**Array of pointers**

Since we have four integer pointers so, we can either create four separate integer pointer variables like ptr1, ptr2, ptr3 and ptr4.

Or, we can create one single integer array of pointers ptr variable that will point at the four variables.

In the following example we are creating an array of integer pointers ptr of size 4.

// array of integer pointers

int \*ptr[4];

**Assign address to array of pointers**

This step is similar to any other pointer variable. We get the address of a variable using the **address of** & operator and then save that address in the array of pointers.

In the following example we are saving the address of the integer variables num, score, run and goal in the integer array of pointers variable ptr.

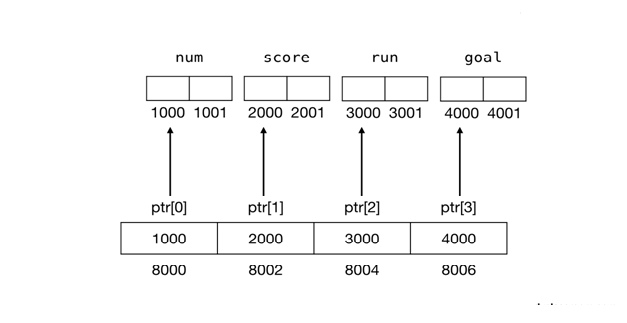
// assign address of variables to ptr

ptr[0] = &num;

ptr[1] = &score;

ptr[2] = &run;

ptr[3] = &goal;



Assuming that an integer address value takes 2 bytes so, each pointer element size is 2 bytes. So, the array of integer pointer ptr takes memory space from 8000 to 8007 i.e., total 8 bytes (2 bytes for each element).

The first element of the array of integer pointer ptr holds the address of the num variable.

Similarly, the second element of the array of integer pointer ptr holds the address of the score variable.

The third element of ptr holds the address of the run variable and the fourth element of ptr holds the address of the goal variable.

## Access value of the variables via array of pointers

To access the value of the variables via array of pointers we have to use the **value at the address of** \* operator.

// print value

printf("num: %d\n", \*ptr[0]);

printf("score: %d\n", \*ptr[1]);

printf("run: %d\n", \*ptr[2]);

printf("goal: %d\n", \*ptr[3]);

### Complete code

#include <stdio.h>

int main(void) {

// integer variables

int num = 10;

int score = 12;

int run = 123;

int goal = 3;

// array of integer pointers

int \*ptr[4];

// assign address of variables to ptr

ptr[0] = #

ptr[1] = &score;

ptr[2] = &run;

ptr[3] = &goal;

// print value

printf("num: %d\n", \*ptr[0]);

printf("score: %d\n", \*ptr[1]);

printf("run: %d\n", \*ptr[2]);

printf("goal: %d\n", \*ptr[3]);

return 0;

}

**Output:**

num: 10

score: 12

run: 123

goal: 3

Below is an array of pointers in [C](https://www.computerhope.com/jargon/c/c.htm) that sets each pointer in one array to point to an [integer](https://www.computerhope.com/jargon/i/integer.htm) in another and then print the values of the integers by dereferencing the pointers. In other words, this code prints the value in memory of where the pointers point.

#include <stdio.h>

const int ARRAY\_SIZE = 5;

int main ()

{

/\* first, declare and set an array of five integers: \*/

int array\_of\_integers[] = {5, 10, 20, 40, 80};

/\* next, declare an array of five pointers-to-integers: \*/

int i, \*array\_of\_pointers[ARRAY\_SIZE];

for ( i = 0; i < ARRAY\_SIZE; i++)

{

/\* for indices 1 through 5, set a pointer to

point to a corresponding integer: \*/

array\_of\_pointers[i] = &array\_of\_integers[i];

}

for ( i = 0; i < ARRAY\_SIZE; i++)

{

/\* print the values of the integers pointed to

by the pointers: \*/

printf("array\_of\_integers[%d] = %d\n", i, \*array\_of\_pointers[i] );

}

return 0;

}

**The output of the above program is:**

array\_of\_integers[0] = 5

array\_of\_integers[1] = 10

array\_of\_integers[2] = 20

array\_of\_integers[3] = 40

array\_of\_integers[4] = 80

## Void Pointer Basics:

1. In C **General Purpose Pointer** is called as void Pointer.
2. It does not have any data type associated with it
3. It can store address of any type of variable
4. A void pointer is a C convention for a raw address.
5. The compiler has no idea what type of object a void Pointer really points to ?

## Declaration of Void Pointer:

void \* pointer\_name;

### Void Pointer Example:

**void** \*ptr; // ptr is declared as Void pointer

**char** cnum;

**int** inum;

**float** fnum;

ptr = &cnum; // ptr has address of character data

ptr = &inum; // ptr has address of integer data

ptr = &fnum; // ptr has address of float data

### Explanation:

**void** \*ptr;

1. **Void pointer** declaration is shown above.
2. We have declared 3 variables of integer,character and float type.
3. When we assign **address of integer** to the void pointer, pointer will become Integer Pointer.
4. When we assign **address of Character** Data type to void pointer it will become Character Pointer.
5. Similarly we can assign address of any data type to the void pointer.
6. It is capable of storing address of any data type

## Function Pointer Syntax

The syntax for declaring a function pointer might seem messy at first, but in most cases it's really quite straight-forward once you **understand** what's going on. Let's look at a simple example:

void (\*foo)(int);

In this example, foo is a pointer to a function taking one argument, an integer, and that returns void. It's as if you're declaring a function called "\*foo", which takes an int and returns void; now, if \*foo is a function, then foo must be a pointer to a function. (Similarly, a declaration like int \*x can be read as \*x is an int, so x must be a pointer to an int.)   
  
The key to writing the declaration for a function pointer is that you're just writing out the declaration of a function but with (\*func\_name) where you'd normally just put func\_name. 

**Reading Function Pointer Declarations**

Sometimes people get confused when more stars are thrown in:

void \*(\*foo)(int \*);

Here, the key is to read inside-out; notice that the innermost element of the expression is \*foo, and that otherwise it looks like a normal function declaration. \*foo should refer to a function that returns a void \* and takes an int \*. Consequently, foo is a pointer to just such a function.

### Initializing Function Pointers

To initialize a function pointer, you must give it the address of a function in your program. The syntax is like any other variable:

#include <stdio.h>

void my\_int\_func(int x)

{

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

}

int main()

{

void (\*foo)(int);

/\* the ampersand is actually optional \*/

foo = &my\_int\_func;

return 0;

}

(Note: all examples are written to be compatible with both C and C++.)

### Using a Function Pointer

To call the function pointed to by a function pointer, you treat the function pointer as though it were the name of the function you wish to call. The act of calling it performs the dereference; there's no need to do it yourself:

#include <stdio.h>

void my\_int\_func(int x)

{

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

}

int main()

{

void (\*foo)(int);

foo = &my\_int\_func;

/\* call my\_int\_func (note that you do not need to write (\*foo)(2) ) \*/

foo( 2 );

/\* but if you want to, you may \*/

(\*foo)( 2 );

return 0;

}

Note that function pointer syntax is flexible; it can either look like most other uses of pointers, with & and \*, or you may omit that part of syntax. This is similar to how arrays are treated, where a bare array decays to a pointer, but you may also prefix the array with & to request its address.

**Example:**

int sum (int num1, int num2)

{

return num1+num2;

}

int main()

{

/\* The following two lines can also be written in a single

\* statement like this: void (\*fun\_ptr)(int) = &fun;

\*/

int (\*f2p) (int, int);

f2p = sum;

//Calling function using function pointer

int op1 = f2p(10, 13);

//Calling function in normal way using function name

int op2 = sum(10, 13);

printf("Output1: Call using function pointer: %d",op1);

printf("\nOutput2: Call using function name: %d", op2);

return 0;

}

**Output:**

Output1: Call using function pointer: 23

Output2: Call using function name: 23

**Some points regarding function pointer:**

1. As mentioned in the comments, you can declare a function pointer and assign a function to it in a single statement like this:

void (\*fun\_ptr)(int) = &fun;

2. You can even remove the ampersand from this statement because a function name alone represents the function address. This means the above statement can also be written like this:

void (\*fun\_ptr)(int) = fun;