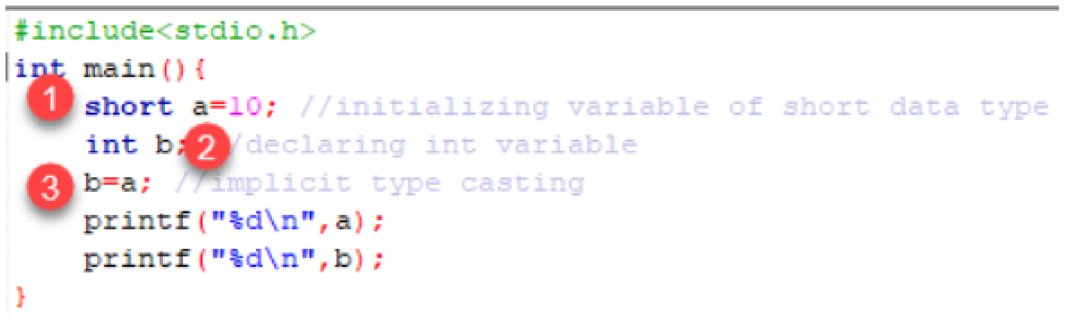
Implicit type casting

- Implicit type casting means conversion of data types without losing its original meaning. This type of typecasting is essential when you want to change data types without changing the significance of the values stored inside the variable.
- Implicit type conversion in C happens automatically when a value is copied to its compatible data type. During conversion, strict rules for type conversion are applied. If the operands are of two different data types, then an operand having lower data type is automatically converted into a higher data type. This type of type conversion can be seen in the following example:



```c
#include<stdio.h>
int main(){
    short a=10; //initializing variable of short data type
    int b; //declaring int variable
    b=a; //implicit type casting
    printf("%d\n",a);
    printf("%d\n",b);
}
```

Output:

```
10
10
```

1. In the given example, we have declared a variable of short data type with value initialized as 10.
2. On the second line, we have declared a variable of an int data type.
3. On the third line, we have assigned the value of variable s to the variable a. On third line implicit type conversion is performed as the value from variable s which is of short data type is copied into the variable a which is of an int data type.

Converting Character to Int:

- Consider the example of adding a character decoded in ASCII with an integer:

```
#include <stdio.h>
main() {
   int  number = 1;
   char character = 'k'; /*ASCII value is 107 */
   int sum;
   sum = number + character;
   printf("Value of sum : %d\n", sum );
}
```

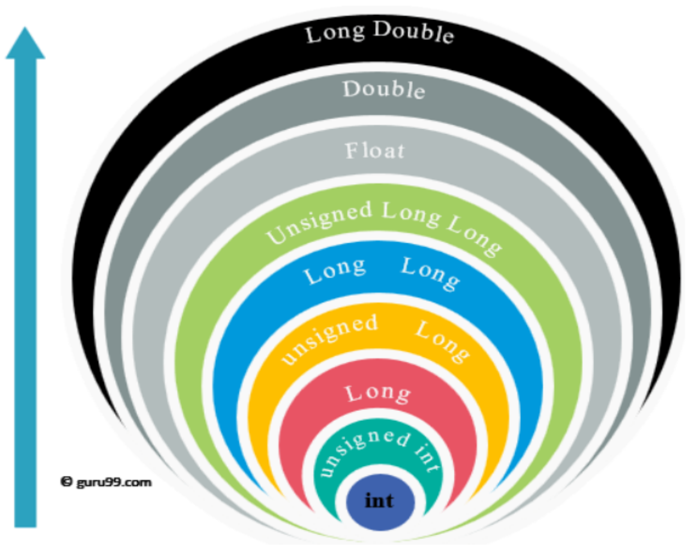
Output:

```
Value of sum : 108
```

- Here, compiler has done an integer promotion by converting the value of ‘k’ to ASCII before performing the actual addition operation.

Arithmetic Conversion Hierarchy

- The compiler first proceeds with promoting a character to an integer. If the operands still have different data types, then they are converted to the highest data type that appears in the following hierarchy chart:



Arithmetic Conversion Hierarchy

Consider the following example to understand the concept:

```
#include <stdio.h>
main() {
   int  num = 13;
   char c = 'k'; /* ASCII value is 107 */
   float sum;
   sum = num + c;
   printf("sum = %f\n", sum );}
```

Output:

```
sum = 120.000000
```

- First, the c variable gets converted to integer, but the compiler converts **num** and **c** into "float" and adds them to produce a 'float' result.

Important Points about Implicit Conversions

- Implicit type of type conversion is also called as standard type conversion. We do not require any keyword or special statements in implicit type casting.
- Converting from smaller data type into larger data type is also called as **type promotion**. In the above example, we can also say that the value of s is promoted to type integer.
- The implicit type conversion always happens with the compatible data types.

- We cannot perform implicit type casting on the data types which are not compatible with each other such as:

  1. Converting float to an int will truncate the fraction part hence losing the meaning of the value.
  2. Converting double to float will round up the digits.
  3. Converting long int to int will cause dropping of excess high order bits.

- In all the above cases, when we convert the data types, the value will lose its meaning. Generally, the loss of meaning of the value is warned by the compiler.

'C' programming provides another way of typecasting which is explicit type casting.

Explicit type casting

- In implicit type conversion, the data type is converted automatically. There are some scenarios in which we may have to force type conversion. Suppose we have a variable div that stores the division of two operands which are declared as an int data type.

```
 int result, var1=10, var2=3;
result=var1/var2;
```

- In this case, after the division performed on variables var1 and var2 the result stored in the variable “result” will be in an integer format. Whenever this happens, the value stored in the variable “result” loses its meaning because it does not consider the fraction part which is normally obtained in the division of two numbers.
-
- To force the type conversion in such situations, we use explicit type casting.it requires a type casting operator. The general syntax for type casting operations is as follows:

```
(type-name) expression
```

Here,

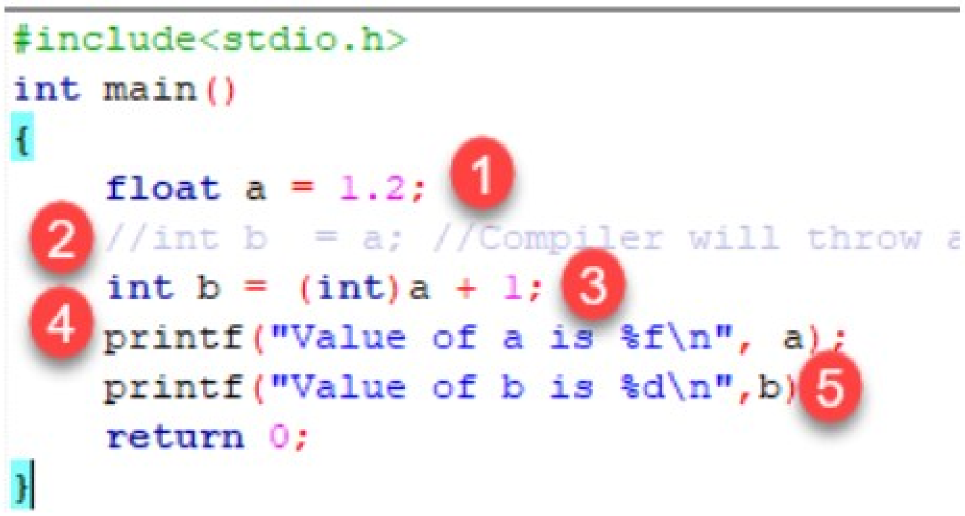
- The type-name is the standard ‘C’ language data type.
- An expression can be a constant, a variable or an actual expression.

Let us write a program to demonstrate how to typecast in C with explicit typecasting.

```
#include<stdio.h>
int main()
{
        float a = 1.2;
        //int b  = a; //Compiler will throw an error for this
        int b = (int)a + 1;
        printf("Value of a is %f\n", a);
        printf("Value of b is %d\n",b);
        return 0;
}
```

Output:

```
Value of a is 1.200000
Value of b is 2
```



```
#include<stdio.h>
int main()
{
    float a = 1.2;
   //int b  = a; //Compiler will throw a
    int b = (int)a + 1;
   printf("Value of a is %f\n", a);
   printf("Value of b is %d\n",b);
    return 0;
}
```

1. We have initialized a variable ‘a’ of type float.
2. Next, we have another variable ‘b’ of integer data type. Since the variable ‘a’ and ‘b’ are of different data types, ‘C’ won’t allow the use of such expression and it will raise an error. In some versions of ‘C,’ the expression will be evaluated but the result will not be desired.
3. To avoid such situations, we have typecast the variable ‘a’ of type float. By using explicit type casting methods, we have successfully converted float into data type integer.
4. We have printed value of ‘a’ which is still a float
5. After typecasting, the result will always be an integer ‘b.’

In this way, we can implement explicit type casting in C programming.

Pointer to an Array:

- A pointer is a very important concept of C language. We can create a pointer to store the address of an array. This created pointer is called a pointer to an array.

- Pointer to an array is also known as an array pointer. We are using the pointer to array to access the elements of the array. It is important to know how to create a pointer to an array when working on a multi-dimension array.

Declaration of a pointer to a 1D Array:

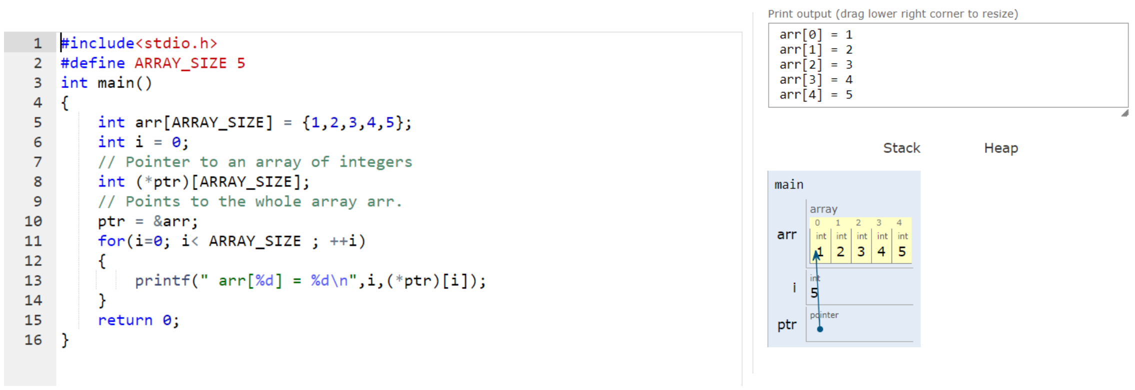
```
data_type (*var_name)[size_of_array];
```

Example,

```
int (*ptr)[5];
```

- Here ptr is a pointer that can point to an array of 5 integers. Since subscript has higher precedence than indirection, it is necessary to enclose the indirection operator and pointer name inside parentheses. Here the type of ptr is ‘pointer to an array of 5 integers’.

- So let see a C program to understand how we can create a pointer to an array and how we can use it in our program.



```c
#include<stdio.h>
#define ARRAY_SIZE 5
int main()
{
    int arr[ARRAY_SIZE] = {1,2,3,4,5};
    int i = 0;
    // Pointer to an array of integers
    int (*ptr)[ARRAY_SIZE];
    // Points to the whole array arr.
    ptr = &arr;
    for(i=0; i< ARRAY_SIZE ; ++i)
    {
        printf(" arr[%d] = %d\n",i,(*ptr)[i]);
    }
    return 0;
}
```

Print output (drag lower right corner to resize)

```
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
```

Array of pointers:

- As we know an array is essentially a collection of elements of the same data types. All elements must be the same and store at the contiguous memory location.

- So, we can create an array of pointers, it is basically an array of the pointer variables. It is also known as pointer arrays.

Declaration of an array of pointers:

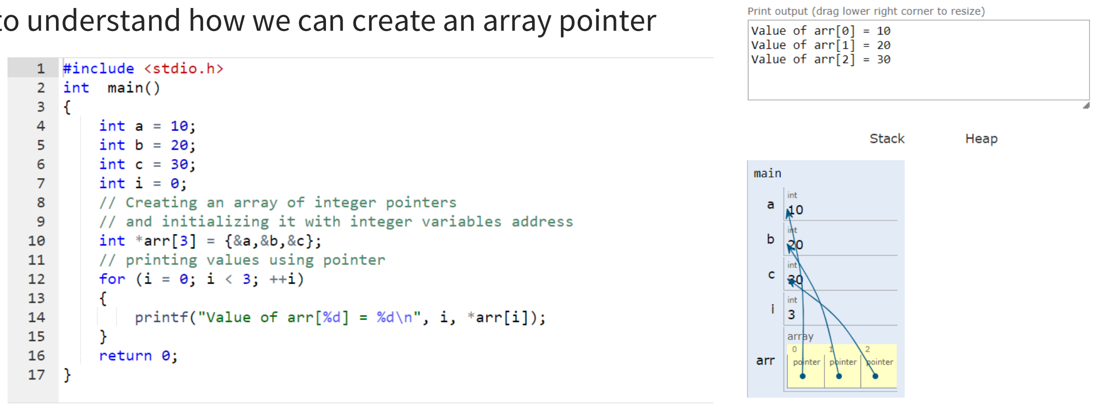
```
data_type *arrar_name[array_size];
```

Example,

```
int *arr[5];
```

Here “arr” is an array of 5 integer pointers.

- So let see a C program to understand how we can create an array pointer and how we can use it in our C program.



```c
#include <stdio.h>
int  main()
{
    int a = 10;
    int b = 20;
    int c = 30;
    int i = 0;
    // Creating an array of integer pointers
    // and initializing it with integer variables address
    int *arr[3] = {&a,&b,&c};
    // printing values using pointer
    for (i = 0; i < 3; ++i)
    {
        printf("Value of arr[%d] = %d\n", i, *arr[i]);
    }
    return 0;
}
```

Print output (drag lower right corner to resize)

```
Value of arr[0] = 10
Value of arr[1] = 20
Value of arr[2] = 30
```

C Function Pointer:

- As we know that we can create a pointer of any data type such as int, char, float, we can also create a pointer pointing to a function. The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

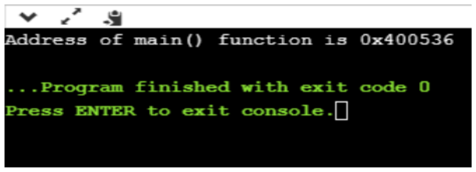
- In the output, we observe that the main() function has some address. Therefore, we conclude that every function has some address.

Let's see a simple example.

```
#include <stdio.h>
int main()
{
   printf("Address of main() function is %p",main);
   return 0;
}
```

The above code prints the address of **main()** function.

Output



```
Address of main() function is 0x400536

...Program finished with exit code 0
Press ENTER to exit console.
```

Declaration of a function pointer:

Syntax of function pointer

```
return type (*ptr_name)(type1, type2…);
```

For example:

```
int (*ip) (int);
```

In the above declaration, *ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

```
float (*fp) (float);
```

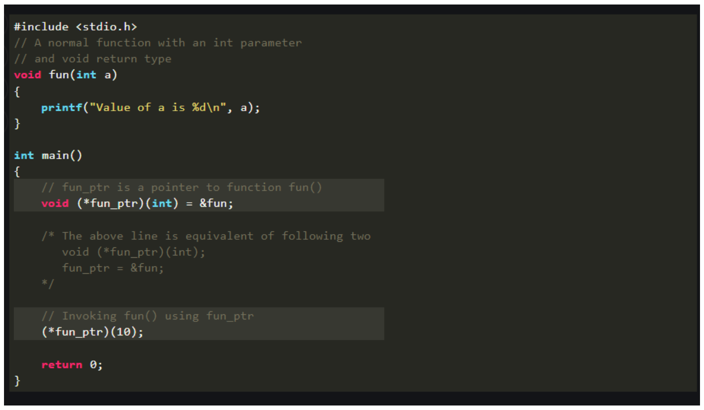
In the above declaration, ***fp** is a pointer that points to a function that returns a float value and accepts a float value as an argument.

We can observe that the declaration of a function is similar to the declaration of a function pointer except that the pointer is preceded by a '*'. So, in the above declaration, fp is declared as a function rather than a pointer.

Till now, we have learnt how to declare the function pointer. Our next step is to assign the address of a function to the function pointer.

```
float (*fp) (int , int);    // Declaration of a function pointer.
float func( int , int );    // Declaration of  function.
fp = func;                  // Assigning address of func to the fp pointer.
```

Following is a simple example that shows declaration and function call using function pointer:



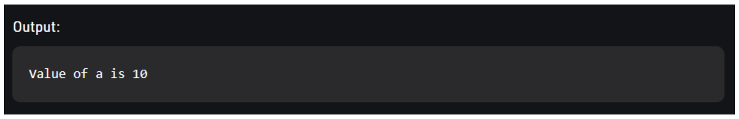
```
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
       void (*fun_ptr)(int);
       fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```



Output:

```
Value of a is 10
```

Following are some interesting facts about function pointers:

1. Unlike normal pointers, a function pointer points to code, not data. Typically, a function pointer stores the start of executable code.
2. Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
3. A function's name can also be used to get functions' address.
4. Like normal pointers, we can have an array of function pointers.
5. Function pointer can be used in place of switch case.
6. Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.