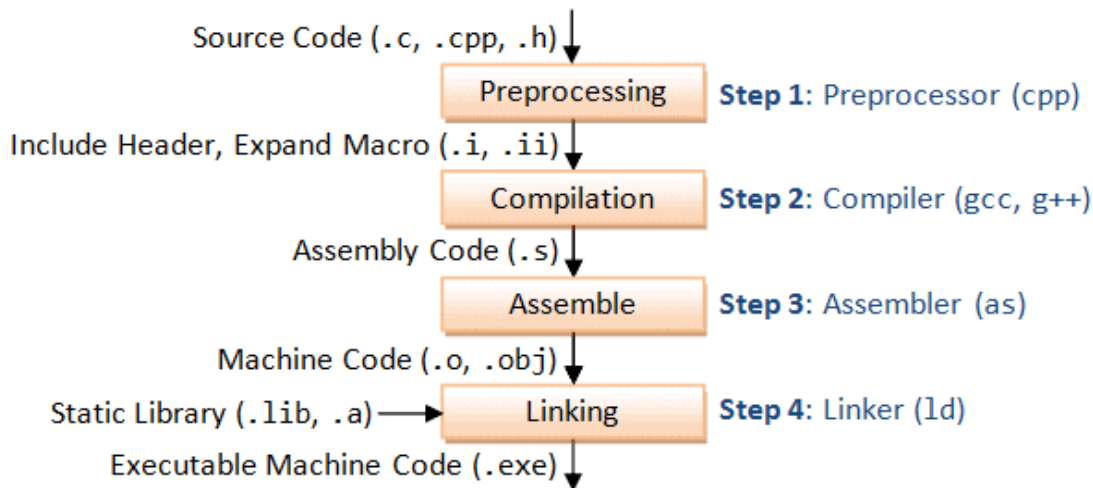


# Compilation Process



## 1. Preprocessing: The First Step

At the outset, the C preprocessor comes into play. It handles preprocessor directives, which begin with a '#' symbol, and includes header files in the code. When you use [hashtag #include](#) <stdio.h>, for example, the preprocessor replaces it with the content of the **stdio.h** file. This step generates an intermediate code with all the necessary declarations and macros.

## 2. Compilation: Translating to Assembly

Once preprocessing is complete, the actual compilation commences. The compiler takes the C code and translates it into assembly code, specific to the target platform. Assembly code represents a low-level view of the program, consisting of instructions that the processor can execute.

## 3. Assembly: From Assembly to Machine Code

Next, the assembler converts the assembly code into machine code, also known as object code. The object code consists of binary representations of instructions and data. This step is a crucial bridge between human-readable code and the language the computer understands.

## 4. Linking: Bringing It All Together

Now that we have object files with machine code, the linker takes center stage. It resolves external dependencies, such as functions from libraries or other object files, and combines all the object files into a single executable. The linker ensures that the final program is self-contained and ready to run.



## Runtime Errors

الأخطاء أثناء وقت التشغيل (Runtime Errors) تحدث بعد إنشاء ملف .exe. وأثناء تنفيذ البرنامج. يمكن تمييز هذا النوع من الأخطاء من خلال مخرجات وحدة التحكم (CMD) حيث تظهر أرقام عشوائية بعد كلمة "returned"، مما يشير إلى حدوث خطأ أثناء وقت التشغيل.

### بعض الأسباب المحتملة لحدوث أخطاء وقت التشغيل:

1. القسمة على صفر: محاولة قسمة عدد على صفر، مما يؤدي إلى خطأ رياضي.
2. فتح ملف غير موجود: محاولة الوصول إلى ملف لا يوجد في المسار المحدد.
3. نفاذ الذاكرة: عدم توفر ذاكرة كافية لتنفيذ العمليات المطلوبة، مما يؤدي إلى فشل البرنامج.
4. خطأ في تقسيم الذاكرة (Segmentation Fault): محاولة الوصول إلى جزء من الذاكرة غير مصرح به أو غير صالح.

This type of error occurs while the program is running. Because this is not a compilation error, the compilation will be completed successfully. These errors occur due to segmentation fault when a number is divided by division operator or modulo division operator.

## Logical Errors

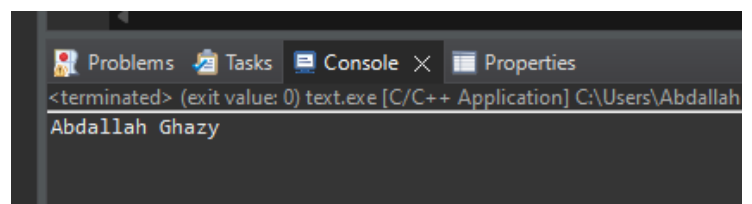
المسؤول عن اكتشاف الأخطاء المنطقية هو المستخدم، حيث أن هذه الأخطاء تؤدي إلى نتائج غير صحيحة أو غير متوقعة. غالباً ما يكون السبب هو عدم ترتيب الكود بشكل صحيح أو استخدام نوع بيانات غير متوافق.

### أسباب حدوث الأخطاء المنطقية:

- خلل في المنطق البرمجي: مثل استخدام عمليات حسابية غير صحيحة أو ترتيب غير صحيح للتعليمات.
- استخدام متغيرات غير مناسبة: مثل استخدام نوع بيانات غير متوافق مع العمليات المطلوبة.
- شروط غير صحيحة: كتابة شروط خاطئة في العبارات الشرطية أو حلقات التكرار.

Even if the syntax and other factors are correct, we may not get the desired results due to logical issues. These are referred to as logical errors. We sometimes put a semicolon after a loop, which is syntactically correct but results in one blank loop. In that case, it will display the desired output.

```
1 // C program to demonstrate
2 // a logical error
3 #include <stdio.h>
4
5 // Driver code
6
7 int main()
8 {
9     int i;
10    for(i = 0; i <= 5; i++);
11    {
12        printf("Abdallah Ghazy");
13    }
14    return 0;
15 }
```



## Linker Errors

المسؤول عن الإعلان عن أخطاء الربط هو اللينكر (Linker) ، وتحدث هذه الأخطاء أثناء عملية توليد الملف التنفيذي (Executable File) حيث يقوم اللينكر بدمج جميع الأكواد في ملف .exe وربطها معاً. يمكن تمييز الخطأ بوجود رسائل تبدأ بـ "error: ld" في شاشة الكونسول.

### أسباب حدوث أخطاء الربط:

- تعارض التعريفات (Definition Conflict) عندما يوجد تعريفات متعددة لنفس المتغير أو الدالة في ملفات مختلفة.
- تعريف مفقود (Missing Definition) عندما لا يمكن للينكر العثور على تعريف دالة أو متغير تم الإشارة إليه في الشفرة الأخرى.
- نوع بيانات غير متوافق (Incompatible Data Types) عندما تكون هناك تباينات في أنواع البيانات بين التعريفات والاستخدامات المختلفة للمتغيرات أو الدوال.

When the program is successfully compiled and attempting to link the different object files with the main object file, errors will occur. When this error occurs, the executable is not generated. This could be due to incorrect function prototyping, an incorrect header file, or other factors. If main() is written as Main(), a linked error will be generated.

```
C:\Users\Rashi\Desktop>gcc linkererror.c -o linkererror.exe
C:/TDM-GCC-64/bin/./lib/gcc/x86_64-w64-mingw32/10.3.0/./././././x86_64-w64-mingw32/bin/ld.exe: C:/TDM-GCC-64/bin/./lib/gcc/x86_64-w64-mingw32/10.3.0/./././././x86_64-w64-mingw32/lib/./lib/libmingw32.a(lib64_libmingw32_a-crt0_c.o): in function `main':
C:/crossdev/src/mingw-w64-v8-git/mingw-w64-crt/crt/crt0_c.c:18: undefined reference to `WinMain'
collect2.exe: error: ld returned 1 exit status
```

## Semantic Errors

أخطاء الدلالة تحدث عندما يتم تنفيذ إجراءات غير مفهومة للمترجم (Compiler) ، وهذه الأخطاء لا تتمثل في صياغة البرنامج بشكل نحوي صحيح ولا تظهر رسائل خطأ أثناء مرحلة الترجمة. بدلاً من ذلك، قد تؤدي هذه الأخطاء إلى سلوك غير متوقع أو أداء غير صحيح للبرنامج عند تنفيذه.

تعريف متغير محلي ثم جمع قيم عشوائية: في هذه الحالة، يمكن أن تكون القيم المخزنة في المتغير المحلي عشوائية، مما يؤدي إلى نتائج غير متوقعة.

الوصول إلى عنصر غير موجود في مصفوفة: يمكن أن يؤدي الوصول إلى عنصر خارج نطاق المصفوفة إلى سلوك غير متوقع أو حتى إلى تعطل البرنامج.

تخزين نوع بيانات غير متوافق في متغير: يمكن أن يؤدي تخزين قيمة من نوع بيانات غير متوافق في متغير إلى تشغيل غير صحيح للبرنامج أو إلى فشل في التنفيذ.

When a sentence is syntactically correct but has no meaning, semantic errors occur. This is similar to grammatical errors. If an expression is entered on the left side of the assignment operator, a semantic error may occur.

### أخطاء الدلالة: (Semantic Errors)

1. التعريف:
  - أخطاء الدلالة تحدث عندما يتم تنفيذ إجراءات غير مفهومة للمترجم (Compiler) ، حيث يؤدي هذا التصرف إلى سلوك غير متوقع للبرنامج عند تشغيله.
2. سبب الحدوث:
  - تحدث هذه الأخطاء نتيجة لاستخدام تعليمات أو أوامر لا تتناسب مع بنية البرنامج، مما يؤدي إلى تنفيذ عمليات غير صحيحة أو غير متوقعة.
3. أمثلة:
  - مثال على أخطاء الدلالة هو محاولة تخزين قيمة من نوع بيانات غير متوافق في متغير، أو استخدام متغيرات بدون تعريفها بشكل صحيح، أو الوصول إلى عناصر خارج نطاق المصفوفة.

### الأخطاء المنطقية: (Logical Errors)

1. التعريف:
  - الأخطاء المنطقية تحدث عندما يكون المنطق البرمجي غير صحيح، حيث يؤدي هذا إلى إنتاج نتائج غير متوقعة أو غير صحيحة عند تنفيذ البرنامج.
2. سبب الحدوث:
  - يحدث هذا النوع من الأخطاء بسبب خلل في التفكير البرمجي أو استخدام خوارزميات أو تسلسلات غير صحيحة للعمليات البرمجية.
3. أمثلة:
  - مثال على أخطاء المنطق هو حساب متوسط القيم لمصفوفة بطريقة خاطئة، أو استخدام شرط غير صحيح في تنفيذ الحلقات، أو استخدام عمليات حسابية غير صحيحة.

# Toolchain

## What is Tool Chaining?

Tool chaining involves integrating multiple tools to create a pipeline or workflow. Each tool in the chain performs a specific function, and the output of one tool becomes the input for the next. This method is commonly used in software development, including:

- **Compilers and Assemblers:** Converting source code to machine code.
- **Linkers:** Combining multiple object files into a single executable.
- **Debuggers:** Analyzing and fixing code issues.
- **Version Control Systems:** Managing changes to source code over time.
- **Continuous Integration/Continuous Deployment (CI/CD) Tools:** Automating the build, test, and deployment processes.

## Native Tool Chaining

**Native Tool Chaining** refers to the process of compiling and building software directly on the target system for which the software is intended to run. The toolchain used in this process is designed to work on the same architecture and operating system as the target system. This approach is commonly used in desktop and server environments where the development and execution environments are the same.

### *Characteristics of Native Tool Chaining:*

- **Same Architecture:** The tools (compiler, linker, etc.) and the target executable run on the same type of CPU and operating system.
- **Simpler Setup:** No need to configure a separate build environment or handle compatibility issues between different architectures.
- **Direct Testing:** Immediate testing on the target system is possible, simplifying debugging and validation.

## Cross Tool Chaining

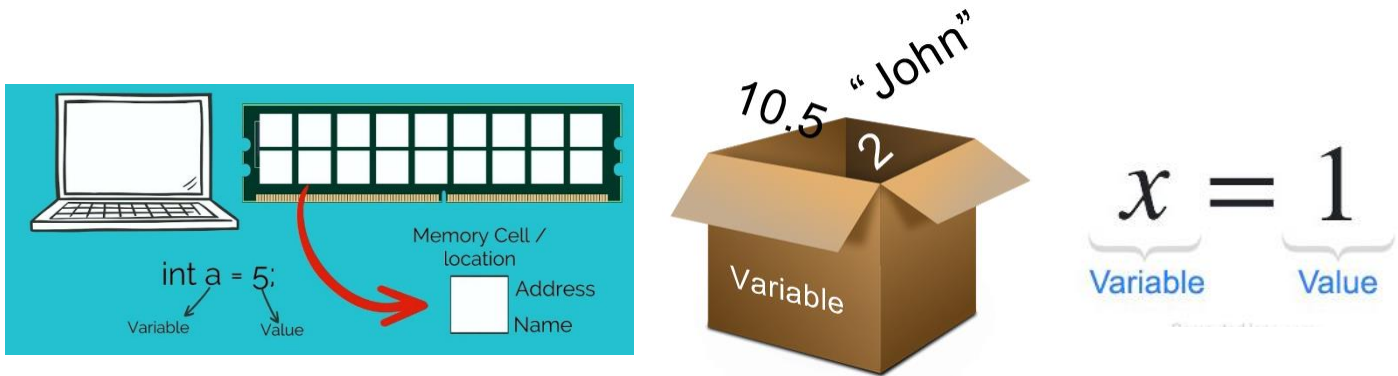
**Cross Tool Chaining** refers to the process of compiling and building software on a host system that is different from the target system where the software will run. The toolchain used in this process is specifically designed to generate code for a different architecture and/or operating system than the one it is running on. This approach is common in embedded systems development, where the target hardware is often different from the development workstation.

### *Characteristics of Cross Tool Chaining:*

- **Different Architectures:** The host system (development machine) and the target system (execution environment) have different architectures.
- **Cross Compiler:** A cross compiler is used to generate binaries for the target architecture.
- **Complex Setup:** Requires setting up a development environment that includes the cross compiler and other necessary tools.
- **Deployment Step:** The compiled code needs to be transferred and flashed onto the target system for testing.

# Variables

المتغيرات هي أسماء تستخدم للإشارة إلى مواقع أو مساحات في الذاكرة الحاسوبية التي تستخدم لتخزين البيانات أثناء تشغيل البرنامج. يمكن أن تكون هذه البيانات أرقاماً، نصوصاً، أو قيم من أنواع بيانات أخرى.



## Variable Name

Variable name can be any set of letters and numbers of a length up to 256 characters.

Following constraints must be respected:

- Do not use any reserved keyword in C like (void, include, int)
- Do not use space or any special character inside variable name except “\_”.
- Do not start with a number

Correct variable names	M n Values	m_name counter name1	name2 min_value
Wrong variables names	Min value max>name void	5names printf	min-value

## Variable Definition, Declaration, initialization in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows -

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows -

```
type variable_name = value;
```

## What is the difference between declaring a variable and defining a variable?

**Declaration** of a variable in C hints the compiler about the **type** and **size** of the variable in compile time. Similarly, **declaration** of a function hints about **type** and **size** of function parameters. **No space is reserved in memory for any variable in case of declaration.**

- ✓ **Example:** int a;  
Here variable 'a' is declared of data type 'int'

**Defining** a variable means **declaring** it and also **allocating space** to hold it.

- ✓ **We can say "Definition = Declaration + Space reservation".**
- ✓ **Example:** int a = 10;  
Here variable "a" is described as an int to the compiler and memory is allocated to hold value 10

## Comments

```
double temprature;

//Supply the temprature in Fahrenheit
printf("Enter the temprature in Fahrenheit : \r\n");
scanf("%lf", &temprature);

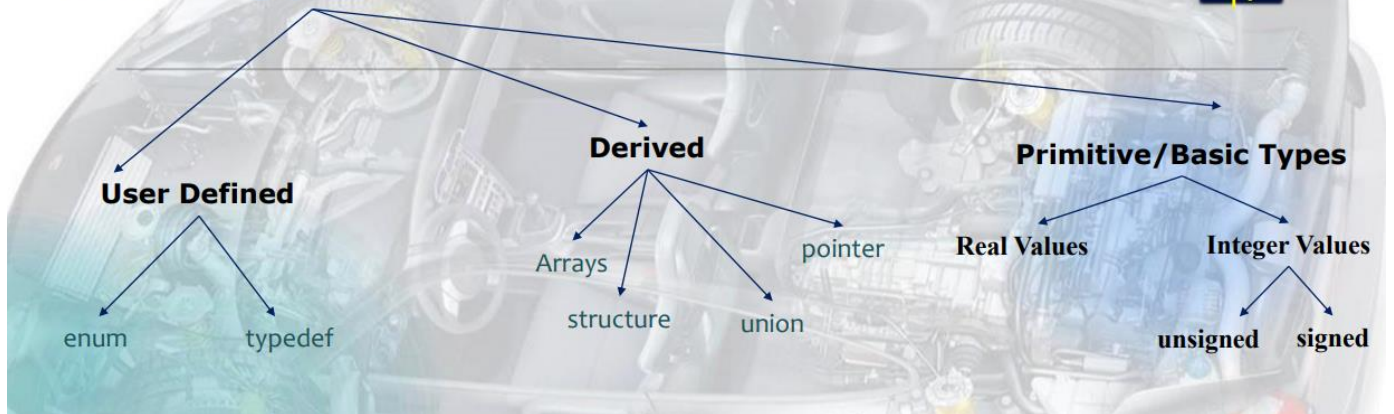
/*Convert temprature from
Fahrenheit to Celsius */
temprature = (temprature - 32.0) * 5.0/9.0;

//prints the
//result
printf("The temprature in Celsius is %lf\r\n",
        temprature);
}
```



# Data Types

## Data Types



## Integer Data Types in C

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	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions **sizeof(type)** yields the storage size of the object or type in bytes.

- **Sizeof** is a unary operator and not a function
- **Unary Operator:** sizeof is considered a unary operator because it operates on a single operand, which can be a type, a variable, or an expression.
- **Sizeof int depends on the machine (4Bytes or 2Bytes)**

### Range: 2Bytes

- ✓ Unsigned range: 0 to 65535 ( $2^N - 1$ )
- ✓ Signed range: -32768 to +32767 (  $2^N$  ) to  $(2^N - 1)$  )

# Modifiers used for integer

## Short, long, signed, unsigned

يمكنك استخدام المَحَدِّدَات (modifiers) مع نوع البيانات `int` في لغة `C` لتعديل حجمه ونطاق الأرقام التي يمكنه تخزينها.

`Sizeof (int) = 4Bytes or 2Bytes` depends on the machine

`sizeof(short int) = 2Bytes`

`sizeof(long int) = 8Bytes`

## Format Specifiers

Format Specifier	Type
<code>%c</code>	Character
<code>%d</code>	Signed integer
<code>%f</code>	Float values
<code>%i</code>	Unsigned integer
<code>%l</code> or <code>%ld</code> or <code>%li</code>	Long
<code>%lf</code>	Double
<code>%Lf</code>	Long double
<code>%lu</code>	Unsigned int or unsigned long
<code>%lli</code> or <code>%lld</code>	Long long
<code>%llu</code>	Unsigned long long

## What Happen When We Exceed Valid Range of Built-in Data Types in C?

### Signed Integers (signed int)

Signed integers can hold both positive and negative values. The range of a signed int typically depends on the compiler and the system architecture, but it is commonly from -2,147,483,648 to 2,147,483,647 (assuming 32-bit int).

- **Exceeding the Upper Limit:** If you exceed the maximum positive value (INT\_MAX), the behavior is undefined. This means the compiler or runtime environment may handle it in unexpected ways. It could wrap around (**become negative**), crash the program, or produce incorrect results.
- **Exceeding the Lower Limit:** Similarly, if you go below the minimum negative value (INT\_MIN), the behavior is undefined. It could wrap around (**become positive**), crash the program, or produce incorrect results.

### Unsigned Integers (unsigned int)

Unsigned integers can only represent non-negative values (including zero). The range of an unsigned int is typically from 0 to 4,294,967,295 (assuming 32-bit unsigned int).

- **Exceeding the Upper Limit:** If you exceed the maximum value (UINT\_MAX), the behavior wraps around. This means the value will start again from zero and continue counting up. For example,  $\text{UINT\_MAX} + 1$  **results in 0**.
- **Exceeding Zero:** Unsigned integers cannot represent negative values. If you attempt to assign a negative value or perform operations that result in a negative value (like subtracting a larger number from a smaller one), the result is still well-defined within the modulus of  $\text{UINT\_MAX} + 1$ .

## Floating-Point Data Types in C

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

يبدو أنك تشير إلى الفروق بين النوعين float و double في لغة البرمجة، حيث تختلف في الحجم والدقة التي يمكن تمثيلها. إليك شرح موجز لكل نوع:

### Float (float)

- حجم الذاكرة: عادةً ما يشغل 4 float بايت في الذاكرة.
- دقة التمثيل: يوفر تمثيلاً بدقة تصل إلى حوالي 7 أرقام بعد الفاصلة العشرية.
- مثال: إذا كان لدينا float x = 3.14159265358979، قد يتم تقريب القيمة المخزنة لتصبح قريبة من 3.141593.

### Double (double)

- حجم الذاكرة: عادةً ما يشغل 8 double بايت في الذاكرة، أي ضعف حجم float.
- دقة التمثيل: يوفر تمثيلاً بدقة أكبر، تصل إلى حوالي 15-16 رقماً بعد الفاصلة العشرية.
- مثال: إذا كان لدينا double y = 3.14159265358979، يمكن أن يحتفظ بالقيمة بدقة أكبر مقارنة بـ float.

### التمثيل الرمزي:

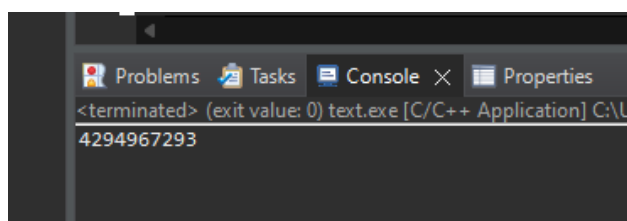
- Float يمكن تمثيله بمثل 3.141593 حيث يكون العدد المكون من 32 بت.
- Double يمكن تمثيله كذلك بـ 3.14159265358979

### What is the output?

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

int main() {
    unsigned int i = 1;
    int j = -4;
    printf("%u", i+j);

    return 0;
}
```

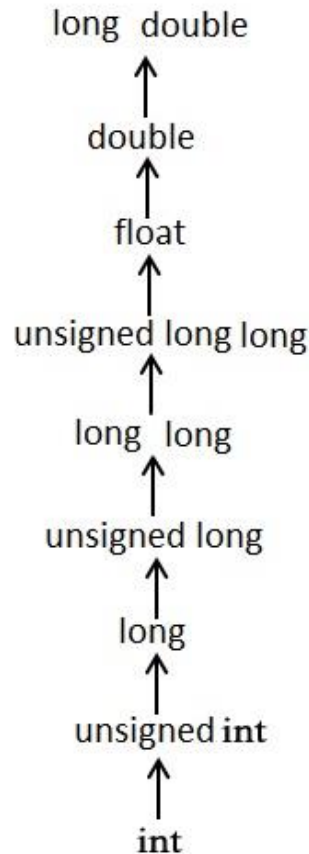


ناتج عملية  $i + j$  هو -3 - لكن بسبب وجود %u و معناها اطبع الأرقام unsigned حيث ان هناك خانة للرقم السالب لكن مع استخدام %u ترجم الخانة علي انها من الأرقام و بدل مايطبع -3 طبع 4294967293

# Type Conversion in C

## Implicit Type Conversion in C

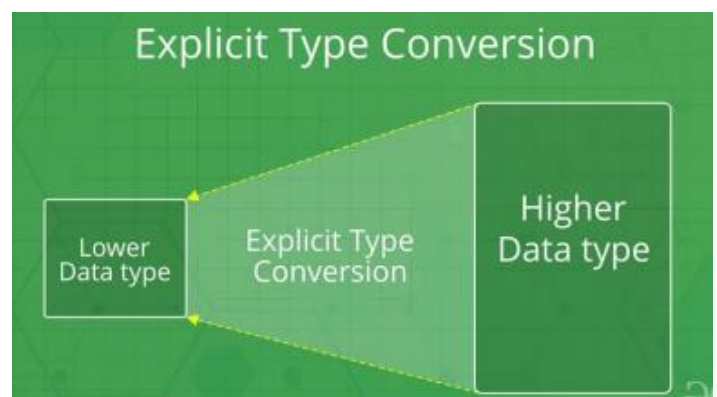
In C, implicit type conversion takes place automatically when the compiler converts the type of one value assigned to a variable to another data type. It typically happens when a type with smaller byte size is assigned to a "larger" data type. In such implicit data type conversion, the data integrity is preserved.



## Explicit Type Conversion in C

When you need to convert a data type with higher byte size to another data type having lower byte size, you need to specifically tell the compiler your intention. This is called explicit type conversion.

```
type2 var2 = (type1) var1;
```



## integer and Float Conversions

int/float >>> float

Operation	Result	Operation	Result
5 / 2	2	2 / 5	0
5.0 / 2	2.5	2.0 / 5	0.4
5 / 2.0	2.5	2 / 5.0	0.4
5.0 / 2.0	2.5	2.0 / 5.0	0.4

k is an integer variable and a is a real variable.

Arithmetic Instruction	Result	Arithmetic Instruction	Result
k = 2 / 9	0	a = 2 / 9	0.0
k = 2.0 / 9	0	a = 2.0 / 9	0.2222
k = 2 / 9.0	0	a = 2 / 9.0	0.2222
k = 2.0 / 9.0	0	a = 2.0 / 9.0	0.2222
k = 9 / 2	4	a = 9 / 2	4.0
k = 9.0 / 2	4	a = 9.0 / 2	4.5
k = 9 / 2.0	4	a = 9 / 2.0	4.5
k = 9.0 / 2.0	4	a = 9.0 / 2.0	4.5

In the first statement, since both 2 and 9 are integers, the result is an integer, i.e. 0. This 0 is then assigned to k. In the second statement 9 is promoted to 9.0 and then the division is performed. Division yields 0.222222. However, this cannot be stored in k, k being an **int**. Hence it gets demoted to 0 and then stored in k.

## C Programming Input Output (I/O): printf() and scanf()

The `scanf()` function reads formatted input from standard input (keyboard) whereas the `printf()` function sends formatted output to the standard output (screen).

Eclipse's terminal emulator might be different and do more buffering.

Try calling `fflush(stdout);` between the `printf()` and the call to `scanf()`.

### scanf()

```
main.c
1 //Prepared by keroles
2 #include <stdio.h>
3 int main()
4 {
5     int testInteger;
6     printf("Enter an integer: ");
7     fflush(stdout);
8     scanf("%d",&testInteger);
9     printf("Number = %d",testInteger);
10    return 0;
11 }
12
```

Problems Tasks Console Properties AVR Device Explorer AVR Supported MCUs  
<terminated> (exit value: 0) first\_c\_code.exe [C/C++ Application] D:\courses\C\_Course\first\_c\_code\Debug\first\_c\_code.exe (3/17/17, 11:51 AM)  
Enter an integer: 12  
Number = 12

قمنا باستخدام الدالة `scanf` وداخلها ستجد وسيطين، الأول نقوم فيه بتحديد نوع القيمة التي سيدخلها المستخدم حيث هنا وضعنا الرمز `%d` سابقا في الدالة `printf` حيث قلنا أنها خاص بالأعداد الصحيحة، وفي الوسيط الثاني يوجد `testInteger`، والرمز `&` يعني وضع القيمة التي أدخلها المستخدم في عنوان المتغير `testInteger`

أما بالنسبة لباقي أنواع المتغيرات فسنستعمل نفس الطريقة فقط نقوم بتغيير الرمز `%d` إلى نوع المتغير الذي نريد إستقباله، فمثلا إذا أردنا من المستخدم أن يقوم بإدخال رمز بدل رقم نضع الرمز `%c` في الدالة `scanf`

## C Floats Input/Output

```
main.c
1 //Prepared by keroles
2 #include <stdio.h>
3 int main()
4 {
5     float f;
6     printf("Enter a number: ");
7     fflush(stdout);
8     // %f format string is used in case of floats
9     scanf("%f",&f);
10    printf("Value = %f", f);
11    return 0;
12 }
13
```

Problems Tasks Console Properties AVR Device Explorer AVR Supported MCUs  
<terminated> (exit value: 0) first\_c\_code.exe [C/C++ Application] D:\courses\C\_Course\first\_c\_code\Debug\first\_c\_code.exe (3/17/17, 11:55 AM)  
Enter a number: 12.548  
Value = 12.548000



<code>scanf("%d/%d", &amp;W, &amp;H); printf("\r\nArea is %d", W*H);</code>		integer value. The combination ' <code>%d/%d</code> ' is used to scan two integer value separated by '/ '.
<code>int x = 172; printf("X equals %x", x);</code>	X equals ac	The directive ' <code>%x</code> ' prints the integer value in small hexadecimal format.
<code>int X = 172; printf("X equals %X", X);</code>	X equals AC	The directive ' <code>%X</code> ' prints the integer value in capital hexadecimal format.
<code>int X; printf("Enter X in hexadecimal format:"); scanf("%X", &amp;X); printf("\r\nX equals %d", X);</code>	Enter X in hexadecimal format: AC X equals 172	The directive ' <code>%X</code> ' also used to scan values in hexadecimal format.
<code>float R = 2.5; printf("R equals %f", R);</code>	R equals 2.5	The directive ' <code>%f</code> ' prints a real (float) value.
<code>int X = 6235; printf("X equals %10d", X);</code>	X equals 6235 -----	Prints the number in 10 digits including the '.' and 2 digits in the fraction part.
<code>float R = 8372.5675365; printf("R equals %10.2f", R);</code>	R equals 8372.56 -----	Prints the number in 10 digits including the '.' and 2 digits in the fraction part.
<code>int X = 15; printf("X equals %05d", X);</code>	X equals 00015 -----	Prints the number in 5 digits and pad it with zeros.

```

1 //Prepared by keroles
2 #include <stdio.h>
3
4 //Prepared by Keroles
5 int main()
6 {
7     unsigned char x=0 ;
8
9     printf("Variable width control:\n");
10    printf("right-justified variable width: '%c'\n", 5, 'x');
11    printf("left-justified variable width : '%c'\n", -5, 'x');
12
13    int r = printf("Strings:\n");
14    printf("(the last printf printed %d characters)\n", r);
15
16    const char* s = "Hello";
17    printf("\t[%10s]\n\t[%-10s]\n\t[%*s]\n\t[%-10.*s]\n\t[%-*.*s]\n",
18          s, s, 10, s, 4, s, 10, 4, s);
19
20    printf("Characters:\t%c %%\n", 65);
21
22    printf("Integers\n");
23    printf("Decimal:\t%i %d %.6i %i %.0i %i %u\n", 1, 2, 3, 0, 0, 4, -1);
24    printf("Hexadecimal:\t%x %x %X %#x\n", 5, 10, 10, 6);
25    printf("Octal:\t%o %#o %#o\n", 10, 10, 4);
26
27    printf("Floating point\n");
28    printf("Rounding:\t%f %.0f %.32f\n", 1.5, 1.5, 1.5);
29    printf("Padding:\t%05.2f %.2f %5.2f\n", 1.5, 1.5, 1.5);
30    printf("Scientific:\t%E %e\n", 1.5, 1.5);
31    printf("Special values:\t1/0=%g\n", 0.0/0.0, 1.0/0.0);
32
33
34    printf("C trick:\t%d %d %d \n", ++x, x, x++);
35    printf("C trick:\t%d %d %d \n", x++, ++x, x);
36    return 0 ;
37 }
38

```



```
1 #include <stdio.h>
2 int main(){
3     int value1;
4     int value2;
5
6     printf("Please Enter Value: \n");
7     fflush(stdout);
8     scanf("%d + %d",&value1,&value2);
9     printf("the entered value 1 is %d |the entered value 2 is  %d",v
10 }
11
```

Problems Tasks Console Properties

<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\New folder (2)\text\Debug\text.exe (6/28/24, 6:12 PM)

Please Enter Value:

12 + 55

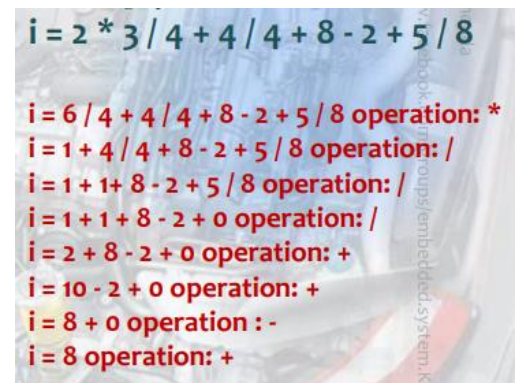
the entered value 1 is 12 |the entered value 2 is 55

# C - Operators

- **Unary operators**
  - ++ (increment)
  - (decrement)
  - ! (NOT)
  - ~ (compliment)
  - & (address of)
  - \* (dereference)
- **Binary operators** - arithmetic, logical and relational operators except !
- **Ternary operators** - The ? operator

## Hierarchy of Operations

Priority	Operators	Description
1 <sup>st</sup>	* / %	multiplication, division, modular division
2 <sup>nd</sup>	+ -	addition, subtraction
3 <sup>rd</sup>	=	assignment



$i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8$

$i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8$  operation: \*

$i = 1 + 4 / 4 + 8 - 2 + 5 / 8$  operation: /

$i = 1 + 1 + 8 - 2 + 5 / 8$  operation: /

$i = 1 + 1 + 8 - 2 + 0$  operation: /

$i = 2 + 8 - 2 + 0$  operation: +

$i = 10 - 2 + 0$  operation: +

$i = 8 + 0$  operation: -

$i = 8$  operation: +

## Arithmetic Operators

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.	$A-- = 9$

## Pre-increment and Post-increment in C

Increment operators are used to increase the value of a variable by 1. This operator is represented by the ++ symbol.

The increment operator can either increase the value of the variable by 1 before assigning it to the variable or can increase the value of the variable by 1 after assigning the variable.

Thus it can be classified into two types:

1. Pre-Increment Operator (a = ++x;)
2. Post-Increment Operator (a = x++;)

## Ex

```
1 #include <stdio.h>
2 int main(){
3     int b = 0;
4     int a = 4;
5
6     int x = a++; //x = 4 ;
7     int y = b++ + a; // y = 0 + 5;
8     int z = a + ++b; // z = 5 + 2;
9     int w = b+++a; // w = 2 + 5;
10    int s = a+++b; // s = 5 + 3;
11
12    printf("The value of \"a++\" is %d \n",x);
13    printf("The value of \"b++ + a\" is %d \n",y);
14    printf("The value of \"a + ++b\" is %d \n",z);
15    printf("The value of \"b+++a\" is %d \n",w);
16    printf("The value of \"a+++b\" is %d \n",s);
17 }
```

Problems Tasks Console Properties

<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\New folder (2)\text\Debug\text.exe (6/28/2

The value of "a++" is 4  
The value of "b++ + a" is 5  
The value of "a + ++b" is 7  
The value of "b+++a" is 7  
The value of "a+++b" is 8

## Token Generation and Lexical Analysis in Compilers

Lexical analysis is the first phase of a compiler, often referred to as the scanning phase. This stage involves the conversion of the source code into a sequence of tokens, which are the fundamental building blocks for syntactic analysis in the subsequent phases of compilation.

### Steps Involved in Lexical Analysis:

#### 1. Scanning:

- The lexical analyzer scans the input source code character by character.
- It groups sequences of characters into lexemes, which represent the smallest meaningful units in the source code.

#### 2. Token Generation:

- Once a lexeme is identified, the lexical analyzer converts it into a token.
- A token typically consists of a token name and an attribute value.

- **Example of Token Generation:**

```
c نسخ الكود  
  
int num = 10;
```

During lexical analysis, this line will be broken down into the following tokens:

1. ``int`` - Keyword
2. ``num`` - Identifier
3. ``=`` - Assignment operator
4. ``10`` - Numeric constant
5. ``;`` - Semicolon (delimiter)

### Summary:

- **Lexical Analysis** is the first phase of the compiler.
- The **lexical analyzer (scanner)** scans the entire source program to identify meaningful sequences of characters, which are then converted into tokens.
- **Tokens** are pairs consisting of a token-name and an attribute-value.
- The **scanner** facilitates the subsequent phases of compilation by generating a stream of tokens from the source code.

### Understanding the Line `int s = a+++b;`

This line of code can be confusing due to the sequence `a+++b`. The compiler interprets this as:

- `a++` (post-increment of `a`)
- `+` (addition operator)
- `b` (identifier)

#### Correct Token Stream:

Lexeme	Token-Name	Attribute-Value
int	KEYWORD	int
s	IDENTIFIER	s
=	OPERATOR	=
a	IDENTIFIER	a
++	OPERATOR	++
+	OPERATOR	+
b	IDENTIFIER	b
;	DELIMITER	;

## Relational Operators

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.

## 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
%=	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

# Logical Operators

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.

## Concept of short circuit in logical operator

In C programming, short-circuit evaluation is a concept associated with the logical operators && (logical AND) and || (logical OR). This concept means that the evaluation of logical expressions can be stopped as soon as the outcome is determined. This can improve efficiency and prevent potential errors by avoiding unnecessary evaluations.

## Short-Circuit Evaluation

### Logical AND (&&)

For the && operator, if the first operand evaluates to false (0), the overall expression cannot be true, regardless of the second operand. Therefore, the second operand is not evaluated.

Example:

```
c نسخ الكود
int a = 0;
int b = 1;

if (a && b) {
    // This block will not execute because 'a' is 0 (false)
}
```

In this example, since `a` is `0` (false), the expression `a && b` short-circuits and `b` is not evaluated.

## Logical OR (||)

For the || operator, if the first operand evaluates to true (non-zero), the overall expression is true, regardless of the second operand. Therefore, the second operand is not evaluated.

Example:

```
c نسخ الكود

int a = 1;
int b = 0;

if (a || b) {
    // This block will execute because 'a' is 1 (true)
}
```

In this example, since `a` is `1` (true), the expression `a || b` short-circuits and `b` is not evaluated.

## Practical Uses

### 1. Avoiding Null Pointer Dereference:

```
c نسخ الكود

char *ptr = NULL;
if (ptr && *ptr == 'A') {
    // The expression '*ptr == 'A'' will not be evaluated if 'ptr' is NULL
}
```

### 2. Preventing Division by Zero:

```
c نسخ الكود

int denominator = 0;
if (denominator != 0 && (10 / denominator) > 1) {
    // The expression '(10 / denominator)' will not be evaluated if 'denominator' is 0
}
```

### 3. Conditional Function Calls:

```
c نسخ الكود

int a = 1;
int b = 0;

if (a || someFunction()) {
    // 'someFunction()' will not be called because 'a' is true
}

if (b && someFunction()) {
    // 'someFunction()' will not be called because 'b' is false
}
```



## Bitwise Operators

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A   B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e., -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

p	q	p & q	p   q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

A = 0011 1100

B = 0000 1101

-----

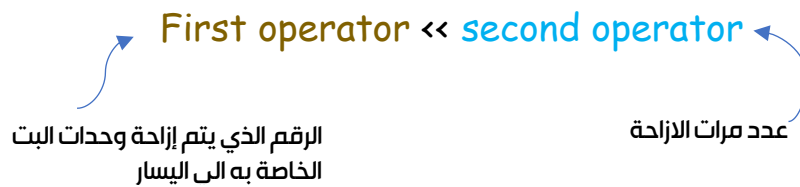
A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

## Shift left operator (<<)



✚ When bits are shifted left then trailing positions are filled with **zero**

## Syntax

**result = value << shift;**

- **value:** The integer value whose bits are to be shifted.
- **shift:** The number of positions to shift the bits to the left.
- **result:** The result of the left shift operation.

## Example

c

نسخ الكود

```
#include <stdio.h>

int main() {
    unsigned int value = 5;    // Binary: 0000 0101
    unsigned int shift = 2;
    unsigned int result = value << shift; // Binary: 0001 0100 (20 in decimal)

    printf("Value before shift: %u\n", value);
    printf("Value after shift: %u\n", result);

    return 0;
}
```

## Output:

bash

نسخ الكود

```
Value before shift: 5
Value after shift: 20
```

🚦 Left shifting is equivalent to multiplication by  $2^{\text{right operands}}$

## Example

Consider an integer value `x`:

c نسخ الكود

```
#include <stdio.h>

int main() {
    unsigned int x = 5; // Binary: 0000 0101
    unsigned int y = x << 1; // Binary: 0000 1010 (10 in decimal)
    unsigned int z = x << 2; // Binary: 0001 0100 (20 in decimal)

    printf("x = %u\n", x);
    printf("x << 1 = %u\n", y); // Equivalent to 5 * 2
    printf("x << 2 = %u\n", z); // Equivalent to 5 * 4

    return 0;
}
```

## Output

bash نسخ الكود

```
x = 5
x << 1 = 10
x << 2 = 20
```

## Explanation

- `x = 5` (binary: `0000 0101`)
- `x << 1` shifts the bits one position to the left: `0000 1010` (binary for 10)
- `x << 2` shifts the bits two positions to the left: `0001 0100` (binary for 20)

## Mathematical Equivalent

- `5 << 1` is equivalent to  $5 * (2^1) = 5 * 2 = 10$
- `5 << 2` is equivalent to  $5 * (2^2) = 5 * 4 = 20$

## Important Points

**Overflow:** Be cautious with large values, as left shifting can lead to overflow if the resulting value exceeds the maximum value that can be stored in the data type.

```
c نسخ الكود  
  
unsigned int a = 1 << 31; // Valid for 32-bit unsigned int  
unsigned int b = 1 << 32; // Undefined behavior if 'unsigned int' is 32 bits
```

```
Problems Tasks Console × Properties  
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\New folder (2)\text\Debug\text.exe (6/28/24  
The value of "unsigned int a = 1 << 31; " is -2147483648  
The value of "unsigned int b = 1 << 32;" is 0
```

**Signed Integers:** When left shifting signed integers, be careful with the sign bit. If the sign bit is shifted into the value, the result might become negative or undefined, depending on the implementation.

```
c نسخ الكود  
  
int c = -1;  
int d = c << 1; // Result depends on how the implementation handles the sign bit
```

```
Problems Tasks Console × Properties  
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah Ghazy\Desktop\New folder (2)\text\Debug\text.exe (6/28/24  
The value of "int d = c << 1;" is -2
```

## Comma ( , ) operator

the comma operator (,) is a binary operator that evaluates two expressions and returns the value of the second expression. This operator is typically used in situations where multiple expressions need to be evaluated in a single statement.

### Syntax

`result = (expression1, expression2);`

- expression1: The first expression to be evaluated.
- expression2: The second expression to be evaluated, whose value is returned.

### الاستخدام

1. يستخدم لفصل المتغيرات عن بعضها البعض

```
int a = 3, b = 4, c = 7;
```

2. اذا تم إعطاء أكثر من قيمة للمتغير بين اقواس `a = (3,4,5,8)` فانه يختار القيمة الاولى من اليمين ولكن يقوم بتنفيذ كل الأوامر التي داخل القوس يعطي خطأ اذا كتبته `int a=3,4,5,8` لانه يبقي شايها `int a = 3, int 4, int 5, int 8` عشان كذا نحطها في اقواس

```
#include <stdio.h>

int main() {
    int a = 1, b = 2;
    int result;

    result = (a = a + 1, b = b + 2);

    printf("a = %d, b = %d, result = %d\n", a, b, result); // Output: a = 2, b = 4, result = 4

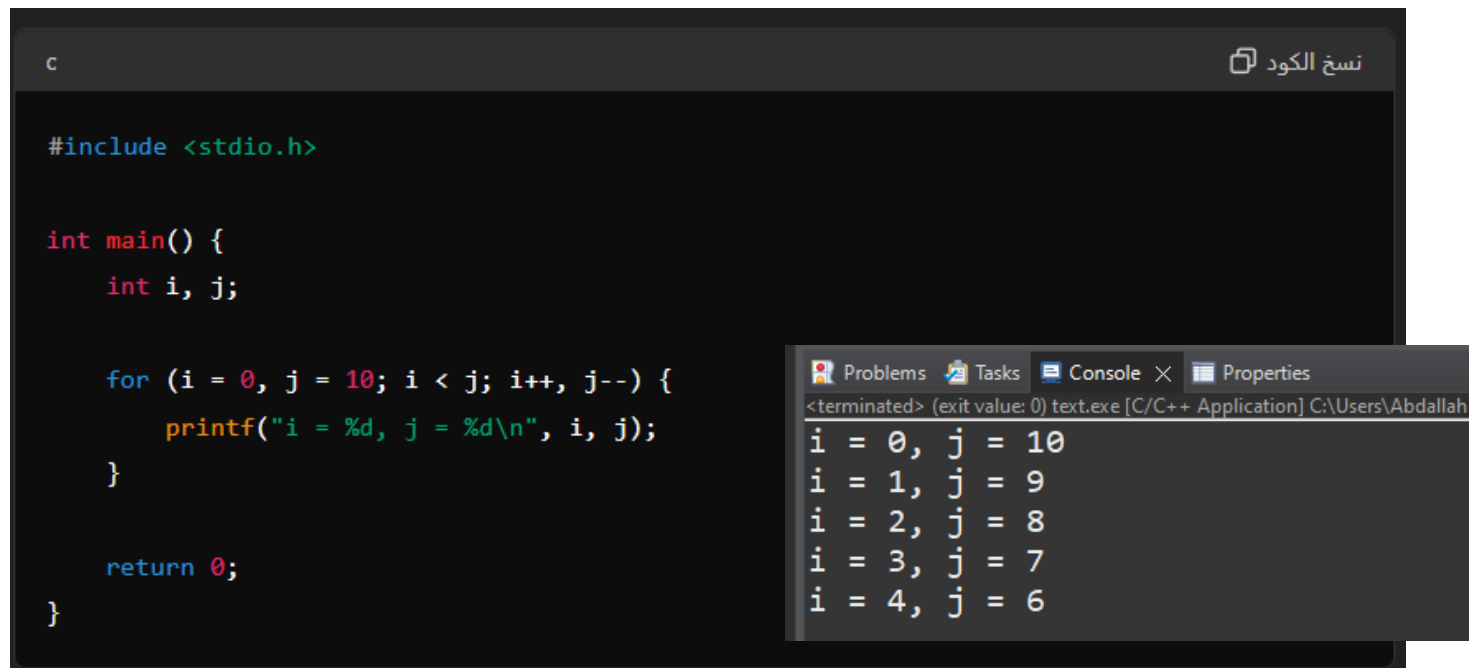
    return 0;
}
```

### In this example:

- `a = a + 1` is evaluated first, setting `a` to 2.
- `b = b + 2` is evaluated next, setting `b` to 4.
- The value of `b` (which is now 4) is assigned to `result`.

## Using Comma Operator in For Loop

The comma operator is commonly used in for loops to handle multiple expressions within the loop header.



The screenshot shows a C code editor with a dark theme. The code defines a `main` function that declares two integers, `i` and `j`. A `for` loop is used with the header `(i = 0, j = 10; i < j; i++, j--)`. Inside the loop, `printf` prints the values of `i` and `j`. The loop terminates when `i` is no longer less than `j`. To the right of the code editor, a console window is open, displaying the output of the program: five lines showing the values of `i` and `j` at each iteration, from `i = 0, j = 10` down to `i = 4, j = 6`. The console window also shows the program's exit status as 0.

```
c
#include <stdio.h>

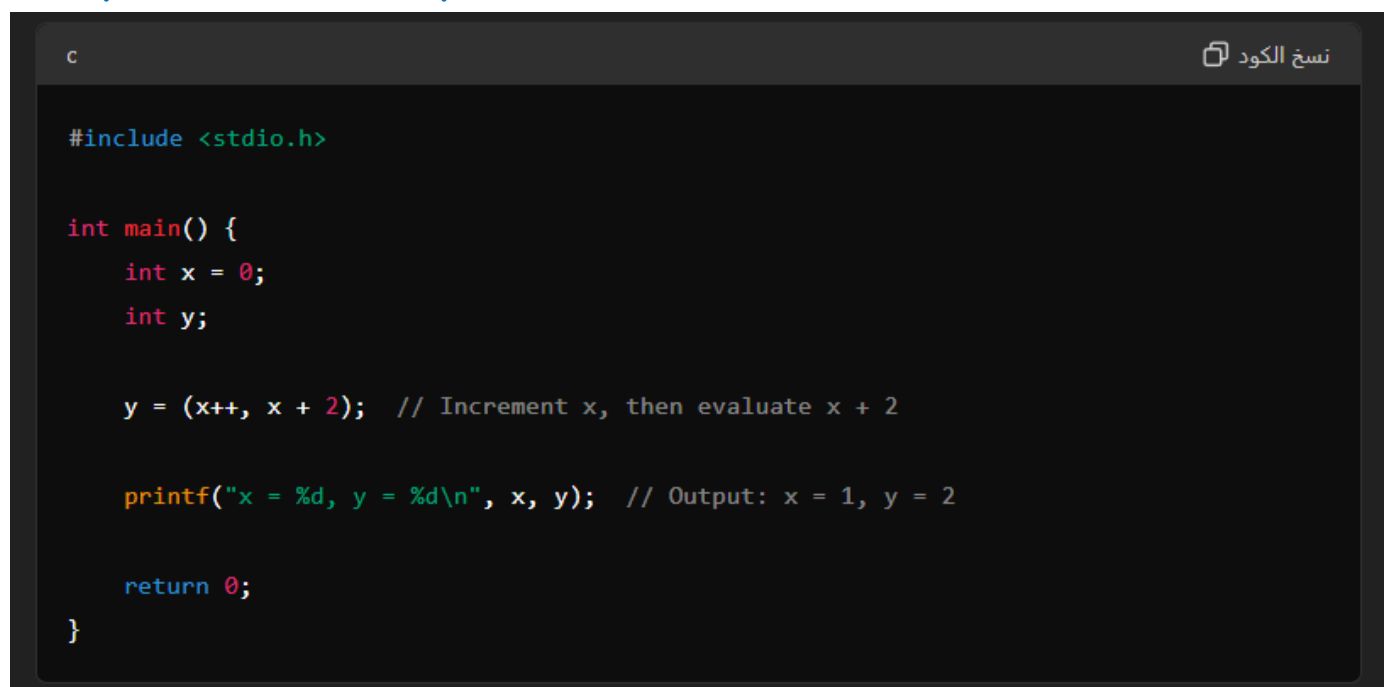
int main() {
    int i, j;

    for (i = 0, j = 10; i < j; i++, j--) {
        printf("i = %d, j = %d\n", i, j);
    }

    return 0;
}
```

Problems Tasks Console Properties  
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\Users\Abdallah  
i = 0, j = 10  
i = 1, j = 9  
i = 2, j = 8  
i = 3, j = 7  
i = 4, j = 6

## Example of Combined Expressions



The screenshot shows a C code editor with a dark theme. The code defines a `main` function that declares two integers, `x` and `y`. `x` is initialized to 0. Then, a single statement `y = (x++, x + 2);` is used, which increments `x` and assigns the result of `x + 2` to `y`. A `printf` statement prints the values of `x` and `y`, showing the output `x = 1, y = 2`. The program then returns 0.

```
c
#include <stdio.h>

int main() {
    int x = 0;
    int y;

    y = (x++, x + 2); // Increment x, then evaluate x + 2

    printf("x = %d, y = %d\n", x, y); // Output: x = 1, y = 2

    return 0;
}
```

# Operators Precedence in C

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
Comma	,	Left to right

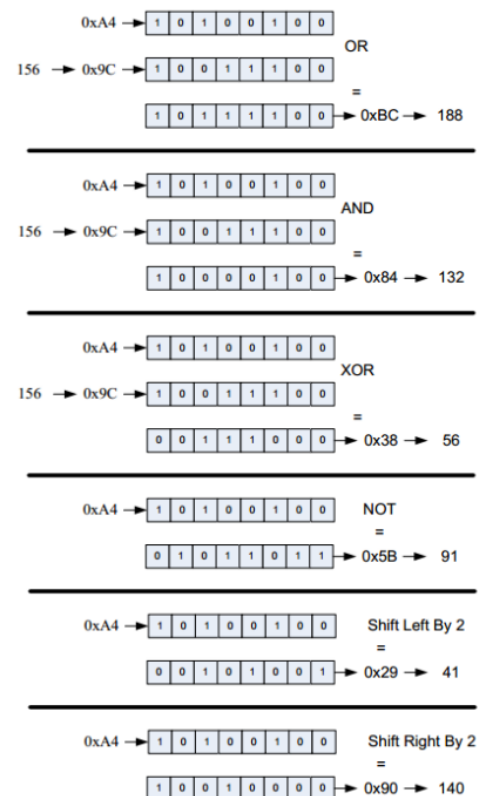
## Example: Using Conditions

```
#include "stdio.h"
#include "math.h"
void main()
{
    int a = 9;
    int b = 8;
    int c = 12;
    printf("%d\r\n", a>b); //prints 1
    printf("%d\r\n", b>c); //prints 0
    printf("%d\r\n", a<=9); //prints 1
    printf("%d\r\n", a!=9); //prints 0
    printf("%d\r\n", (a-b)>(c-b)); //prints 0
    printf("%d\r\n", a>b && c>b); //prints 1
    printf("%d\r\n", a>b && c<b); //prints 0
    printf("%d\r\n", a>b || c<b); //prints 1
    printf("%d\r\n", !(a<b)); //prints 1
    printf("%d\r\n", 3 && 0); //prints 0
    printf("%d\r\n", -15 || 0); //prints 1
    printf("%d\r\n", !(-15)); //prints 0
}
```

# Mathematical and Logical Expressions

Following examples provides some specific C expressions:

C Expression	Meaning
<code>X = X + 9;</code>	Calculate X+9 then stores the result in X
<code>X++;</code>	Add one to X
<code>X--;</code>	Subtract one from X
<code>X = 10;</code> <code>Y = 5;</code> <code>X = X + Y++;</code>	Add X+Y → 15 then increment Y → 6 Store 15 in X
<code>X = 10;</code> <code>Y = 5;</code> <code>X = X + ++Y;</code>	Increment Y → 6 Add X+Y → 16 Store 16 in X
<code>unsigned char X = 0xA4;</code> <code>unsigned char Y = 156;</code> <code>unsigned char Z = X Y;</code>	Calculate the X OR Y → 0xBC (188)
<code>unsigned char X = 0xA4;</code> <code>unsigned char Y = 156;</code> <code>unsigned char Z = X&amp;Y;</code>	Calculate the X AND Y → 0x84 (132)
<code>unsigned char X = 0xA4;</code> <code>unsigned char Y = 156;</code> <code>unsigned char Z = X^Y;</code>	Calculate the X XOR Y → 0x38 (56)
<code>unsigned char X = 0xA4;</code> <code>unsigned char Z = ~X;</code>	Calculate the (NOT X) → 0x5B (91)
<code>unsigned char X = 0xA4;</code> <code>unsigned char Z = X&gt;&gt;2;</code>	Calculate the X shifted by two bits to right → 0x29 (41)
<code>unsigned char X = 0xA4;</code> <code>unsigned char Z = X&lt;&lt;2;</code>	Calculate the X shifted by two bits to left → 0x90 (144)





# Identifiers

Identifiers are the names that are given to various program elements such as variables, symbolic constants and functions.

✚ **Identifier can be freely named, the following restrictions.**

- Alphanumeric characters ( a ~ z , A ~ Z , 0 ~ 9 ) and half underscore ( \_ ) can only be used.
- The first character of the first contain letters ( a ~ z , A ~ Z ) or half underscore ( \_ ) can only be used.

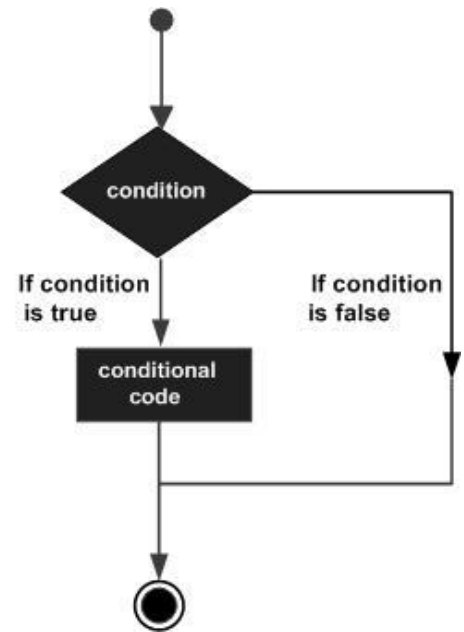
✚ **Here are the rules you need to know:**

- Identifier name must be a sequence of letter and digits, and must begin with a letter
- The underscore character ( '\_' ) is considered as letter.
- Names shouldn't be a keyword (such as int, float, if, break, for etc)
- Both upper-case letter and lower-case letter characters are allowed. However, they're not interchangeable.
- No identifier may be keyword.
- No special characters, such as semicolon ,period ,blank space, slash or comma are permitted

## If Statement in C Programming

### Syntax

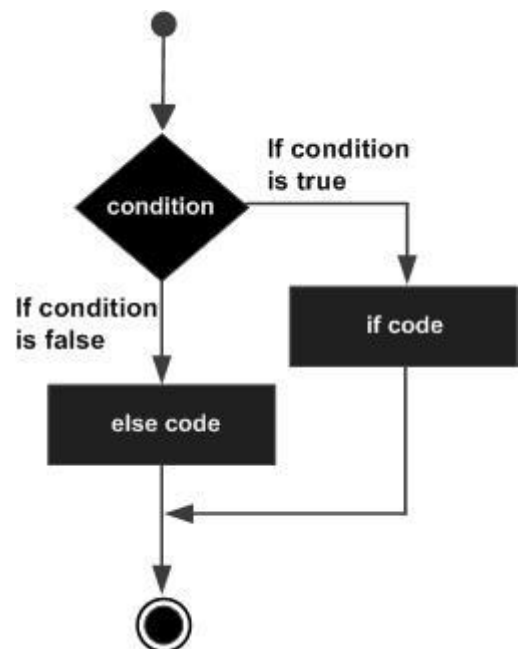
```
if (Boolean expr){  
    expression;  
    . . .  
}
```



## If...else Statement in C Programming

### Syntax

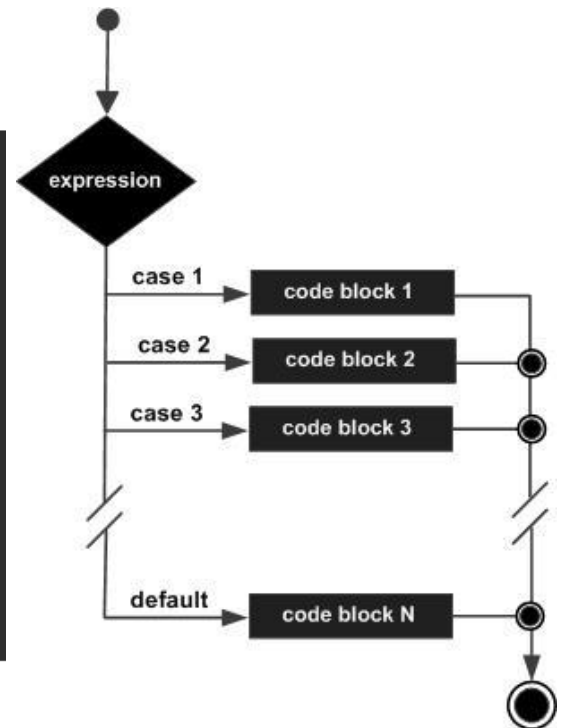
```
if (Boolean expr){  
    expression;  
    . . .  
}  
else{  
    expression;  
    . . .  
}
```



# Switch Statement in C Programming

## Syntax

```
switch(expression) {  
  
    case constant-expression :  
        statement(s);  
        break; /* optional */  
  
    case constant-expression :  
        statement(s);  
        break; /* optional */  
  
    /* you can have any number of case statements */  
    default : /* Optional */  
        statement(s);  
}
```



## Differences Between if and switch Statements

### if Statement

1. The if statement can handle complex expressions, including relational, logical, and arithmetic operations. Supports conditions involving any data type (integer, floating-point, pointers, etc.).
2. May be slower compared to switch in scenarios involving multiple conditions because each condition is evaluated until a match is found.
3. Conditions are evaluated sequentially from top to bottom.

### switch Statement

1. faster in execution time for cases involving a large number of discrete values.
2. Utilizes a jump table or binary search for faster lookup.
3. The switch expression must evaluate to an integer or an enumeration type.
4. Does not support floating-point numbers or complex conditions.
5. Faster for large numbers of discrete values because of its optimized handling via jump tables or binary searches.
6. Less flexible compared to if statements in terms of condition complexity.

7. لا يسمح لك بوضع قيم مكررة

```
case 1:
    printf("x is 1\n");
    break;
case 1:
    printf("x is 2\n");
    break;
```

8. لا يمكن كتابة المعادلة داخل switch

```
switch (x+a*b) {
```

9. يمكن كتابة معادلة بدل تسمية الحالة

```
case 1+2*4:
```

10. يتم تنفيذ كل ال Cases في البداية و بعدها ينفذ ال default حتي لو مكتوبه في بداية ال switch

```
switch (x) {
default:
    printf("x is not 1, 2, or 3\n");
```

# The ?: Operator in C Programming

**condition ? expression1 : expression2;**

- condition: The expression to be evaluated. If it is true (non-zero), expression1 is evaluated and returned.
- expression1: The expression evaluated and returned if condition is true.
- expression2: The expression evaluated and returned if condition is false.

## Example

Here's a simple example to illustrate how the ternary operator works:

```
c نسخ الكود

#include <stdio.h>

int main() {
    int a = 10;
    int b = 20;
    int max;

    // Using the ternary operator to find the maximum value
    max = (a > b) ? a : b;

    printf("The maximum value is %d\n", max);

    return 0;
}
```

## Explanation

- Condition: ``a > b``
- Expression1: ``a`` (if ``a > b`` is true)
- Expression2: ``b`` (if ``a > b`` is false)

## Nested Ternary Operators

```
c نسخ الكود

#include <stdio.h>

int main() {
    int x = 5;
    int y = 10;
    int z = 15;
    int max;

    // Using nested ternary operators to find the maximum value
    max = (x > y) ? ((x > z) ? x : z) : ((y > z) ? y : z);

    printf("The maximum value is %d\n", max);

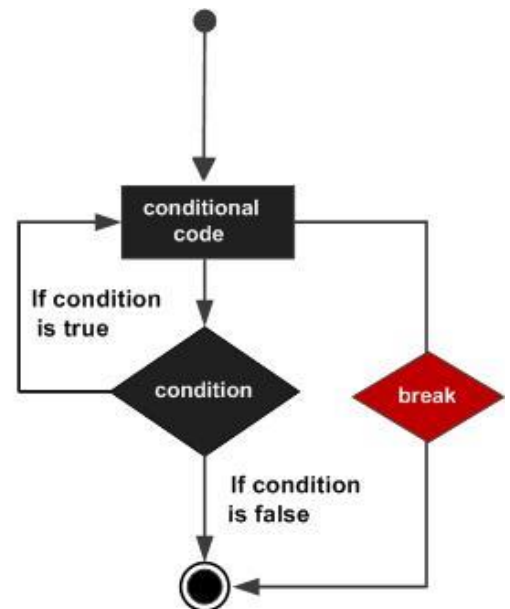
    return 0;
}
```

Problems Tasks Console X Properties  
<terminated> (exit value: 0) text.exe [C/C++ Application] C:\  
The maximum value is 15

## The Break Statement in C Programming

In C, the break statement is used in switch-case constructs as well as in loops. When used inside a loop, it causes the repetition to be abandoned.

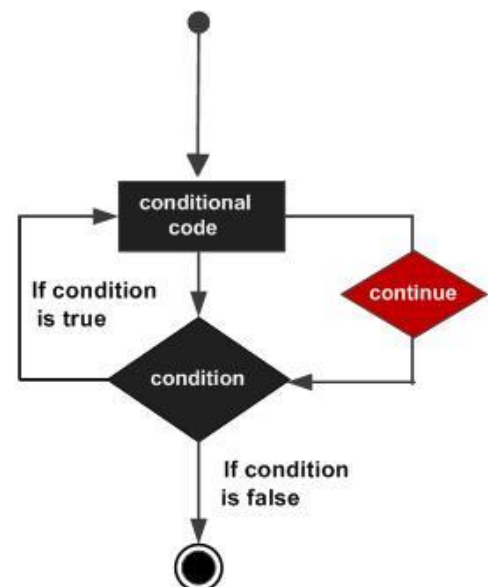
```
while(condition1){  
    . . .  
    . . .  
    if(condition2)  
        break;  
    . . .  
    . . .  
}
```



## The Continue Statement in C Programming

In C, the continue statement causes the conditional test and increment portions of the loop to execute.

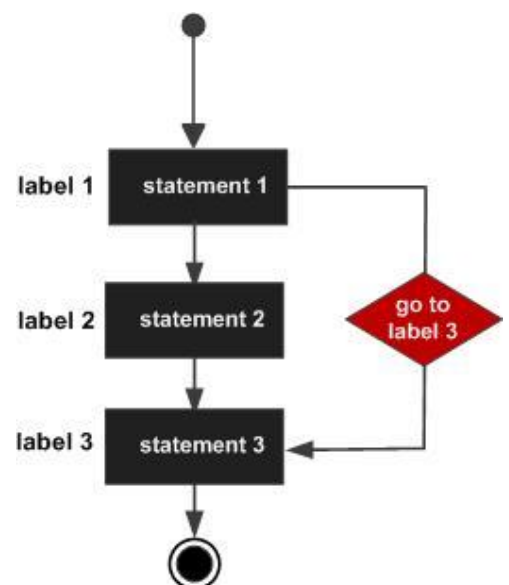
```
while (expr){  
    . . .  
    . . .  
    if (condition)  
        continue;  
    . . .  
}
```



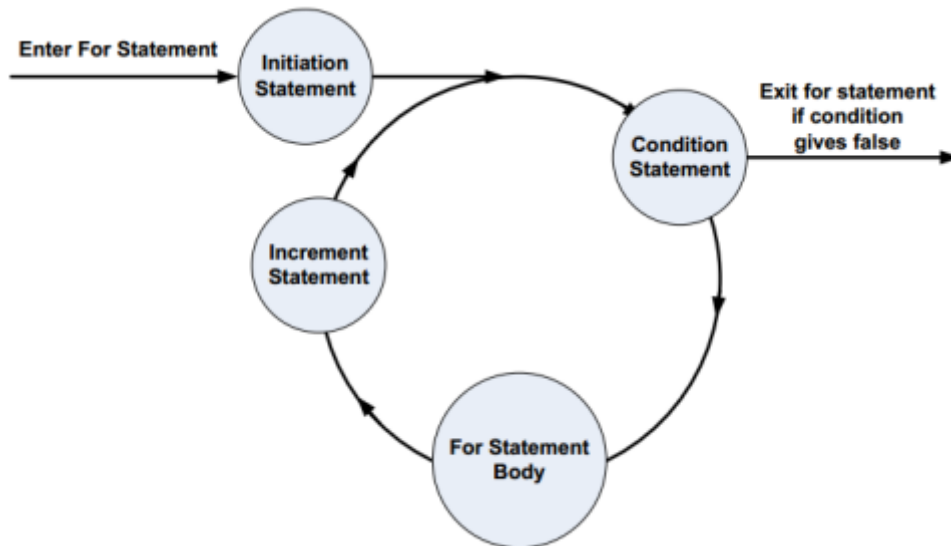
## The goto Statement in C Programming

C also has a goto keyword. You can redirect the program flow to any labelled instruction in the program.

```
goto label;  
..  
.  
label: statement;
```



# for Statement



## Syntax of for Loop

```
for (init; condition; increment){  
    statement(s);  
}
```

Using the for loop in C is very useful when you need to execute a block of code a specific number of times. Always be mindful of the counter's size to ensure it has enough space to cover all required iterations without causing an overflow.

## Infinite Loop in C

</>

Open Compiler

Edit & Run

```
#include <stdio.h>  
  
int main (){  
  
    for( ; ; ){  
        printf("This loop will run forever. \n");  
    }  
  
    return 0;  
}
```

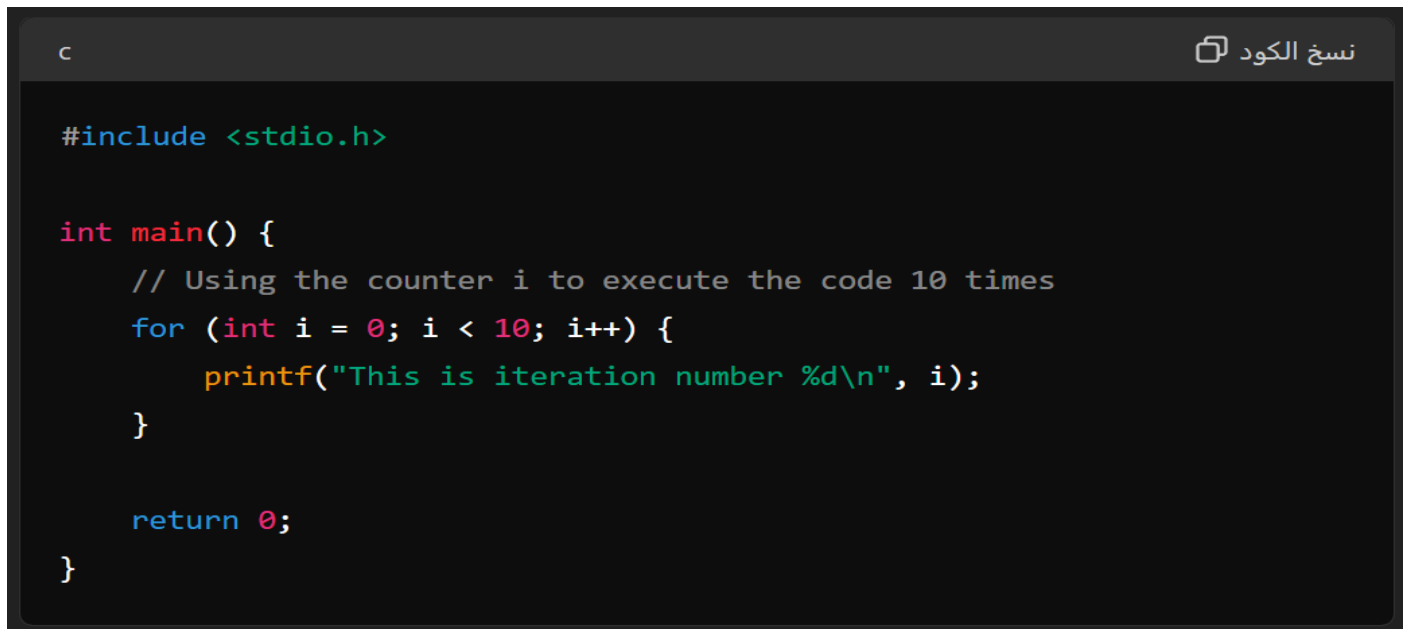
## C - While Loop

The while loop in C is ideal for situations where the number of iterations is not known in advance. This loop continues to execute the code block as long as the condition remains true.

### Syntax of C while Loop

```
while(expression){  
    statement(s);  
}
```

### Infinite Loop Example



```
c  
#include <stdio.h>  
  
int main() {  
    // Using the counter i to execute the code 10 times  
    for (int i = 0; i < 10; i++) {  
        printf("This is iteration number %d\n", i);  
    }  
  
    return 0;  
}
```

## Do-While Loop in C

```
do {  
    statement(s);  
} while(condition);
```

The do-while loop in C is used when you need to execute a block of code at least once and then repeat the execution as long as a given condition is true. This loop is similar to the while loop, but with a key difference: the condition is checked after the execution of the loop body, ensuring that the code inside the loop runs at least once.



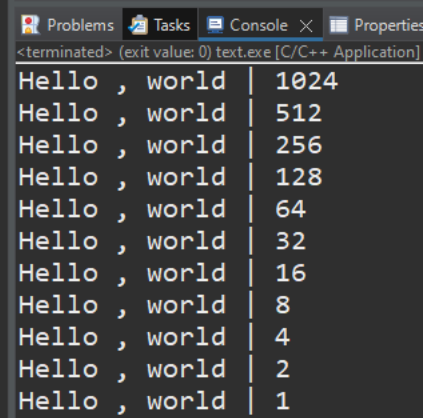
## Loops → solved problems

Ex output ?

```
#include <stdio.h>

int main() {
    int i = 1024;
    for ( ; i ; i>>=1){
        printf("Hello , world | %d \n",i);
    }

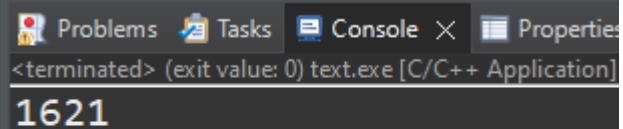
    return 0;
}
```



Output	i
Hello , world	1024
Hello , world	512
Hello , world	256
Hello , world	128
Hello , world	64
Hello , world	32
Hello , world	16
Hello , world	8
Hello , world	4
Hello , world	2
Hello , world	1

Ex output ?

```
1 #include <stdio.h>
2
3 int main() {
4     int i;
5     for(i=0; i<20 ; i++){
6         switch(i){
7             case 0: i+= 5;
8             case 1: i+= 2;
9             case 5: i+= 5;
10            default : i+= 4;
11        }
12        printf("%d",i);
13    }
14    return 0;
15 }
16
```



Output
1621

## Special programs

1. Write a program to check whether a number is an Armstrong number or not.
2. Write a program to check whether a number is a strong number or not.
3. Write a program to  $2^3$  and  $2^{-3}$
4. Write a program to check whether a year is a leap year or not.
5. Write a program to add two numbers without using + operator
6. Write a program convert binary to decimal
7. Swapping Two Variables Without a Temporary Variable
8. Finding the Maximum of Two Numbers Using Ternary Operator
9. Checking if a Number is Even or Odd
10. Printing a Number in Binary Format
11. Write a program to find factorial of the given number.
12. Write a program to swap two numbers using a temporary variable.
13. Write a program to swap two numbers without using a temporary variable
14. Write a program to swap two numbers using bitwise operators.
15. Write a program to find the greatest of three numbers.
16. Write a program to find the greatest among ten numbers.
17. Write a program to check whether the given number is a prime.
18. Write a program to check whether the given number is a palindromic number
19. Write a program to check whether the given string is a palindrome
20. Write a program to generate the Fibonacci series.
21. Write a program to print "Hello World" without using semicolon anywhere in the code.
22. Write a program to print a semicolon without using a semicolon anywhere in the code.

ملحوظة ان if بتنفذ ال بداخلها الأول حتي لو كان x++  
هيروح يضيف 1 علي x مش هينزل وبعدين يضيف

Q23) \*

```
#include<stdio.h>
void main() {
    int x=-1,y=-1;
    if(++x==++y)
        printf("R.T. Ponting");
    else
        printf("C.H. Gayle");
}
```

- ☐ R.T Ponting
- ☐ C.H. Gayle
- ☐ Warning: x and y are assigned a value that is never used
- ☐ Warning: Condition is always true
- ☒ Compilation error

