C Programming for Embedded Systems

By: **Dr Vahid Nabaei**

University of Glasgow
UESTC College

Learning outcomes and topics

- Programming Languages (C) for Embedded Systems
- C Programming Language

  Topic
  - Comment
  - Data Types
  - Operators
  - Conditional statements
  - Functions
  - Arrays
  - Structure and Pointers

A review on C programming for Embedded Systems

Online Sources

Online C Tutorials

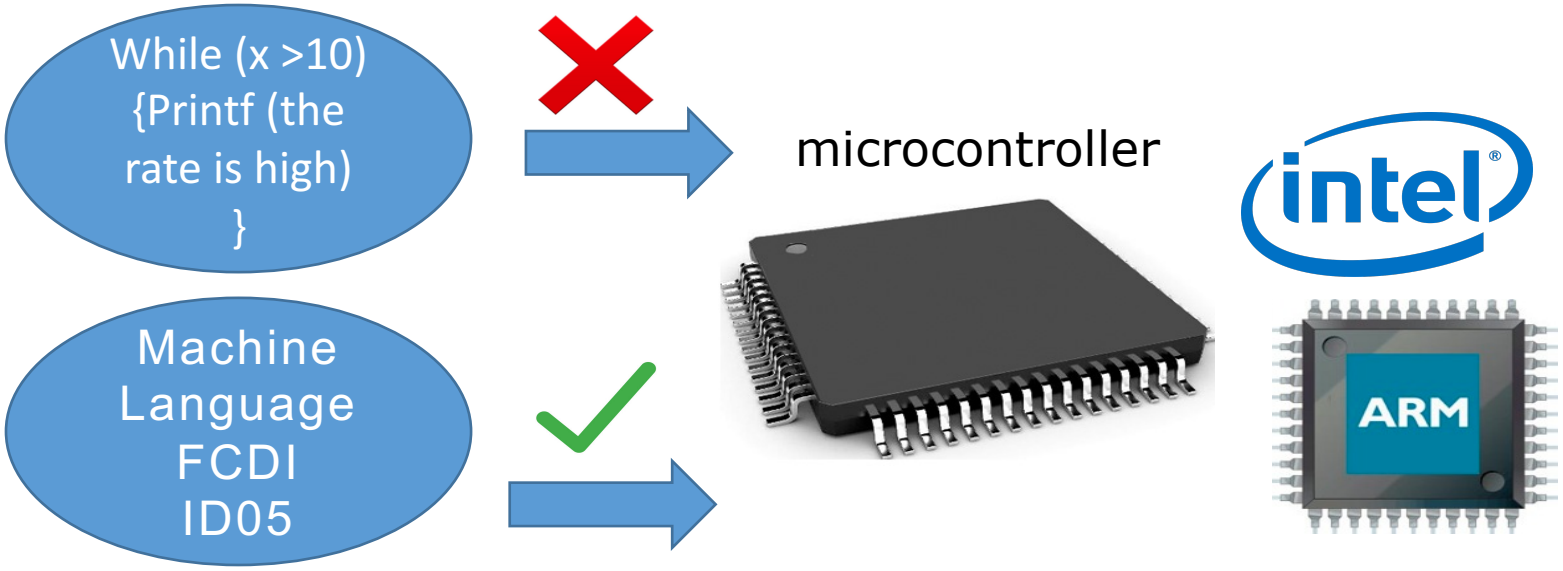
There are many free tutorials on C programming available online. Some that I found useful are:

- http://www.tutorialspoint.com/cprogramming/
- http://fresh2refresh.com/c-programming/
- http://www.cprogramming.com/tutorial/c/

Some of the slides come from these sites.

Programming Languages for Embedded Systems

- Program developed in C has to be processed first (compiled or interpreted).
- The microcontroller does not understand C, C++, Java, Python, or any other similar languages.
- The microcontroller understands its own machine language.



- We code in high-level languages, but it is not actually executed.
- High-level languages needs to be converted to machine languages execution.

Compilation vs Interpretation

❖ **Compilation**: The translation from high level language to the machine code (executable) just **ONCE** before running the code.

- This executable code can be run **every time** you run the program. For instance C, C++, and Java (partially)

- In Arduino, which is basically on C, you just **compile once** and never compile it again when you run the code. (unless a modification is required)

❖ **Interpretation**: The translation from high level language to the machine code is performed **at the time** you run the code.

❖ For instance, Basic, Visual Basic, Python languages are based on interpretation.

❖ The programmer does not have to deal with every details (easy to develop the program).

Why C Programming Language?

- High level language allows the programmer to develop the code faster and without knowing all the hardware details. But this reduce the code developer ability to use the hardware in a most effective way.

- **C has some features that make it attractive for Embedded Systems programing applications:**

- It allows the programmer to manipulate memory locations with very little overhead added.

- C also allows the code-developer to control I/O directly.

- C is faster than other languages (e.g. Python), because of compilation and interpretation differences.

- **Assembly language** can be used also, but it requires much more works, appropriate knowledge, and computer architecture.

C-Cross Compiler vs C-Compiler

- Compiler is a program developed to translate a high-level language source code into the machine code of the system of interest.

What does it mean the C-Cross compilation?

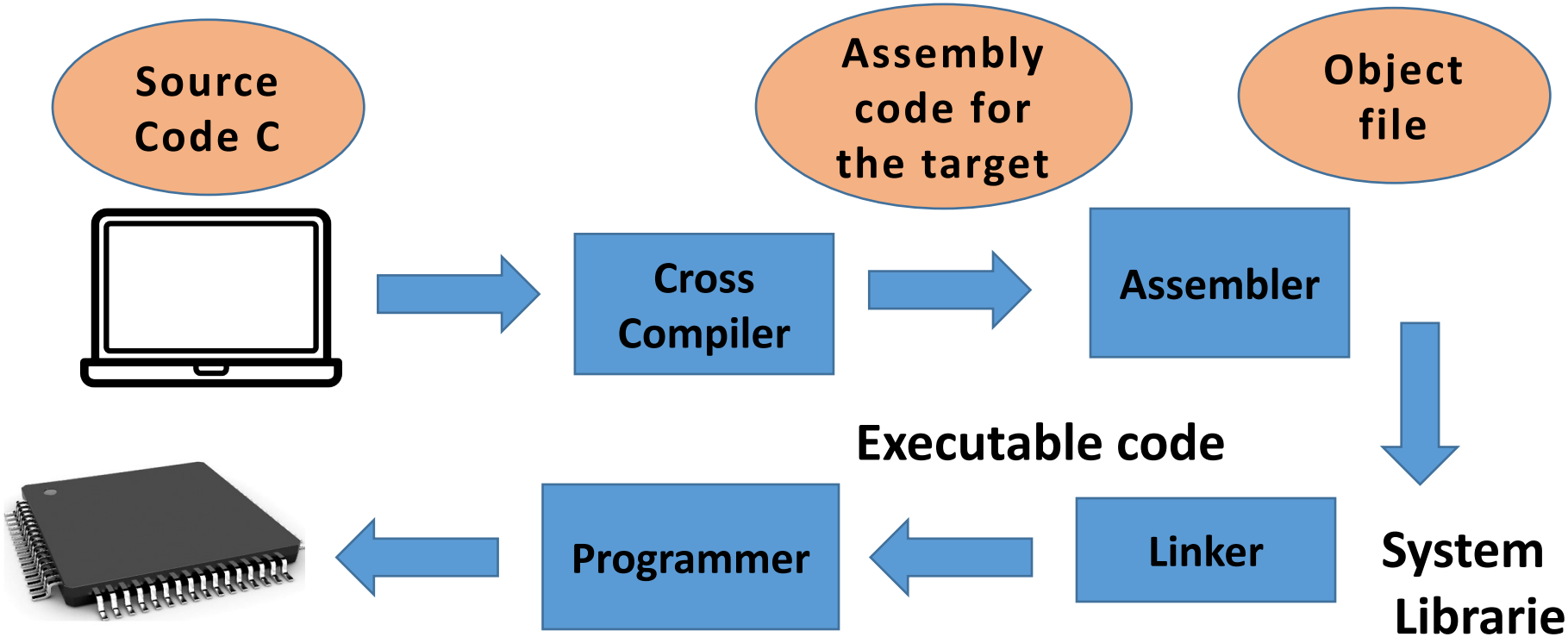
- It is the compilation of a code in one machine to be used for another machine; for instance, we use laptop to compile the code to be used and executed in Arduino, AVR, ATmega328 processor

- So, we compile the program for example on Intel, but we use in another different microprocessor, e.g. AVR microcontroller.

- Note that, the compiled code by Intel microprocessor, which will used in another microprocessor, will **NOT** work on Intel.

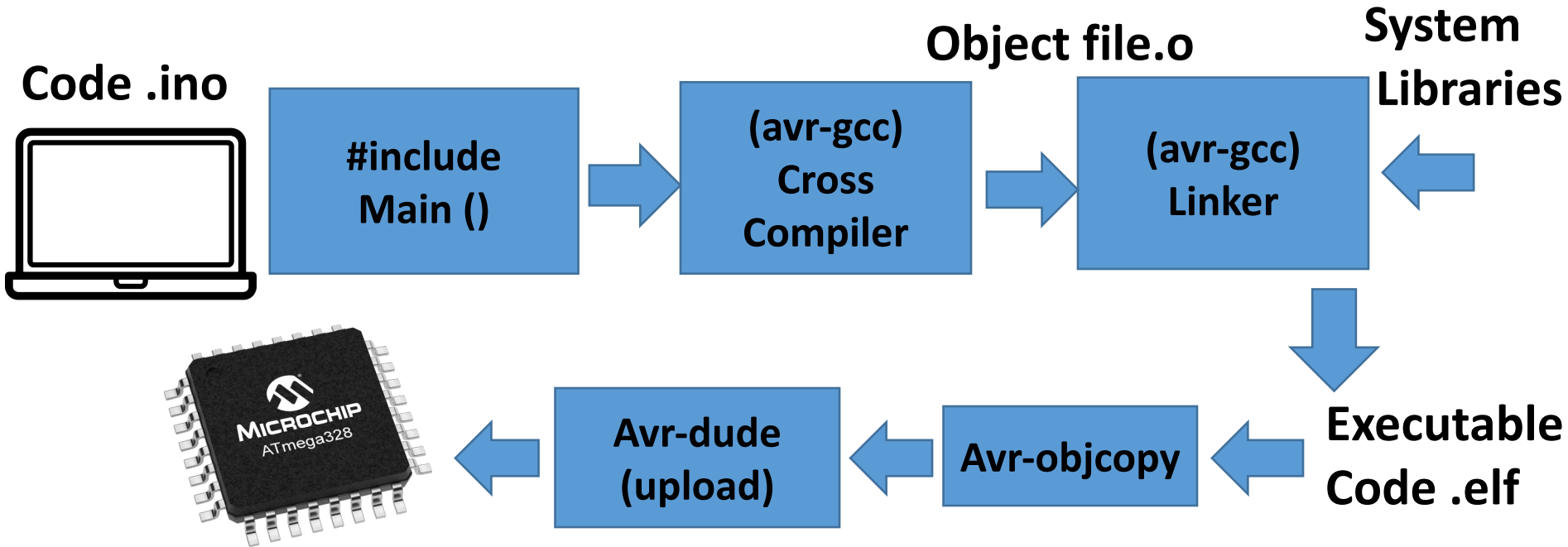
Software Toolchain (in general)

- A toolchain is the description for the sequence of tools (software tools) that you have to use in order to convert a program (source code) to an executable program (machine code) for the greatest platform (target microcontroller).
- The library takes the library's code and insert it into your code. It creates link between your code and library code that you use.
- The programmer write the executable code from the host (PC) to the microcontroller (flash memory) through USP cable.



Software Toolchain

- AVR-GCC is recalled to cross-compile the code to produce the executable code for the AVR
- Generate **object file.o**
- The linker take the object file and link to the Arduino library function to produce **.elf** executable file
- The Arduino does not understand **.elf**
- AVR-objcopy is recalled to change the format of the executable
- AVR-dude used to upload this file to the flash memory of the AVR



Classes in Arduino Language

- Arduino use C++, C to write its code with some Arduino libraries functions
- C++ is superset of C (what is written in C, can be compiled C++)
- A Class is used in OOP (object oriented programming)
- OOP is a way for organizing your code (encapsulate it)
- Group data, function together that are related into a single class
- Class can be defined by the programmer, it can be considered as type (integer, float, ect)
- The class has its own number and the functions (+, -,)

C Language Basics

- Basic C Operator
- #Defined
- Logical operator (==, >=, !, …)
- Conditional operation (if else, switch,)
- Loops
- Functions

Book: Introduction to Embedded Systems Using Microcontrollers and the MSP430
By:
Manuel Jimenez, Rogelio Palomera, Isidoro Couvertier

Programming

- There are a number of challenges when programming an embedded system project. It is common, first to develop the software design structure, particularly with large project.

- It is often not possible to program all functions in a single control loop, so the approach is structuring code and breaking it up into understandable parts

- Design your program: Use Flowchart to define code structure

Programming in C

Comments

- Two ways of commenting are used.
  - One is to place the comment between the markers **/*** and…***/**
  - Alternatively, use **//**
  - Comments are for **humans** to read. They are *ignored* by the compiler.

```
/*A program which flashes mbed
LED1 on and off. */
```

```
#include “mbed.h”  //include the mbed header file as part of this program
```

Data Types: Character and Integer

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Data Type: Floating Point

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Arithmetic Operators

where A = 10 and B = 20

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Arithmetic Operators

Example
Try the following example to understand all the arithmetic operators available in C

C program code

```c
#include <stdio.h>

main() {

   int a = 21;
   int b = 10;
   int c ;

   c = a + b;
   printf("Line 1 - Value of c is %d\n", c );

   c = a - b;
   printf("Line 2 - Value of c is %d\n", c );

   c = a * b;
   printf("Line 3 - Value of c is %d\n", c );

   c = a / b;
   printf("Line 4 - Value of c is %d\n", c );

   c = a % b;
   printf("Line 5 - Value of c is %d\n", c );

   c = a++;
   printf("Line 6 - Value of c is %d\n", c );

   c = a--;
   printf("Line 7 - Value of c is %d\n", c );
}
```

Results after compilation

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 22
Line 7 - Value of c is 21
```

Relational Operators

where A = 10 and B = 20

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

Suppose A = 1 and B = 0

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Logical Operators

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to float or double.

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

Bitwise Logical Operators

Now lets see truth table for bitwise &, | and ^

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.

Assignment Operators

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A

Assignment Operators (continued)

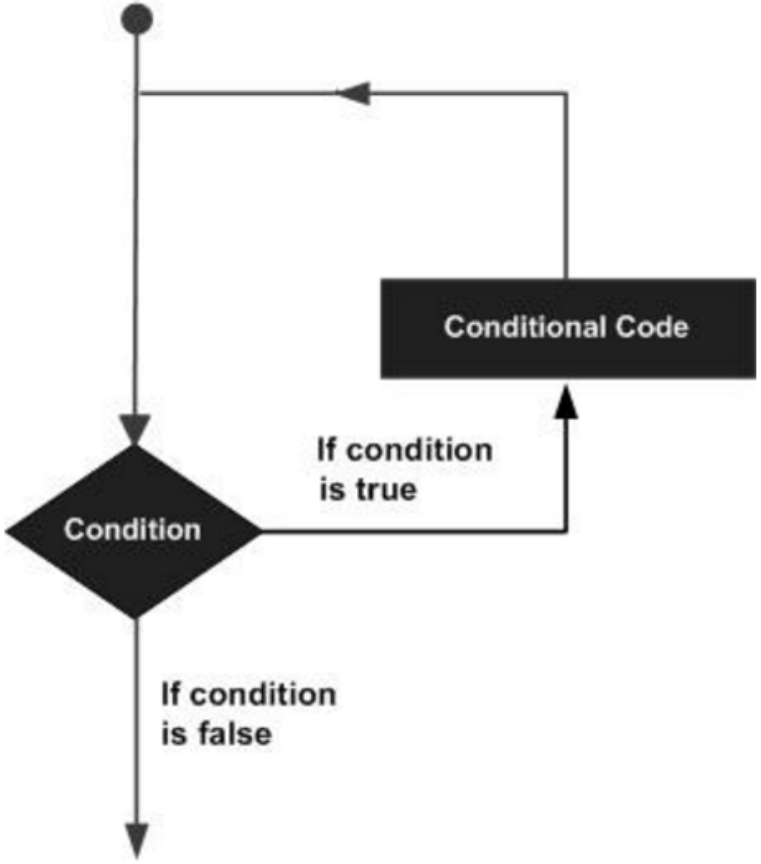
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Repetition in the Programs

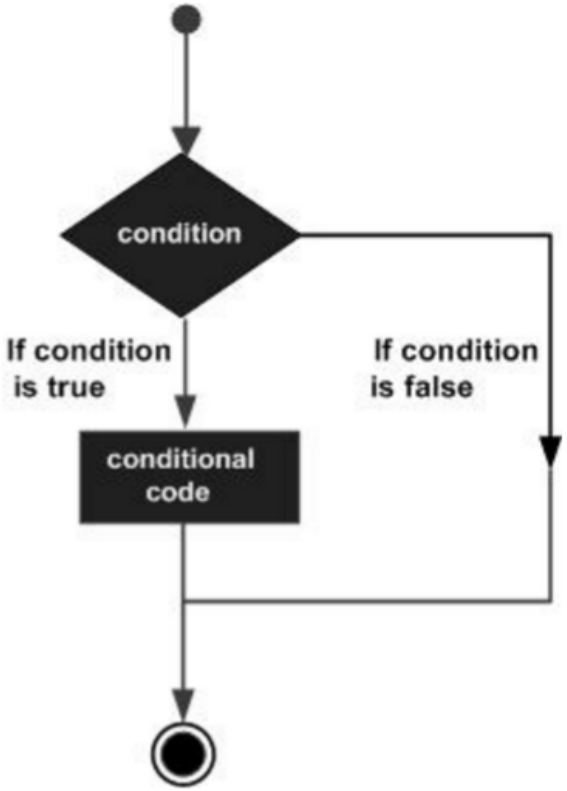
- In most software, the statements in the program may need to repeat for many times.
  - e.g., calculate the value of *n*!.
  - If *n* = 10000, it's not elegant to write the code as 1*2*3*…*10000.
- Loop is a control structure that repeats a group of steps in a program.
  - Loop body stands for the repeated statements.
- There are three C loop control statements:
  - While
  - for
  - do-while.

Conditional Statements

- Loops



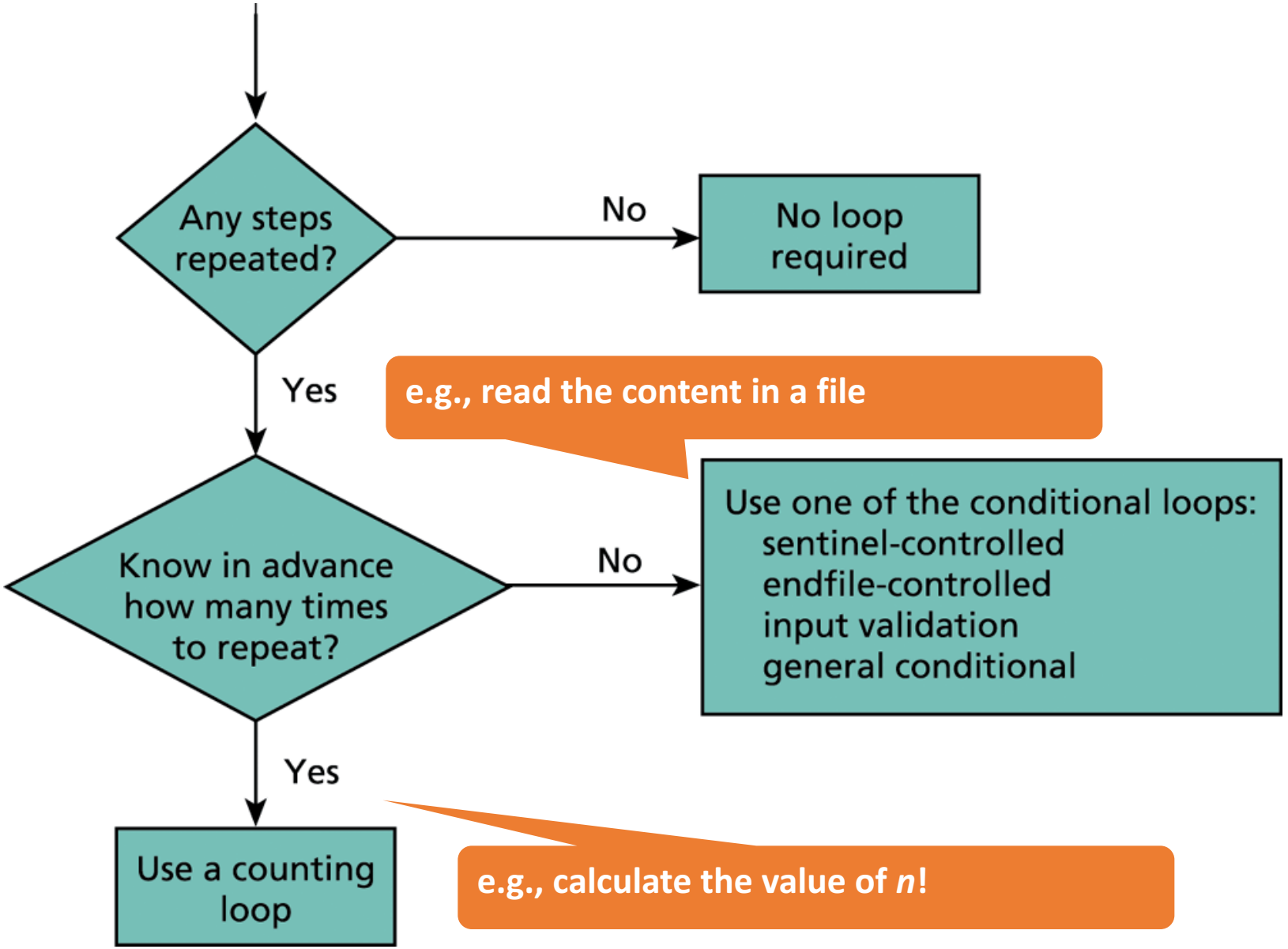
- Single Decisions



Loops

Loop Type & Description
while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop It is more like a while statement, except that it tests the condition at the end of the loop body.
nested loops You can use one or more loops inside any other while, for, or do..while loop.

Flow Diagram of Loop Choice Process



Comparison of Loop Choices (1/2)

Kind	When to Use	C Structure
Counting loop	We know how many loop repetitions will be needed in advance.	while, for
Sentinel-controlled loop	Input of a list of data ended by a special value	while, for
Endfile-controlled loop	Input of a list of data from a data file	while, for

Comparison of Loop Choices (2/2)

Kind	When to Use	C Structure
Input validation loop	Repeated interactive input of a value until a desired value is entered.	do-while
General conditional loop	Repeated processing of data until a desired condition is met.	while, for

The **`while`** Statement in C

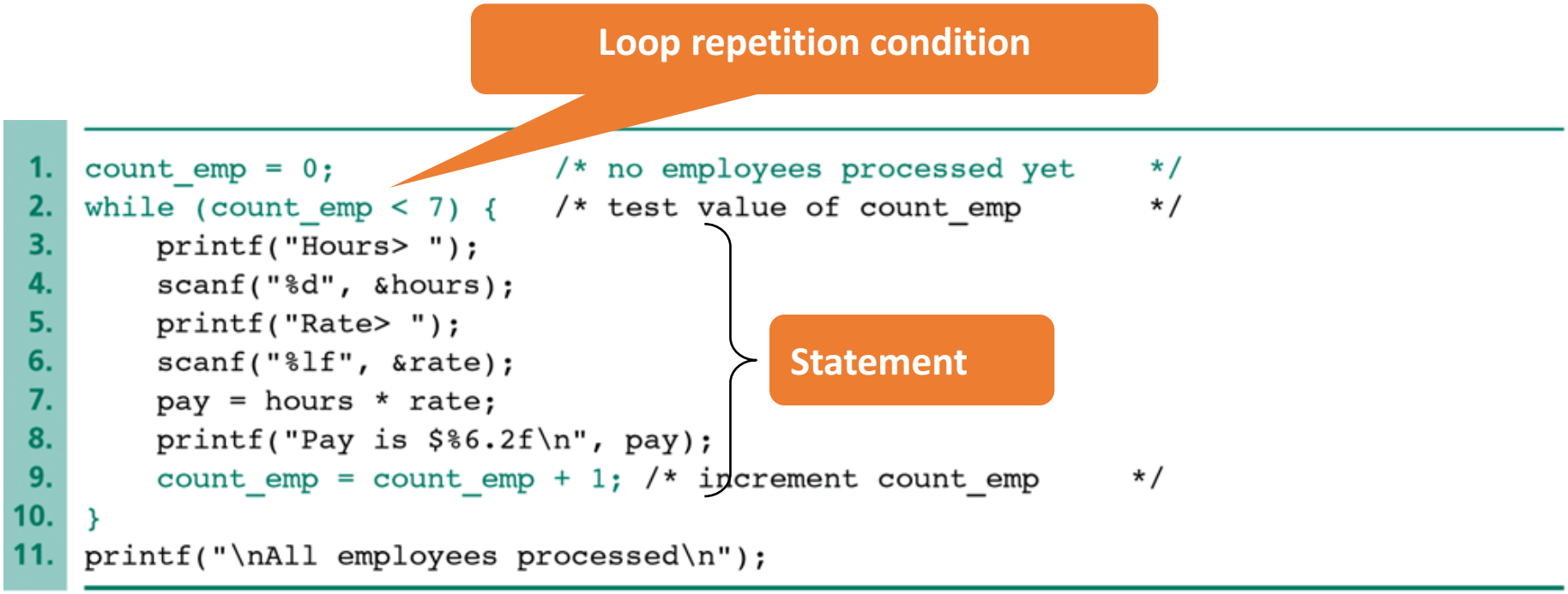
- The syntax of **while** statement in C:

  while (**loop repetition condition**)

  statement

- **Loop repetition condition** is the condition which controls the loop.
- The ***statement*** is repeated as long as the loop repetition condition is **true**.
- A loop is called an **infinite loop** if the loop repetition condition is always true.

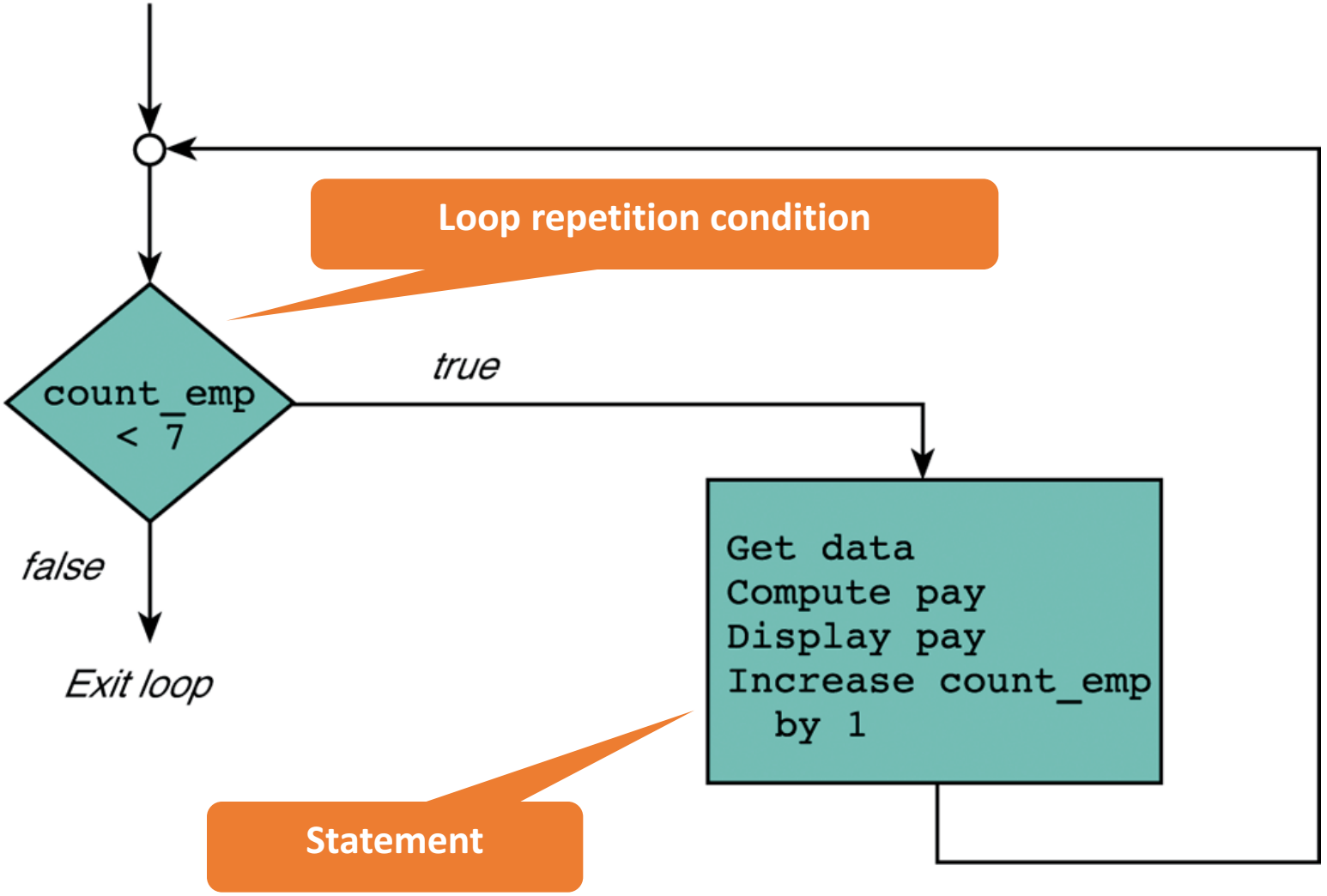
An Example of a `while` Loop



Loop control variable is the variable whose value controls loop repetition.

In this example, **`count_emp`** is the loop control variable.

Flowchart for a `while` Loop



The **`for`** Statement in C

- The syntax of **`for`** statement in C:

  `for` (**initialization expression**;

  loop repetition condition;

  update expression)

  statement

- The **initialization expression** set the initial value of the loop control variable.
- The **loop repetition condition** test the value of the loop control variable.
- The **update expression** update the loop control variable.

An Example of the `for` Loop

```
1.  /* Process payroll for all emp
2.  total_pay = 0.0;
3.  for   (count_emp = 0;
4.         count_emp < number_emp;
5.         count_emp += 1) {                    /* update                    */
6.       printf("Hours> ");
7.       scanf("%lf", &hours);
8.       printf("Rate > $");
9.       scanf("%lf", &rate);
10.      pay = hours * rate;
11.      printf("Pay is $%6.2f\n\n", pay);
12.      total_pay = total_pay + pay;
13. }
14. printf("All employees processed\n");
15. printf("Total payroll is $%8.2f\n", total_pay);
```

Initialization Expression

Loop repetition condition

Update Expression

`count_emp` is set to 0 initially.

`count_emp` should not exceed the value of `number_emp`.

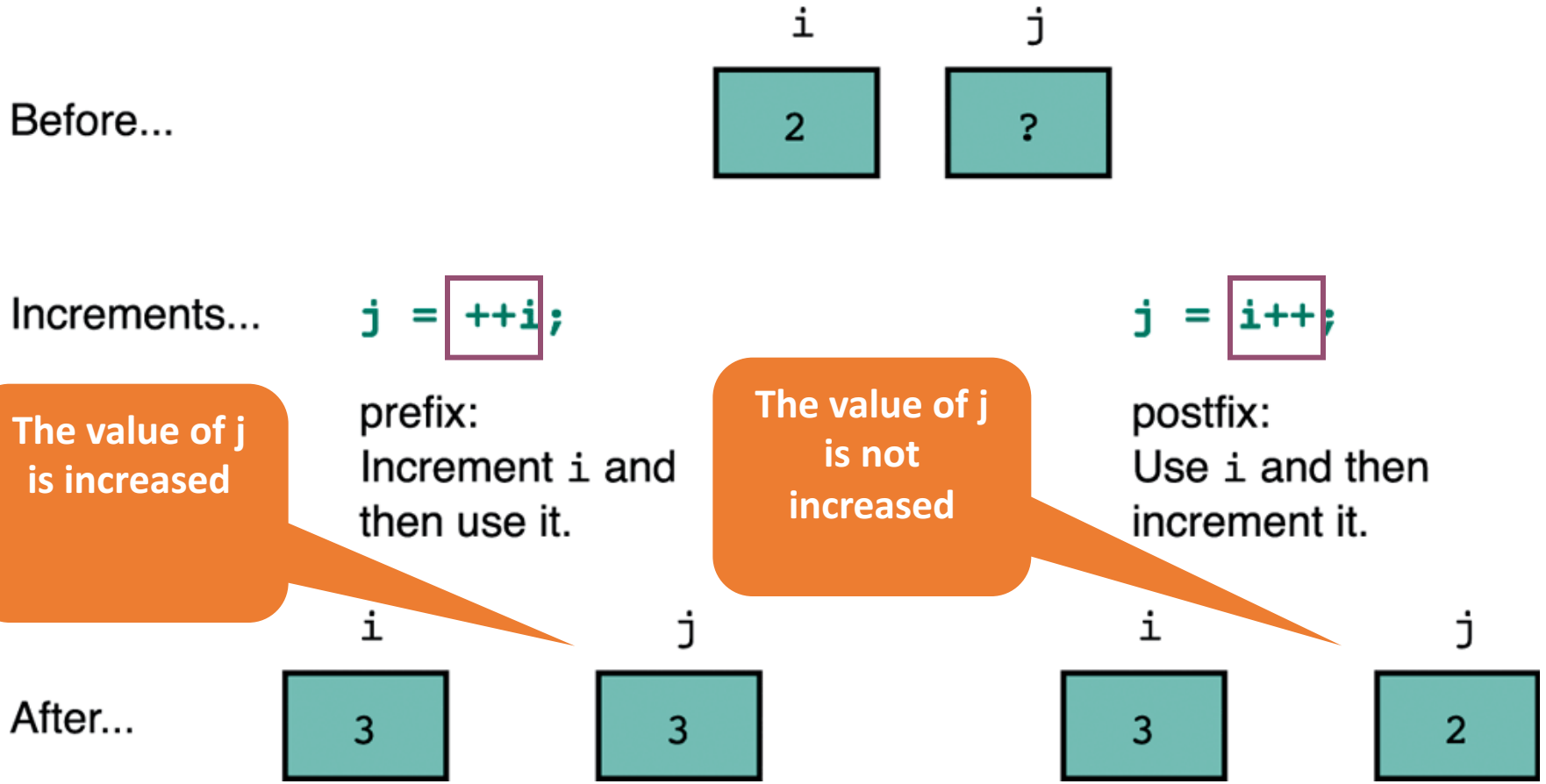
`count_emp` is increased by one after each iteration.

Increment and Decrement Operators

- The statements of increment and decrement are commonly used in the `for` loop.
- The increment (i.e., **++**) or decrement (i.e., **--**) operators are the frequently used operators which take only one operand.
- The increment/decrement operators increase or decrease the value of the single operand.
  - e.g., for (int i = 0; i < 100; **i++**){ … }
  - The variable i increase one after each iteration.

Comparison of Prefix and Postfix Increments

The value of the expression (that uses the ++/-- operators) depends on the position of the operator.



Sentinel-Controlled Loops

- Sometimes we may not know how many times the loop will repeat.
- One way to do this is to choose a **sentinel value** as an end marker.
  - The loop exits when the **sentinel value** is read.
- If the user wish to exit the loop, he or she has to input the **sentinel value**.
  - It is similar to the “logout” function in many applications.

An Example of Sentinel-Controlled while Loops

```
1.  /* Compute the sum of a list of exam scores. */
2.
3.  #include <stdio.h>
4.
5.  #define SENTINEL -99
6.
7.  int
8.  main(void)
9.  {
10.         int sum = 0,    /* output - sum of scores input so far                */
11.             score;      /* input - current score                              */
12.
13.         /* Accumulate sum of all scores.                                      */
14.         printf("Enter first score (or %d to quit)> ", SENTINEL);
15.         scanf("%d", &score);        /* Get first score.                       */
16.         while (score != SENTINEL) {
17.             sum += score;
18.             printf("Enter next score (%d to quit)> ", SENTINEL);
19.             scanf("%d", &score);    /* Get next score.                        */
20.         }
21.         printf("\nSum of exam scores is %d\n", sum);
22.
23.         return (0);
24. }
```

If the user wish to exit the loop, he or she has to input -99.

Nested Loops

- Nested loops consist of an **outer loop** with one or more **inner loops**.
- e.g.,

```
for (i=1;i<=100;i++){
    for(j=1;j<=50;j++){
            …
    }
}
```

Outer loop

Inner loop

- The above loop will run for 100*50 iterations.

The `do-while` Statement in C

- The syntax of do-while statement in C:

  do
  statement
  while (**loop repetition condition**);

- The *statement* is first executed.
- If the **loop repetition condition** is true, the *statement* is repeated.
- Otherwise, the loop is exited.

An Example of the `do-while` Loop

```
/* Find even number input */
do{
  printf("Enter a value: ");
  scanf("%d", &num);
}while (num % 2 !=0)
```

This loop will repeat if the user inputs odd number.

Homework #4 (1/2)

- Write a program that prompts the user to input an integer ***n***.
- Draw a triangle with ***n*** levels by star symbols. For example,

  n = 3,

  *

  **

  ***

- After drawing the triangle, repeat the above process until the user input a negative integer.

Homework #4 (2/2)

- An usage scenario:

```
Please input: 2
*
**
Please input: 3
*
**
***
Please input: -9
Thank you for using this program.
```

Infinite Loop

- When the conditional statement is empty, the loop will run forever (or until you turn the power off or hit the Ctrl + C keys).

```c
#include <stdio.h>

int main () {

   for( ; ; ) {
      printf("This loop will run forever.\n");
   }

   return 0;
}
```

Single Decision

Statement & Description
if statement An **if statement** consists of a boolean expression followed by one or more statements.
if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
switch statement A **switch** statement allows a variable to be tested for equality against a list of values.
nested switch statements You can use one **switch** statement inside another **switch** statement(s).

If-Else

- The ***if-else*** statement is used to make decisions.

```
if(expression)
      statement_1
else
      statement_2
```

expression: x = 0
statement: **x = 0;**

Questions:

1) if(expression == 0)
        statement_1
2) if(expression = 0)
        statement_1
3) if(expression == 0);
        statement_1

Switch

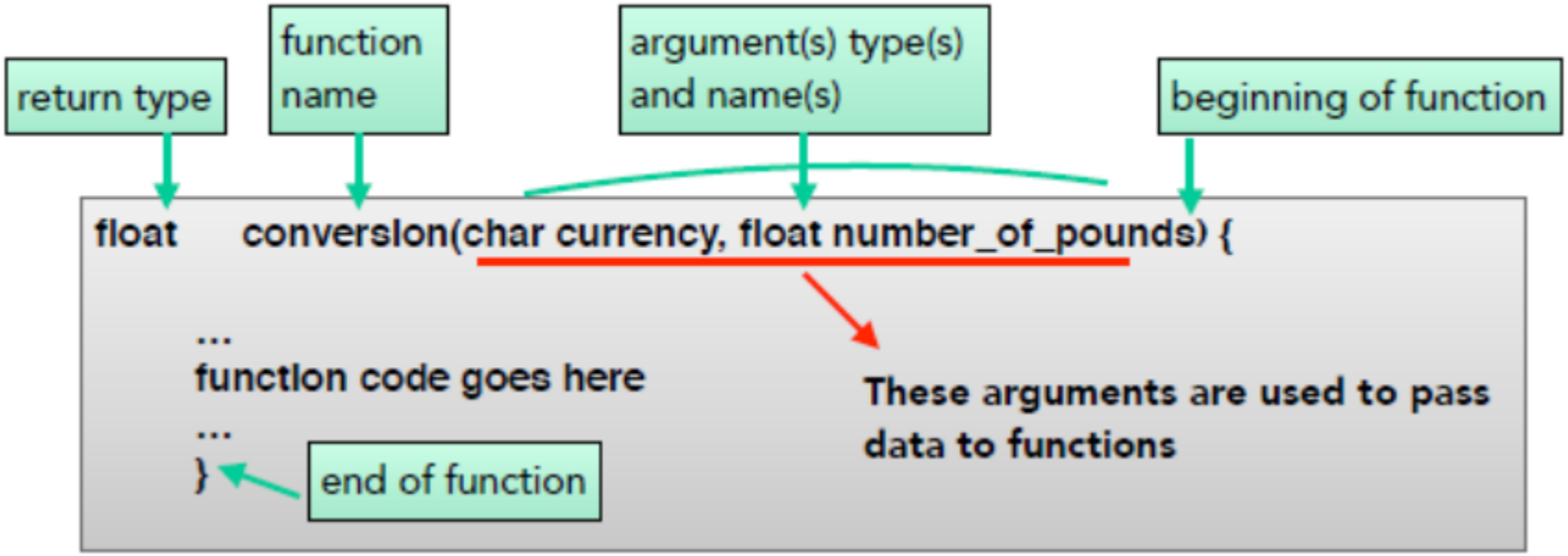
- The ***switch*** statement is special multi-way decision maker that tests whether an expression matches one of a number of constant values, and branches accordingly.

```
switch(i)
{
      case 0: display = 1; break;
      case 1: display = 2; break;
}
```

i is an integer

Functions

- Function Definitions



  - Only one return variable is allowed
  - The final statement of the function may be a **return**, which will specify the value returned to the calling program
- The ***main*** function
  - Program execution starts at the beginning of main() and ends at the end of main().
  - **Other functions may be written outside main(), and called from within it.**

Function: Void Return

Some functions perform the desired operations without returning a value.

In this case, the **return type** is **void**.

Delay

- How to **define** a delay function using e.g., for loop

```
void delay(int y)
{
        int i = 0;
        for(i = 0; i<10000*y; i++)
        {
        }
}
```

- How to **call** the delay function

```
delay(3);                    delay() = 3;
```

- Infinite Loops

```
while(1)
        {
                ....
        }
```

Arrays

- An array is a set of data elements, each of which has the same type.
- The declaration of an array:

```
int a_1[10];
```

Name of the array:	**a_1**
Number of the elements:	**10**
Elements of the array:	a_1[0], a_1[1], … a_1[9]
Data type of its elements:	**integer**

Structure

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

//A struct declaration defines a type

```
struct size{
        int w;
        int h;
};
```

size: a structure tag (optional)
members: w, h

//Declare instances of the structure
struct{int w; int h} size1, size2;
/* If the declaration is tagged, however, the tag can be used in definition of instances of the structure. */
struct size sz;

//Access the members
structure-name.member (e.g., **sz.w**): "." is structure member operator

Structure can mix data types

Datatype	C VARIABLE		C ARRAY		C STRUCTURE	
	Syntax	Example	Syntax	Example	Syntax	Example

int	int a	a = 20	int a[3]	a[0] = 10
				a[1] = 20
				a[2] = 30
					struct student
					{
					int a;
					char b[10];
					}
						a = 10
						b = "Hello"

char	char b	b='Z'	char b[10]	b="Hello"		

Pointer

- A pointer is a variable that contains the address of a variable.
  - If c is a char and p is a pointer that points to it

    p = &c; &: the address of

Operator * is dereferencing operator. When it is applied to a pointer, it accesses the object the pointer points to.

C Programming

Example

```
// Declare variables
char c1, c2;
char* p;

c1 = 'z';     // c1 is assigned 'z'
p = &c1;    // p is assigned the address of c1


c2 = *p;// c2 is assigned the value stored in the address stored in p

// Now, c2 == 'z'
// The above has the equivalent result as:

c1 = 'z'
c2 = c1;
```

Structure Pointers

- Structure pointers are just like pointers to ordinary variables
  - e.g., struct size *pp;
- Access the members
  - 1) **structure.member-of-structure**

  e.g., (*pp).w //*pp is the structure, and w is the member-of-structure
- 2) **pointer->member-of-structure**
- e.g., pp->w //pp is the pointer, w is the member-of-structure

```
struct size origin, *pp;
pp = &origin;
pp->w
```

Typedef

- C provides a facility called typedef for creating new data type names
  - e.g., typedef int Length;

```
typedef struct size{
        int w;
        int h;
} Treepoint;
```

This creates a new type keyword called Treepoint (a structure)

- A typedef declaration does not create a new type in any sense; it merely adds a new name for some existing type.

Compiler Directives

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol
  - #define – This macro defines constant value and can be any of the basic data types.
  - #include <file_name> – The source code of the file “file_name” is included in the main C program where “#include <file_name>” is mentioned.

Compiler Directive — #define directive

- #define directive
  - Define a *symbolic name* or *symbolic constant* to be a particular string of characters.

For example: #define PI 3.14

In LPC17xx.h file:

```
typedef struct
{
        ....
} GPIO_TypeDef;

#define GPIO0_BASE  constant_value
```

```
#define GPIO0    ((GPIO_TypeDef *) GPIO0_BASE)
```

Compiler directives are messages to the compiler

Compiler directives all start with a hash, #

Compiler Directive — #include directive

- #include directive
  - The #include directive directly inserts another file into the file that invokes the directive.
  - e.g., **#include <>** /*used to enclose files held in a directory different from the current working directory*/
  - **#include "mbed.h"** /*Used to contain a file located within the current working directory */

QUESTIONs?