a loop */
#include <stdio.h>
int main(void)
{
   float length, width;
   printf(“Enter rectangle length:\n”);
   while (scanf(“%f”, &length) == 1) {
      printf(“Enter its width:\n”);
      if (scanf(“%f”, &width) != 1)
         break;
      printf(“The area = %6.3f\n\n”,
         length * width);
      printf(“Enter rectangle length:\n”);
   }
   return 0;
}
```

Output:

```
Enter rectangle length:
2
Enter its width:
10
The area = 20.000

Enter rectangle length:
4
Enter its width:
a
```

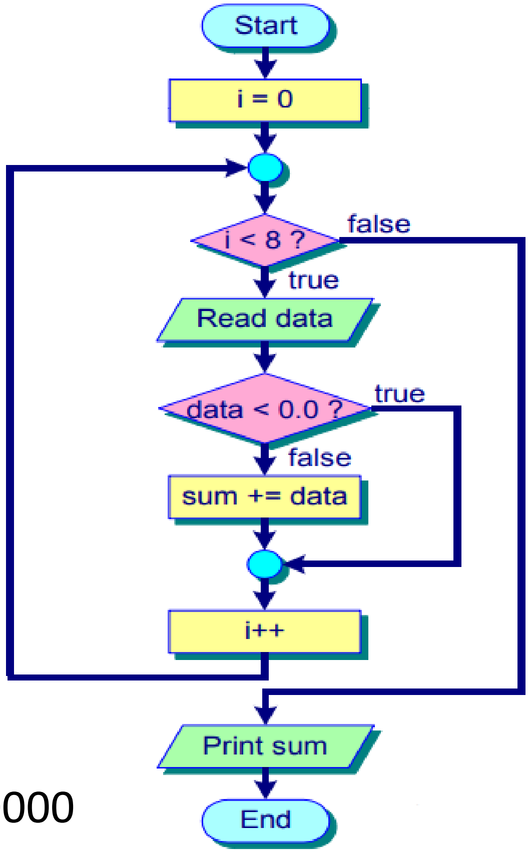
The continue Statement

- The control immediately passed to the (update and) 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(void)
{
   int i;
   float data, sum = 0.0;
   /* 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;
}
```

Output:

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



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.

```
/* count the number of different strings of a, b, c
*/
#include <stdio.h>
int main(void)
{
   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:

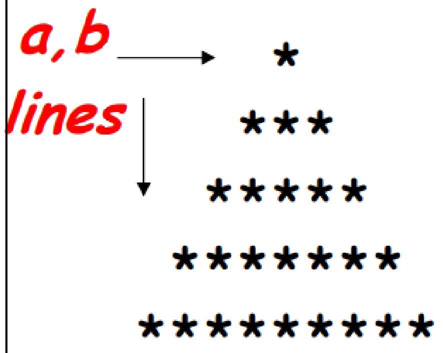
```
aa ab ac
ba bb bc
ca cb cc
9 different strings of letters.
```

Nested Loops: Example

```c
#include <stdio.h>
int main(void)
{
   int a, b, height, lines;

   printf("Enter the height of pattern: ");
   scanf("%d", &height);
   for (lines=1; lines <= height; lines++) {
      for (a=1; a <= (height – lines); a++)
         putchar(' '); // print blank space
      for (b=1; b <= (2*lines – 1); b++)
         putchar('*'); // print asterisk
      putchar('\n');
   }
   return 0;
}
```

Output:

Enter the height of pattern: 5



```
    *
   ***
  *****
 *******
*********
```