Cheat Sheet for C

## Data Types

Data types of all variables must be known (specified at declaration) at compilation time. The main categories of data types in C includes,

### Basic / Built-in:

* + void – absence of type
    - int main(void)
  + char – ASCII characters
  + int – integer, signed
  + float – floating point

Data types defined in **stdint.h** have guaranteed fixed data size.

|  |  |  |  |
| --- | --- | --- | --- |
| **Signed** | **Unsigned** | **Data Size (Bytes)** | **Min & Max Value** |
| int8\_t | int8\_t | 1 | -27 to (27 - 1) / 0 to (28 – 1) |
| int16\_t | uint16\_t | 2 | -215 to (215 - 1) / 0 to (216 – 1) |
| int32\_t | uint32\_t | 4 | -231 to (231 - 1) / 0 to (232 – 1) |
| int64\_t | uint64\_t | 8 | -263 to (263 - 1) / 0 to (264 – 1) |

### Derived

* + array, pointers, references, etc.

### User-defined

* + structure, union, enumeration, etc.

There are two forms of data type conversion,

### Implicit (automatic)

For expressions with mixed data types, compiler automatically promotes lower type to higher type to avoid loss of data. Note that information lose is possible for implicit conversions (eg. int to unsigned). Conversion is based on the following Data Type Orde (lowest to highest),

bool > char > short > unsigned short > int > unsigned > long > unsigned long > long long > unsigned long long > float > double …

### Explicit (type casting)

Explicit is user-defined to make a convert to a specific data type. Results can become unpredictable as per the following example. Use the following C syntax,

|  |
| --- |
| *(****type****) expression* |

Example:

|  |
| --- |
| *float x = 1.99;*  *int sum = (int) x + 2;*  *printf(“sum = %d”, sum);*  Output will be …  sum = 3 |

## Variables and Constants

Have meaningful naming conventions which indicates its purpose concisely using only alphanumeric characters and underscore, not forgetting all variables are case sensitive.

Do not begin with a number and some keywords / reserved words are not allowed. Note that names begin with ‘\_’ are for reserved identifiers while a trailing ‘\_t’ are reserved for standard types.

### Variables

Usually in lowercase

### Constants

Usually in uppercase. Declaration of a **constant** is similar to a **variable** by prepending ***const*** keyword and the value specified, eg.

*const int FYP = 2024*

Alternatively, declare a constant using preprocessor directive ***#define***. It creates a macro which is the association of an identifier or parameterised identifier with a token string, eg.

#define FYP 2024

## Operators

|  |  |  |
| --- | --- | --- |
| **Group** | **Operator(s)** | **Remarks** |
| Arithmetic | + - \* / % ++ -- | Addition, subtraction, multiplication, division, modulo, increment, decrement |
| Comparison / Relational | < <= > >= == != | Less than, equal or less than, more than, equal or more than, equals to, not equals to |
| Logical | ! && || | NOT, AND, OR  To combine 2 or more Boolean expressions |
| Bitwise | ~ << >> & ^ | | NOT, left shift, right shift, AND, XOR, OR  Manipulate data, perform bit-level operations on operands |
| Assignment | = += -= \*= /= %= ^= |= <<= >>= | Assign/Set value to a named variable |
| Miscellaneous | () [] . -> (type) \* & sizeof ?: , |  |

### **sizeof** operator

A **compile-time** (when source code is converted to binary code, as compared to run-time) unary operator (operator that perform operations on a single operand to produce a new value) to compute size of its operand. Result of sizeof is unsigned integral type usually denoted by size\_t, can be applied to any data type (integer, floating point, etc.)

Syntax:

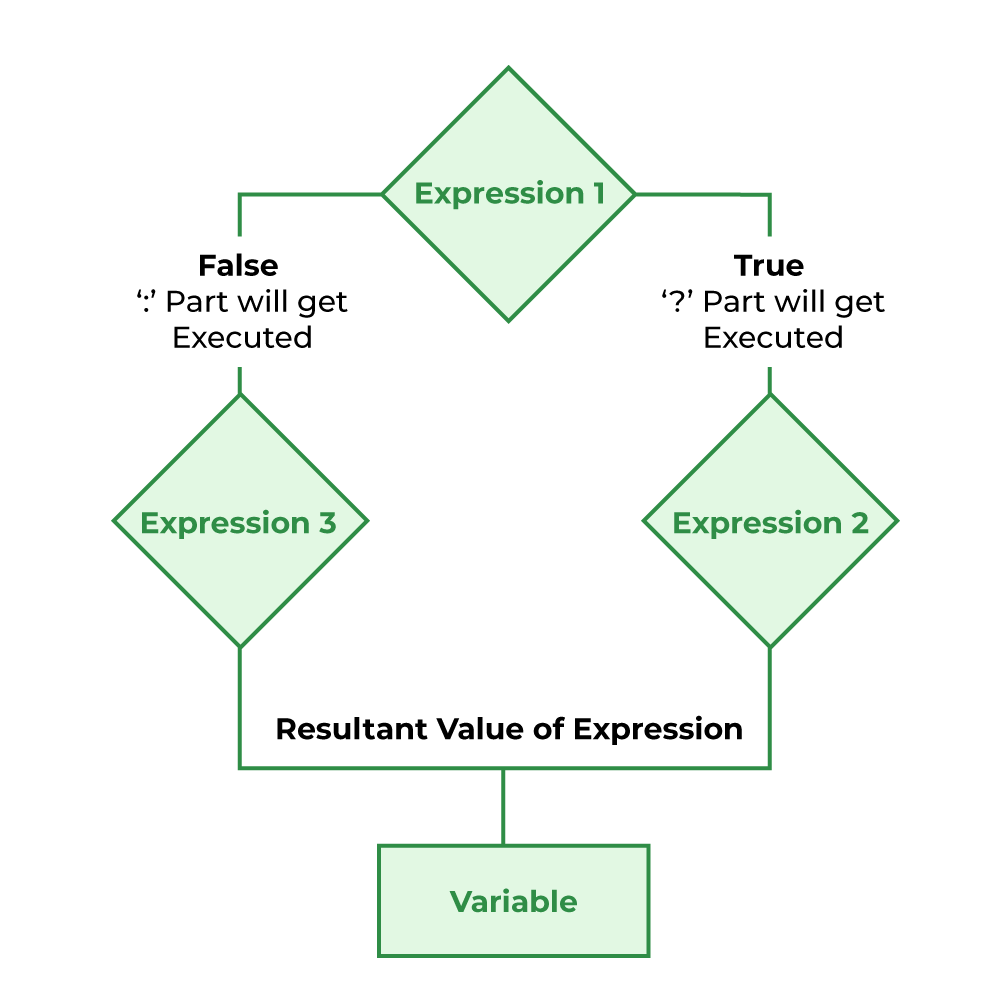
|  |
| --- |
| ***sizeof****(expression);* |

### **?:** (conditional / ternary) operator

Conditional operator similar to if-else statements that takes less space.

Syntax:

|  |
| --- |
| *variable = expression\_1* ***?*** *expression\_2* ***:*** *expression\_3;*  *OR*  *variable = (condition)* ***?*** *expression\_2* ***:*** *expression\_3;*  OR  *(condition)* ***?*** *(variable = expression\_2)* ***:*** *(variable = expression\_3);* |



## Selection

Selections or conditional are used to determine a different set of steps to execute based on Boolean expression(s).

### **if , else if , else**

* Use **if** to specify a block of code only to be executed if a specified condition is true (condition\_1)
* [Optional] Use **else if** to specify a new / additional condition (condition\_2) to test after condition\_1 is false.
* [Optional] Use **else** to specify a block of code to be executed if both conditions are false.

*Syntax:*

|  |
| --- |
| ***if*** *(condition\_1)*  *{*  *// code block to be executed if condition\_1 is true*  *}*  ***else******if*** *(condition\_2)*  *{*  *// code block to be executed if condition\_1 is false and condition\_2 is true*  *}*  ***else***  *{*  *// code block to be executed if both conditions are false*  *}* |

### **switch** , **case**

Use **switch**-**case** to specify many alternative blocks of code to be executed instead of using multiple **if**-**else** statements.

Syntax:

|  |
| --- |
| **switch** (expression)  {  **case** a:  //block to be executed if a is true  break; //ends this case & passes to next statement after the switch-case construct  .  . // multiple case blocks to be tested  .  **case** z:  //block to be executed if z is true  break;  **default:**  //block to be executed if no case matches  } |

## Iteration

**Iteration** or loops are used to repeatedly execute a block of code based on either a count or condition.

You can ‘exit’ any loop with the **break** keyword or by using **if-else** and continue immediately with the program after the loop.

Also, using the **continue** keyword within any loop to jump to the end of the code block, skipping statements between the keyword and end of the loop code block, the loop then continues normally

### **for**

To repeatedly execute a block of code based on **count**

Syntax:

|  |
| --- |
| **for (** *initial condition ; test ; update* **)**  {  //code block to be executed as long as test result is true  } |

Initial condition: Expression to initialise a local loop variable (eg. *int count = 1* )

Test: Checks that the loop variable meets the test condition to repeatedly execute the block of code, otherwise it ends the loop (eg. *count <= 10* )

Update: Loop variable automatically increases / decreases based on this expression (eg. *i++* )

### **while**

A simpler **for**-loop with only a single test

Syntax:

|  |
| --- |
| **while(** *test* **)**  {  //code block to be executed while test result is true  //usually ‘update’ within the code block  } |

### **do**-**while**

An ‘inverted’ **while** loop that execute the code block once first before test.

Syntax & Example:

|  |
| --- |
| int x = 5;  **do**  {  printf(“%d \n”, x);  x = x – 2;  } **while(** *x >= 0* **);** |

Result will be,

|  |
| --- |
| 5 // code block executed once first, ‘update’ is done  3 // code block executed as test (x >= 0) is true and x = 3  1 // code block executed as test (x >= 0) is true and x = 1 |

## Function