C Programming Lecture 4

Wednesday, 12 June 2024 8:17 PM

Type Casting in C

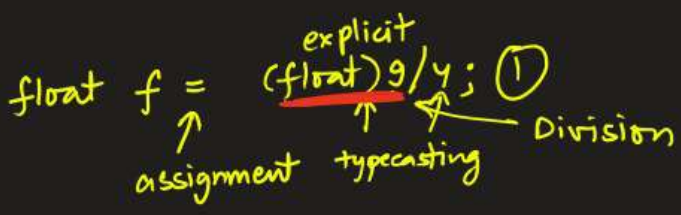
Data types are crucial in determining the nature and behavior of variables in the realm of programming. Typecasting allows us to convert one data type into another. This procedure is called type casting, and the C programming language offers it as a useful tool.

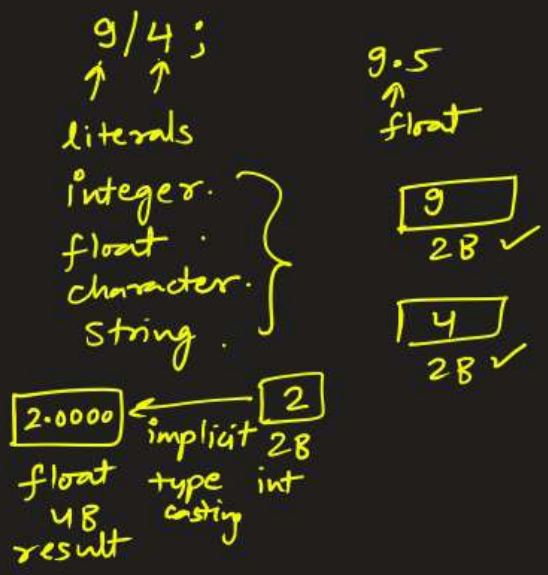
The procedure of changing a variable's data type is known as ***type casting***. It can be helpful in a variety of situations, for as when processing ***user input***, doing ***mathematical calculations***, or ***interacting*** with other libraries that need data types. In C language, we use cast operator for typecasting which is denoted by (type).

Syntax:

```
(type)value;
```

```
float f = 9/4;
printf("f: %f\n", f) ; //Output: 2.000000

With Type Casting:
float f = (float)9/4;
printf("f: %f\n", f) ; //Output: 2.250000
```





typecast > division > assignment

float f = 9/(float)4; (2)

float f = (float)9/(float)4; (3)

float f = (float)(9/4);
↑
explicit

9 int 4 int 2 int → 2.000 float f

```c
int main() {
    float f1 = 9/4;
    float f2 = (float)9/4;
    float f3 = 9/(float)4;
    float f4 = (float)9/(float)4;
    float f5 = (float)(9/4);

    printf("%f, %f, %f, %f, %f",
           f1, f2, f3, f4, f5);

    return 0;
}
```

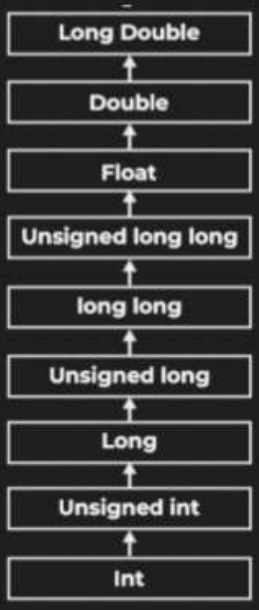
Implicit Casting

When the compiler ***automatically transforms*** the data from one type to another, it is referred to as ***implicit casting*** or ***automated type conversion***. This casting is based on a set of guidelines that outline how various kinds of data can coexist.

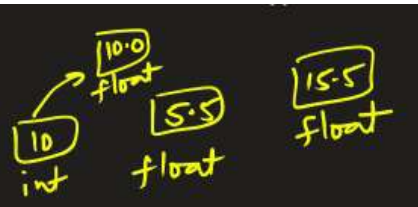
Implicit casting doesn't require any special syntax because the compiler takes care of it.

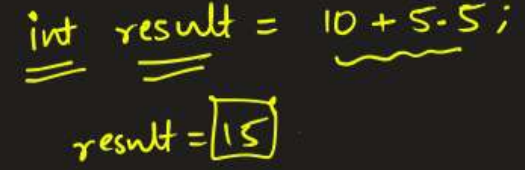
Conversion of lower data type to higher data type will occur automatically.

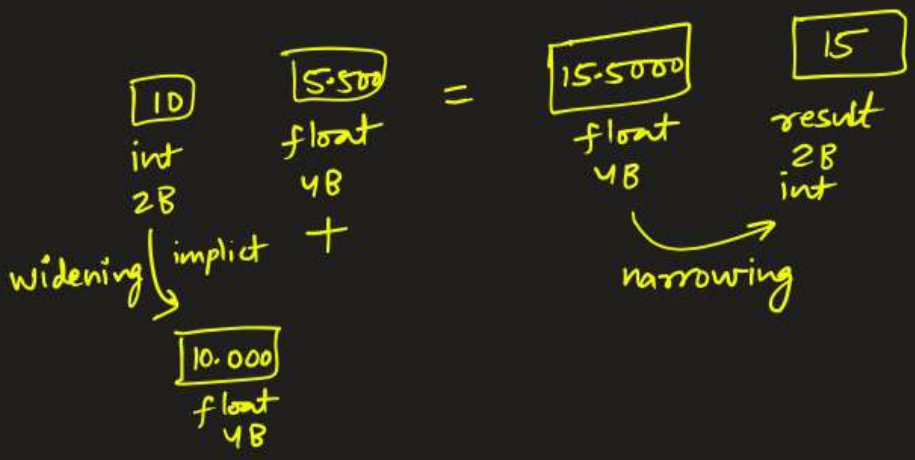
Integer promotion will be performed first by the compiler. After that, it will determine whether two of the operands have different data types.



```c
int main() {
    int num1 = 10;
    float num2 = 5.5;
    float result;

    result = num1 + num2; // Implicit cast from int to float

    printf("The result is: %f\n", result);

    return 0;
}
```







Explicit Casting

Using the ***cast operator, explicit casting*** entails explicitly changing one data type to another. It needs explicit instructions in the code and allows the programmer control over type conversion.

```c
int main() {
    float num1 = 15.6;
    int num2;

    num2 = (int) num1; // Explicit cast from float to int

    printf("The result is: %d\n", num2);

    return 0;
}
```

Narrowing Conversion

When a ***value*** is converted to a ***data type*** with a ***narrower range*** or ***precision***, it is said to be ***narrowing transformed***, which may result in data loss. If it is not handled appropriately, it may lead to unexpected behavior and needs explicit casting.

Example:

```c
int main() {
    double num1 = 1234.56789;
    int num2;

    // Narrowing conversion from double to int
    num2 = (int) num1;
    printf("The result is: %d\n", num2);

    return 0;
}
```

Result: 1234

Widening Conversion

When a value is converted to a ***data type*** with a wider range or greater accuracy, this process is called ***widening conversion***. Since it occurs implicitly, casting is not necessary.

Example:

```c
int main() {
    int num1 = 10;
    double num2;

    // Widening conversion from int to double
    num2 = num1;
    printf("The result is: %f\n", num2);

    return 0;
}
```

Result: 10.00

Advantages of Type Casting

Its numerous advantages make type casting in C a useful feature for programmers. Let's examine a few of the main advantages:

- **Precision Control:** ✓
  Type casting enables programmers to exert precise control over the accuracy of calculations. They can alter the ***decimal places***, eliminate the ***fractional component***, or get more precise answers by converting variables to various data types.

- **Data Compatibility:** ✓
  When using ***external libraries*** or ***APIs***, type casting is essential for assuring data compatibility. Specific data types are expected as ***input arguments*** by many libraries or functions. Programmers may easily link their code with these external components using type casting, facilitating effective data transmission and interoperability.

- **Greater Flexibility:**
  Type casting allows for greater adaptability when managing user input. Programmers can convert a string to the necessary ***data type*** for additional processing or validation, for instance, when accepting input as a string. With more robust input processing and improved error checking made possible by this flexibility, programs are more dependable.

- **Effective Memory Usage:**
  By downsizing variables to smaller data types where it is feasible, ***type casting*** enables programmers to efficiently use memory. For instance, ***type casting*** can assist conserve memory if an ***integer*** value can be securely stored in a smaller data type like char or short. This benefit becomes more significant in systems with limited resources when memory efficiency is crucial.

- **Code Reusability:**
  By enabling programmers to employ functions or components created for a single data type with various ***data types, type casting*** enhances code ***reusability***. They may reuse existing code and prevent duplication by casting variables effectively, cutting down on development time and effort.

- **Error Handling:**
  Error management and ***detection*** can benefit from type casting. An error or exception can be produced. For example, if a conversion fails owing to incompatible data types, giving the chance to resolve the circumstance graciously. During the development and testing phases, this error detection technique aids in locating and fixing any problems.

- **Keeping Data Safe:**
  Casting explicitly in C serves as a reminder of the possibility of data loss during conversions. Programmers must be mindful of the destination type's constrained range or precision when converting a variable to a smaller data type. They can assess the effect on data integrity and, if necessary, create suitable error handling procedures by explicitly casting the variable.

- **Enhancing Performance:**
  By executing some processes more efficiently, type casting can occasionally increase performance. For instance, ***explicit casting*** can aid the compiler in producing more effective machine code, leading to quicker execution, when dealing with complicated expressions containing mixed data types.

Difference Between Type Casting and Type Conversion

implicit casting

Type Casting	Type Conversion
Type casting is a mechanism in which one data type is converted to another data type using a casting () operator by a programmer.	Type conversion allows a compiler to convert one data type to another data type at the compile time of a program or code.
It can be used both compatible data type and incompatible data type.	Type conversion is only used with compatible data types, and hence it does not require any casting operator.
It requires a programmer to manually casting one data into another type.	It does not require any programmer intervention to convert one data type to another because the compiler automatically compiles it at the run time of a program.
It is used while designing a program by the programmer.	It is used or take place at the compile time of a program.
It is more reliable and efficient.	It is less efficient and less reliable.
There is a possibility of data or information being lost in type casting.	In type conversion, data is unlikely to be lost when converting from a small to a large data type.

`float b = 3.0;`
`int a = (int) b`
	`int x = 5, y = 2, c;`
	`float q = 12.5, p;`
	`p = q/x;`

12.5/5 → float

Keywords in C

Keywords in C are ***reserved words*** that have predefined meanings and are part of the C language syntax. These keywords cannot be used as ***variable names, function names,*** or any other identifiers within the program except for their intended purpose. They are used to define the structure flow and behavior of a C program.

The ***compiler*** recognizes a defined set of keywords in the C programming language. These keywords have ***specialized purposes*** and play critical roles in establishing the logic and behavior of a program. Here are some characteristics of C keywords:

- **Reserved:** The C language ***reserves keywords*** are those keywords that cannot be used as identifiers in programs. Using a ***keyword*** as a variable name or other identifier will cause a ***compilation error***.
- **Predefined Meaning:** Each ***keyword*** has a specific meaning that is assigned by the C language. These meanings are built into the ***C language's grammar*** and ***syntax*** and the compiler interprets them accordingly.
- **Specific Use:** ***Keywords*** are designed for specific purposes and contexts within the C language. They define ***control structures, data types, flow control,*** and other language constructs. Attempting to use a keyword outside of its intended purpose will result in a compilation error.
- **Standardized:** C language ***keywords*** are standardized across different compilers and implementations. It ensures the consistency and portability of C programs across different platforms and environments.

A list of 32 keywords in the c language is given below:

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

= → flow control

rename a data type

Storage classes

Identifiers in C

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc.

We can say that an identifier is a collection of alphanumeric characters that begins either with an alphabetical character or an underscore, which are used to represent various programming elements such as variables, functions, arrays, structures, unions, labels, etc. There are 52 alphabetical characters (uppercase and lowercase), underscore character, and ten numerical digits (0-9) that represent the identifiers. There is a total of 63 alphanumerical characters that represent the identifiers.

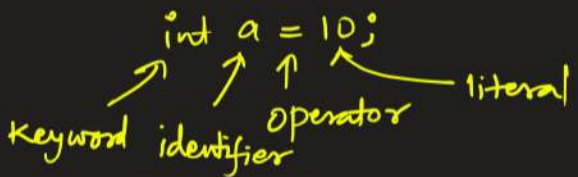
Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Literals in C

Literals are the constant values assigned to the constant variables. We can say that the literals represent the fixed values that cannot be modified. It also contains memory but does not have references as variables. For example, const int =10; is a constant integer expression in which 10 is an integer literal.

Types of literals



There are four types of literals that exist in C programming:

- Integer literal ✓
- Float literal ✓
- Character literal ✓
- String literal ✓

Integer literal

9 A B C D E F

It is a numeric literal that represents only integer type values. It represents the value neither in fractional nor exponential part.

It can be specified in the following three ways:

0–7
8–15
10 11 12

int a = (45);

base 2 ✓
base 8 ✓
base 16 ✓
base 10 ✓

Decimal number (base 10)

It is defined by representing the digits between 0 to 9. For example, 45, 67, etc.

Octal number (base 8)

int a = 012; (10) decimal

It is defined as a number in which 0 is followed by digits such as 0,1,2,3,4,5,6,7. For example, 012, 034, 055, etc.

Hexadecimal number (base 16)

int a = 0x12; (18 decimal) 0XA;

It is defined as a number in which 0x or 0X is followed by the hexadecimal digits (i.e., digits from 0 to 9, alphabetical characters from (a-z) or (A-Z)).

An integer literal is suffixed by following two sign qualifiers:

L or l: It is a size qualifier that specifies the size of the integer type as long.

U or u: It is a sign qualifier that represents the type of the integer as unsigned. An unsigned qualifier contains only positive values.

int 2B

long int 4B

long int a = 1234567890l;

Float literal

It is a literal that contains only floating-point values or real numbers. These real numbers contain the number of parts such as integer part, real part, exponential part, and fractional part. The floating-point literal must be specified either in decimal or in exponential form.

Decimal form

float f = 2.34;

float literal

The decimal form must contain either decimal point, exponential part, or both. If it does not contain either of these, then the compiler will throw an error. The decimal notation can be prefixed either by '+' or '-' symbol that specifies the positive and negative numbers.

Exponential form

float f = -3.14e6;

-3.14×10^6

$= -3140000$

The exponential form is useful when we want to represent the number, which is having a big magnitude. It contains two parts, i.e., mantissa and exponent. For example, the number is 2340000000000, and it can be expressed as 2.34e12 in an exponential form.

Syntax of float literal in exponential form

```
[+/-] <Mantissa> <e/E> [+/-] <Exponent>
```

f = -3.14e-2

-3.14×10^{-2}

```c
#include <stdio.h>
int main() {
    int x1 = 20;
    int x2 = 024;
    int x3 = 0x14;
    unsigned int x4 = 20u;
    long int x5 = 20l;
    float f1 = -0.034;
    float f2 = -3.4e-2;
    printf("%d, %d, %d, %u, %li, %f, %f",
           x1, x2, x3, x4, x5, f1, f2);
    return 0;
}
```

decimal
Octal
hexadecimal
unsigned
long

$= -0.034$

-0.034

-3.4×10^{-2}

$= -0.034$

Character literal

A character literal contains a single character enclosed within single quotes. If multiple characters are assigned to the variable, then we need to create a character array. If we try to store more than one character in a variable, then the warning of a **multi-character character constant** will be generated

String literal

A string literal represents multiple characters enclosed within double-quotes. It contains an additional character, i.e., '\0' (null character), which gets automatically inserted. This null character specifies the termination of the string.

ASCII value in C

Char → 1B = 8 bit

The full form of ASCII is the **American Standard Code for information interchange**. It is a character encoding scheme used for electronics communication. Each character or a special character is represented by some ASCII code, and each ascii code occupies 7 bits in memory.

In C programming language, a character variable does not contain a character value itself rather the ascii value of the character variable. The ascii value represents the character variable in numbers, and each character variable is assigned with some number range from 0 to 127. For example, the ascii value of 'A' is 65.
In the above example, we assign 'A' to the character variable whose ascii value is 65, so 65 will be stored in the character variable rather than 'A'.

char a = 'A';


```c
#include <stdio.h>
int main() {
    int k;   // variable declaration
    for(int k=0; k<=127; k++) { // for loop from 0-127
        printf("The ascii value of %c is %d \n", k,k);
    }
    return 0;
}
```

int x = (int) 'A';


```c
char a = 'A';
printf("%c", a);   A
printf("%d", a);   65
```

ASCII Table

```
Dec  = Decimal Value
Char = Character

'5' has the int value 53 ✓
if we write '5'-'0' it evaluates to 53-48, or the int 5
if we write char c = 'B'+32; then c stores 'b'
```

ASCII Table

Dec = Decimal Value
Char = Character

'5' has the int value 53
if we write '5'-'0' it evaluates to 53-48, or the int 5
if we write char c = 'B'+32; then c stores 'b'

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

printf("%d", 'A'); 65

0: 48
A: 65
a: 97

'd' - 'a'

printf("%d", 'd' - 'a'); 100 97 3