Type Conversion in C

The C compiler attempts data type conversion, especially when dissimilar data types appear in an expression. There are certain times when the compiler does the conversion on its own (implicit type conversion) so that the data types are compatible with each other. On other occasions, the C compiler forcefully performs the conversion (explicit type conversion), which is carried out by the type cast operator.

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.

While performing implicit or automatic type conversions, the C compiler follows the rules of type promotions. Generally, the principle followed is as follows −

- Byte and short values: They are promoted to int.
- If one operand is a long: The entire expression is promoted to long.
- If one operand is a float: The entire expression is promoted to float.
- If any of the operands is double: The result is promoted to double.

Integer Promotion

Integer promotion is the process by which values of integer type "smaller" than int or unsigned int are converted either to int or unsigned int.

Example

Consider an example of adding a character with an integer −

</> Open Compiler

```
#include <stdio.h>

int main(){

   int  i = 17;
   char c = 'c';  /* ascii value is 99 */
```

```
   int sum;
   sum = i + c;

   printf("Value of sum: %d\n", sum);

   return 0;
}
```

Output

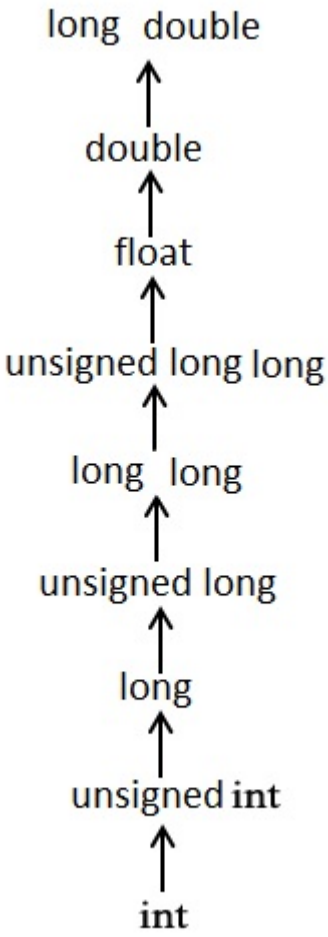
When you run this code, it will produce the following output –

```
Value of sum: 116
```

Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of "c" to ASCII before performing the actual addition operation.

Usual Arithmetic Conversion

Usual arithmetic conversions are implicitly performed to cast their values to a common type. The compiler first performs integer promotion; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy –



Example

Here is another example of implicit type conversion –

</> Open Compiler

```
#include <stdio.h>

int main(){

   char a = 'A';
   float b = a + 5.5;

   printf("%f", b);

   return 0;
}
```

Output

Run the code and check its output –

```
70.500000
```

When the above code runs, the char variable "a" (whose int equivalent value is 70) is promoted to float, as the other operand in the addition expression is a float.

Explicit Type Conversion in C

When you need to covert 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.

C provides a typecast operator. You need to put the data type in parenthesis before the operand to be converted.

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

Note that if type1 is smaller in length than type2, then you don’t need such explicit casting. It is only when type1 is greater in length than type2 that you should use the typecast operator.

Typecasting is required when we want to demote a greater data type variable to a comparatively smaller one or convert it between unrelated types like float to int.

Example

Consider the following code –

</> Open Compiler

```
#include <stdio.h>

int main(){

   int x = 10, y = 4;
   float z = x/y;

   printf("%f", z);
```

```
   return 0;
}
```

Output

On running this code, you will get the following output –

```
2.000000
```

While we expect the result to be 10/4 (that is, 2.5), it shows 2.000000. It is because both the operands in the division expression are of int type. In C, the result of a division operation is always in the data type with larger byte length. Hence, we have to typecast one of the integer operands to float, as shown below –

Example

Take a look at this example –

</> Open Compiler

```
#include <stdio.h>

int main(){

   int x = 10, y = 4;
   float z = (float)x/y;

   printf("%f", z);

   return 0;
}
```

Output

Run the code and check its output –

```
2.500000
```

If we change the expression such that the division itself is cast to float, the result will be different.

Typecasting Functions in C

The standard C library includes a number of functions that perform typecasting. Some of the functions are explained here –

The atoi() Function

The atoi() function converts a string of characters to an integer value. The function is declared in the **stdlib.h** header file.

Example

The following code uses the atoi() function to convert the string "123" to a number 123 –

</> Open Compiler

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

int main(){

   char str[] = "123";
   int num = atoi(str);

   printf("%d\n", num);

   return 0;
}
```

Output

Run the code and check its output –

```
123
```

The itoa() Function

You can use the itoa() function to convert an integer to a null terminated string of characters. The function is declared in the **stdlib.h** header file.

Example

The following code uses itoa() function to convert an integer 123 to string "123" −

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

int main(){

   int num = 123;
   char str[10];

   itoa(num,str, 10);

   printf("%s\n", str);

   return 0;
}
```

Output

Run the code and check its output −

```
123
```

Other examples of using typecasting include the following −

The malloc() Function − The malloc() function is a dynamic memory allocation function.

```
Int *ptr = (int*)malloc(n * sizeof(int));
```

In function arguments and return values − You can apply the typecast operator to formal arguments or to the return value of a user-defined function.

Example

Here is an example −

</> Open Compiler

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

float divide(int, int);

int main(){
   int x = 10, y = 4;
   float z = divide(x, y);

   printf("%f", z);

   return 0;
}

float divide(int a, int b){
   return (float)a/b;
}
```

Output

When you run this code, it will produce the following output –

```
2.500000
```

Employing implicit or explicit type conversion in C helps in type safety and improved code readability, but it may also lead to loss of precision and its complicated syntax may be confusing.