CS101: Problem Solving through C Programming

Sachchida Nand Chaurasia
Assistant Professor

Department of Computer Science
Banaras Hindu University
Varanasi
Email id: snchaurasia@bhu.ac.in, sachchidanand.mca07@gmail.com

February 22, 2021

Pointer I

What is a Pointer?

✓ A pointer is a variable used to store a memory address. Let's first learn how memory is organized inside a computer.

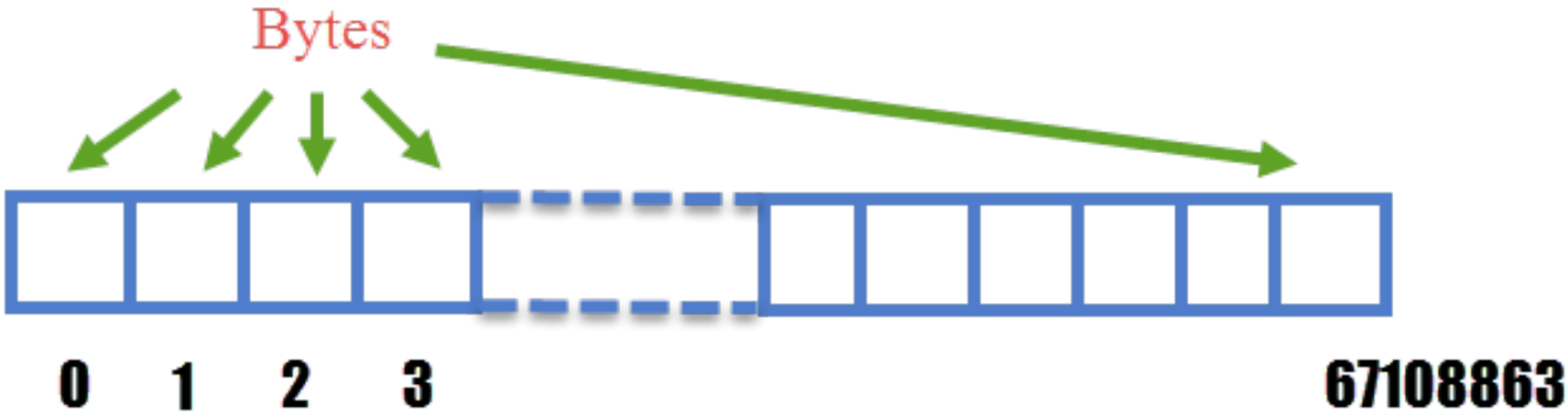
✓ Memory in a computer is made up of bytes (A byte consists of 8 bits) arranged in a sequential manner.

✓ Each byte has a number associated with it just like index or subscript in an array, which is called the address of the byte.

✓ The address of byte starts from 0 to one less than the size of memory.

✓ For example, say in a 64MB of RAM, there are 64 * 2^20 = 67108864 bytes. Therefore the address of these bytes will start from 0 to 67108863.

Pointer II



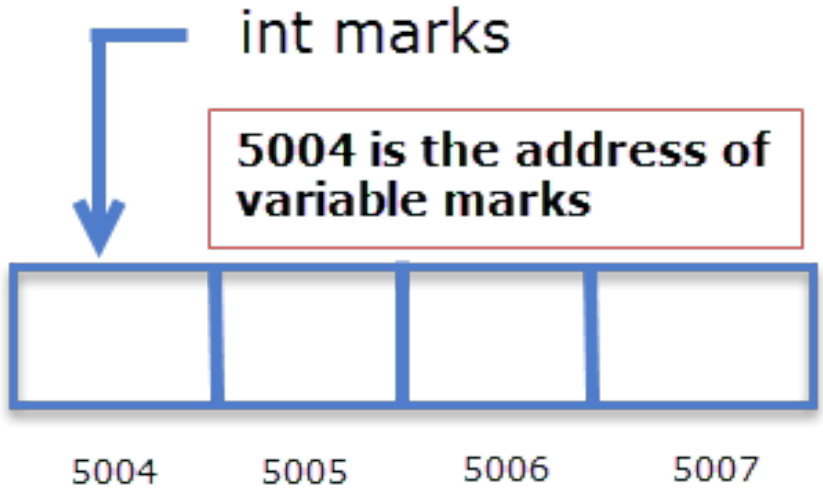
Let's see what happens when you declare a variable:

✓ As we know an int occupies 4 bytes of data (assuming we are using a 32-bit compiler), so compiler reserves 4 consecutive bytes from memory to store an integer value.

✓ The address of the first byte of the 4 allocated bytes is known as the address of the variable marks.

✓ Let's say that address of 4 consecutive bytes are 5004, 5005, 5006 and 5007 then the address of the variable marks will be 5004.

Pointer III



Address Operator (&):

✓ To find the address of a variable, C provides an operator called address operator (&).

✓ To find out the address of the variable marks we need to place & operator in front of it, like this:

```
1 &marks
```

Pointer IV

```c
// Program to demonstrate address(&) operator
#include<stdio.h>
int main()
{
     int i = 12;
     printf("Address of i = %u \n", &i);
     printf("Value of i = %d ", i);
     // signal to operating system program ran fine
     return 0;
}
```

How it works:

✓ To find the address of the variable, precede the variable name by & operator.

✓ Another important thing to notice about the program is the use of %u conversion specification.

✓ %u conversion specification is used to print unsigned decimal numbers and since the memory addresses can't be negative, you must always use %u instead of %d.

✓ Address of operator (&) can't be used with constants or expression, it can only be used with a variable.

Pointer V

```
1 &var; // ok
2
3 &12; // error because we are using & operator with a constant
4
5 &(x+y) // error because we are using & operator with an expression</pre>
```

We have been using address operator(&) in the function scanf() without knowing why? The address of a variable is provided to scanf(), so that it knows where to write data.

Declaring Pointer Variables:

✓ A pointer is a variable that stores a memory address.

✓ Just like any other variables you need to first declare a pointer variable before you can use it. Here is how you can declare a pointer variable.

Syntax: data_type *pointer_name;

data_type is the type of the pointer (also known as the base type of the pointer).

pointer_name is the name of the variable, which can be any valid C identifier.

Let's take some examples:

Pointer VI

```
1 int *ip;
2 float *fp;
```

✓ int *ip means that ip is a pointer variable capable of pointing to variables of type int. In other words, a pointer variable ip can store the address of variables of type int only.

✓ Similarly, the pointer variable fp can only store the address of a variable of type float.

✓ The type of variable (also known as base type) ip is a pointer to int and type of fp is a pointer to float.

✓ A pointer variable of type pointer to int can be symbolically represented as (int *).

✓ Similarly, a pointer variable of type pointer to float can be represented as (float *).

✓ Just like other variables, a pointer is a variable so, the compiler will reserve some space in memory.

✓ All pointer variable irrespective of their base type will occupy the same space in memory.

✓ Normally 4 bytes or 2 bytes (On a 16-bit Compiler) are used to store a pointer variable (this may vary from system to system).

Assigning Address to Pointer Variable:

Pointer VII

✓ After declaring a pointer variable the next step is to assign some valid memory address to it.

✓ You should never use a pointer variable without assigning some valid memory address to it, because just after declaration it contains garbage value and it may be pointing to anywhere in the memory.

✓ The use of an unassigned pointer may give an unpredictable result. It may even cause the program to crash.

```
1 int *ip, i = 10;
2 float *fp, f = 12.2;
3
4 ip = &i;
5 fp = &f;
```

Pointer VIII

✓ Here ip is declared as a pointer to int, so it can only point to the memory address of an int variable.

✓ Similarly, fp can only point to the address of a float variable.

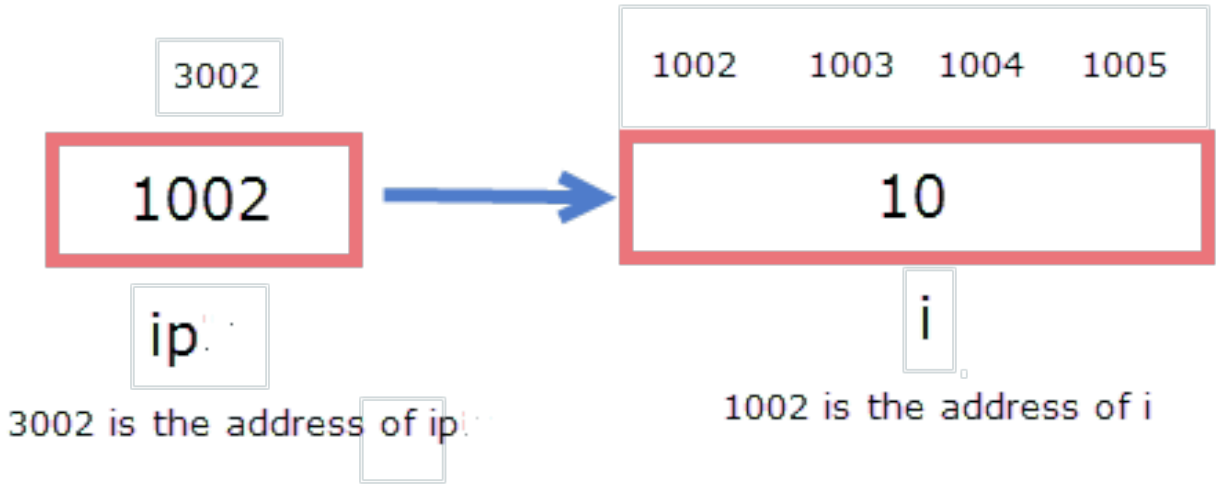
✓ In the last two statements, we have assigned the address of i and f to ip and fp respectively.

✓ Now, ip points to the variable i and fp points to variable f.

✓ It is important to note that even if you assign an address of a float variable to an int pointer, the compiler will not show you any error but you may not get the desired result.

✓ So as a general rule always you should always assign the address of a variable to its corresponding pointer variable of the same type.

Pointer IX



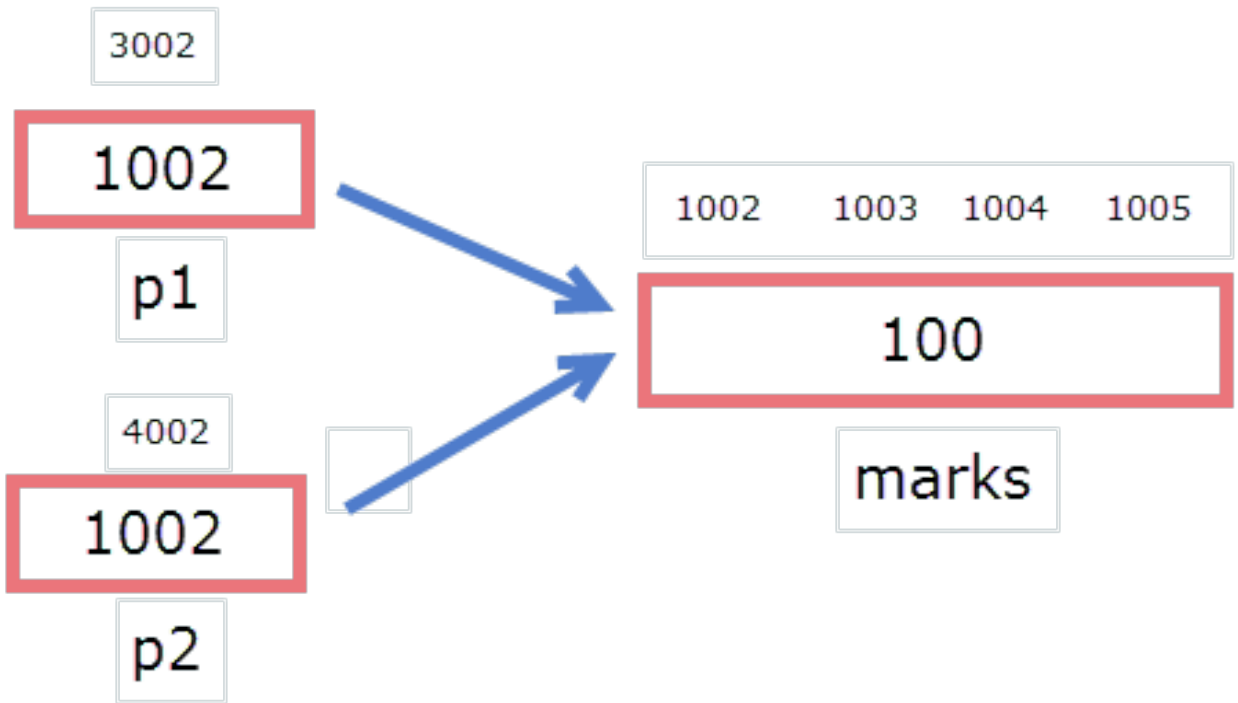
ip pointing to address of i

✓ We can initialize the pointer variable at the time of declaration, but in this case, the variable must be declared and initialized before the pointer variable.

```
1 int i = 10, *iptr = &i;
```

Pointer X

```
1 int marks = 100, *p1, *p2;
2
3 p1 = &marks;
4
5 p2 = p1;
6
7 //After the assignment, p1 and p2 points to the same variable marks.
```



Pointer XI

✓ As already said when a pointer variable is declared it contains garbage value and it may be point anywhere in the memory.

✓ You can assign a symbolic constant called NULL (defined in stdio.h) to any pointer variable.

✓ The assignment of NULL guarantees that the pointer doesn't point to any valid memory location.

```
1 int i = 100, *iptr;
2
3 iptr = NULL;
```

Dereferencing Pointer Variable:

✓ Dereferencing a pointer variable simply means accessing data at the address stored in the pointer variable.

✓ Up until now, we have been using the name of the variable to access data inside it, but we can also access variable data indirectly using pointers.

✓ To make it happen, we will use a new operator called the indirection operator (*).

✓ By placing indirection operator (*) before a pointer variable we can access data of the variable whose address is stored in the pointer variable.

Pointer XII

```
1 int i = 100, *ip = &i;
```

✓ Here ip stores address of variable i, if we place * before ip then we can access data stored in the variable i. It means the following two statements does the same thing.

```
1 printf("%d\n", *ip); // prints 100
2 printf("%d\n", i); // prints 100
```

✓ Indirection operator (*) can be read as value at the address. For example, *ip can be read as value at address ip.

Note: It is advised that you must never apply indirection operator to an uninitialized pointer variable, doing so may cause unexpected behaviour or the program may even crash.

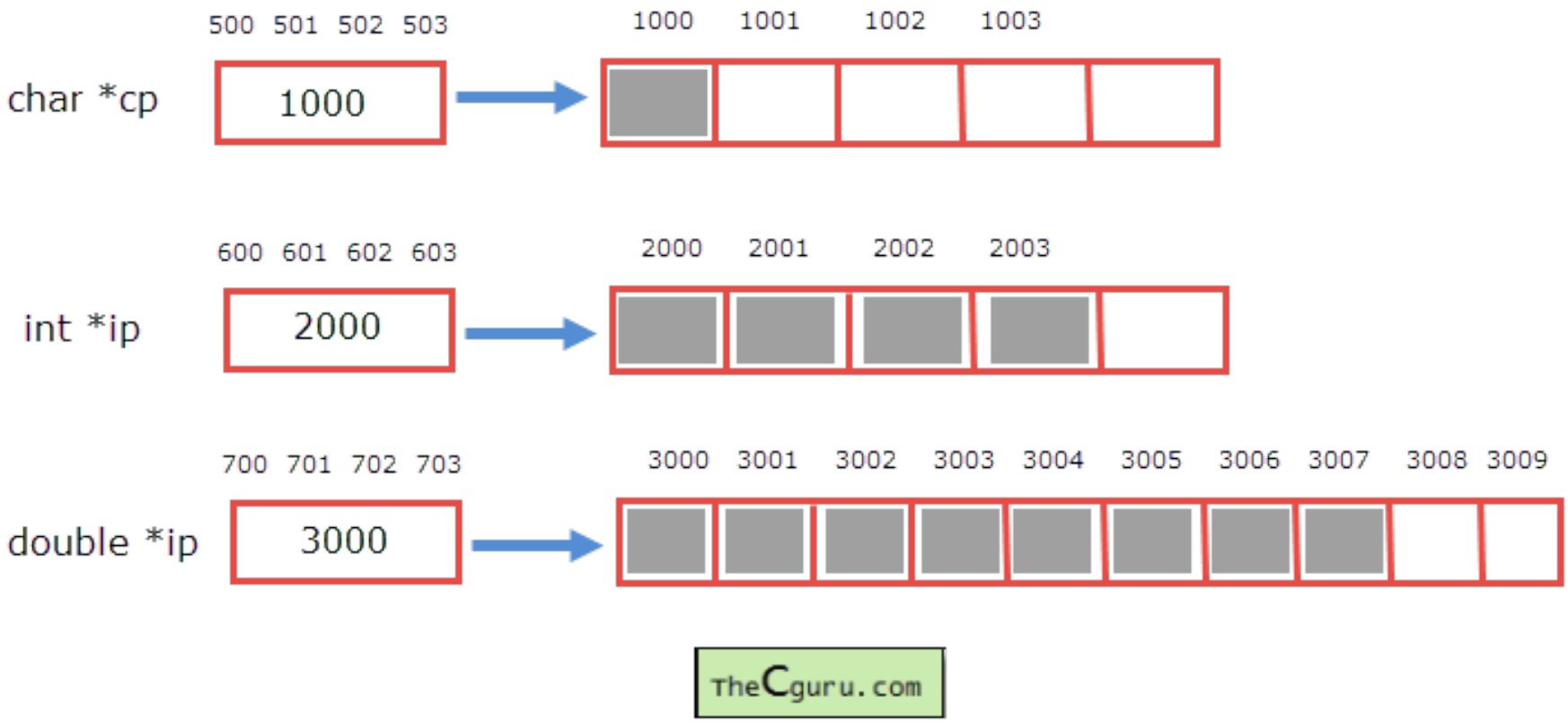
```
1 int *ip;
2 printf("%d", *ip); // WRONG
```

Pointer XIII

✓ Now we know by dereferencing a pointer variable, we can access the value at the address stored in the pointer variable. Let's dig a little deeper to view how the compiler actually retrieves data.

```
1 char ch = 'a';
2 int i = 10;
3 double d = 100.21;
4
5 char *cp = &ch;
6 int *ip = &i;
7 double *ip = &d;
```

Pointer XIV



Interpret the meaning of the following expression:

```
1 *(&i) , where i is a variable of type int
```

Pointer XV

✓ We know from the precedence table that parentheses () has the highest precedence, so &i is evaluated first.

✓ Since &i is the address of variable i, so dereferencing it with * operator will give us the value of the variable i.

✓ So we can conclude that writing *(&i) is same as writing i.

✓ The following example demonstrates everything we have learned about pointers so far.

```
#include<stdio.h>
int main()
{
     int i = 12, *ip = &i;
     double d = 2.31, *dp = &d;
     printf("Value of ip = address of i = %d\n", ip);
     printf("Value of fp = address of d = %d\n\n", d);
     printf("Address of ip = %d\n", &ip);
     printf("Address of dp = %d\n\n", &dp);
     printf("Value at address stored in ip = value of i = %d\n", *ip);
     printf("Value at address stored in dp = value of d = %f\n\n", *dp);
     // memory occupied by pointer variables
```

Pointer XVI

```
13      // is same regardless of its base type
14      printf("Size of pointer ip = %d\n", sizeof(ip));
15      printf("Size of pointer dp = %d\n\n", sizeof(dp));
16      // signal to operating system program ran fine
17      return 0;
18  }
```

Note: Memory address may vary every time you run the program.

Pointer Arithmetic in C I

✓ We can't perform every type of arithmetic operations with pointers.

✓ Pointer arithmetic is slightly different from arithmetic we normally use in our daily life.

✓ The only valid arithmetic operations applicable on pointers are:

1. Addition of integer to a pointer
2. Subtraction of integer to a pointer
3. Subtracting two pointers of the same type

✓ The pointer arithmetic is performed relative to the base type of the pointer.

✓ For example, if we have an integer pointer ip which contains address 1000, then on incrementing it by 1, we will get 1004 (i.e 1000 + 1 * 4) instead of 1001 because the size of the int data type is 4 bytes.

✓ If we had been using a system where the size of int is 2 bytes then we would get 1002 (i.e 1000 + 1 * 2).

✓ Similarly, on decrementing it we will get 996 (i.e 1000 - 1 * 4) instead of 999.

✓ So, the expression ip + 4 will point to the address 1016 (i.e 1000 + 4 * 4).

Pointer Arithmetic in C II

```
1 int i = 12, *ip = &i;
2 double d = 2.3, *dp = &d;
3 char ch = 'a', *cp = &ch;
```

✓ Suppose the address of i, d and ch are 1000, 2000, 3000 respectively, therefore ip, dp and cp are at 1000, 2000, 3000 initially.

Pointer Arithmetic on Integers:

Pointer Expression	How it is evaluated ?
ip = ip + 1	ip =>ip + 1 =>1000 + 1*4 =>1004
ip++ or ++ip	ip++ =>ip + 1 =>1004 + 1*4 =>1008
ip = ip + 5	ip =>ip + 5 =>1008 + 5*4 =>1028
ip = ip - 2	ip =>ip - 2 =>1028 - 2*4 =>1020
ip– or –ip	ip =>ip + 2 =>1020 + 2*4 =>1028

Pointer Arithmetic on Float:

Pointer Arithmetic in C III

Pointer Expression	How it is evaluated ?
dp + 1	dp = dp + 1 =>2000 + 1*8 =>2008
dp++ or ++dp	dp++ =>dp+1 =>2008+1*8 =>2016
dp = dp + 5	dp =>dp + 5 =>2016+5*8 =>2056
dp = dp - 2	dp =>dp - 2 =>2056-2*8 =>2040
dp– or –dp	dp =>dp - 1=>2040-1*8=>2032‘

Pointer Arithmetic on Characters:

Pointer Expression	How it is evaluated ?
cp + 1	cp = cp + 1 =>3000 + 1*1 =>3001
cp++ or ++cp	cp =>cp + 1 =>3001 + 1*1 =>3002
cp = cp + 5	cp =>cp + 5 =>3002 + 5*1 =>3007
cp = cp - 2	cp =>cp + 5 =>3007 - 2*1 =>3005
cp– or –cp	cp =>cp + 2 =>3005 - 1*1 =>3004

Note: When we increment or decrement pointer variables using pointer arithmetic then, the address of variables i , d , ch are not affected in any way.

Pointer Arithmetic in C IV

✓ Arithmetic operation on type char seems like ordinary arithmetic because the size of char type is 1 byte.

✓ Another important point to note is that when we increment and decrement pointer variable by adding or subtracting numbers then it is not necessary that the pointer variable still points to a valid memory location.

✓ So, we must always pay special attention when we move the pointer in this way.

✓ Generally, we use pointer arithmetic with arrays because elements of an array are arranged in contiguous memory locations.

program shows pointer arithmetic:

```
#include<stdio.h>
int main()
{
     int i = 12, *ip = &i;
     double d = 2.3, *dp = &d;
     char ch = 'a', *cp = &ch;

     printf("Value of ip = %u\n", ip);
     printf("Value of dp = %u\n", dp);
     printf("Value of cp = %u\n\n", cp);
```

```
11
12      printf("Value of ip + 1 = %u\n", ip + 1);
13      printf("Value of dp + 1 = %u\n", dp + 1);
14      printf("Value of cp + 1 = %u\n\n", cp + 1);
15
16      printf("Value of ip + 2 = %u\n", ip + 2);
17      printf("Value of dp + 2 = %u\n", dp + 2);
18      printf("Value of cp + 2 = %u\n", cp + 2);
19
20      return 0;
21  }
```

Pointer Arithmetic between Two Pointers:

✓ If we have two pointers p1 and p2 of base type pointer to int with addresses 1000 and 1016 respectively, then p2 - p1 will give 4, since the size of int type is 4 bytes.

✓ If you subtract p2 from p1 i.e p1 - p2 then the answer will be negative i.e -4.

The following program demonstrates pointer arithmetic between two pointers of the same type.

Pointer Arithmetic in C VI

```c
#include<stdio.h>

int main()
{
    int i1 = 12, *ip1 = &i1;
    int i2 = 12, *ip2 = &i2;

    printf("Value of ip1 or address of i1 = %u\n", ip1);
    printf("Value of ip2 or address of i2 = %u\n\n", ip2);

    printf("ip2 - ip1 = %d\n", ip1 - ip2);
    printf("ip1 - ip2 = %d\n", ip2 - ip1);

    return 0;
}
```

Combining Indirection operator (*) and Increment/Decrement operator:

✓ While processing elements of an array C programmers often mix indirection operator (*) and increment/decrement operator(++ and –).

Always remember that the precedence of indirection operator (*) and increment/decrement operator are same and they associate from right to left

✓ Suppose x is integer variable and p is a pointer to int. Now consider the following statements and try to interpret them.

Example 1:

```
1 x = *p++;
```

✓ Since * and ++ operators have the same precedence and associate from right to left ++ will be applied to p, not to *p. Because the increment operator is postfix, so first the value of p is used in the expression then it will be incremented. Therefore, the first integer pointed by p will be dereferenced and assigned to x, then the value of p will be incremented by 1.

Example 2:

```
1 x = ++*p;
```

✓ Here * operator will be first applied to p then ++ will be applied to *p. Therefore first integer pointer is dereferenced, the value obtained from dereferencing is incremented and eventually assigned to x.

Example 3:

```
1 x = *++p;
```

✓ The ++ operator is prefixed, so first, p will be incremented, then the value at the new address is dereferenced and assigned to x.

Note: If you still have any confusion, you can always use () around expression which you want to evaluate first.

Pointer Comparison

✓ You can use relational operators (<, <=, >, >= , == , !=) with pointers.

✓ The == and !=operators are used to compare two pointers whether they contain the same address or not.

✓ Two pointers are equal when they both are null, or contains the address of the same variable.

✓ Use of these (i.e == and !=) operators are valid only when pointers are of the same base type, or between a null pointer and any other pointer, or void pointer(will be discussed later) and any other pointer.

✓ Use of other relational operators (<, <=, >, >=) to compare two pointers is meaningful only when they both point to the elements of the same array.

Pointer to Pointer

✓ As we know the pointer is a variable that contains the memory address. The pointer variable itself occupies some space in memory and hence it also has a memory address.

✓ We can store the address of the pointer variable in some other variable, which is known as pointer to pointer.

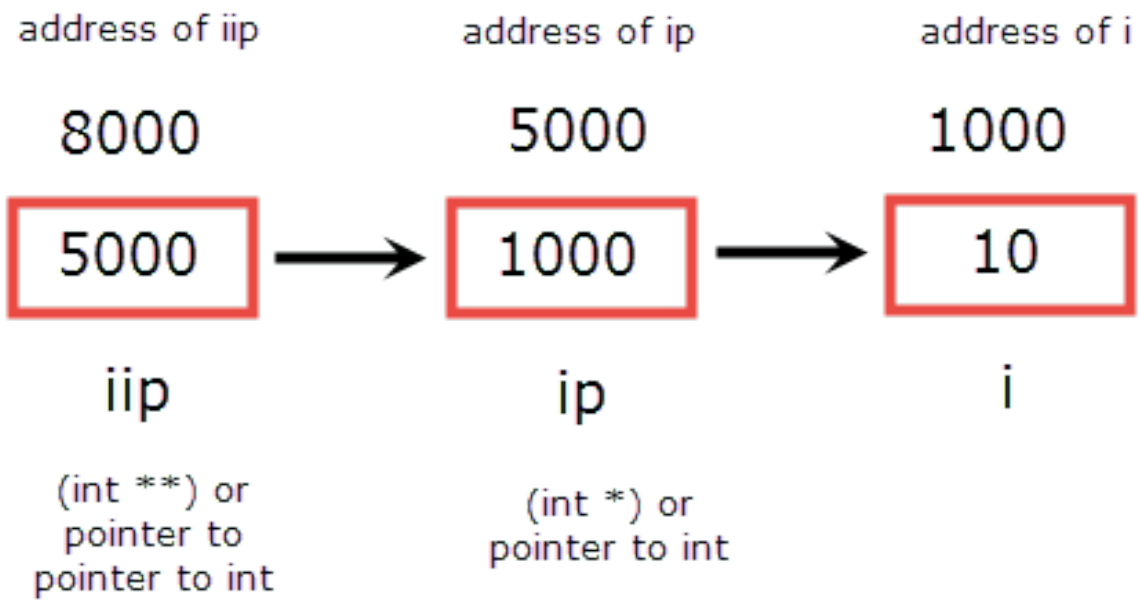
✓ The syntax of declaring a pointer to a pointer is as follows:

Syntax: data_type **p;

```
1 int i = 10;
2 int *ip = &i;
3 int **iip = &ip;
```

Pointer Arithmetic in C X

✓ Here ip is of type (int *) or pointer to int, iip is of type (int **) or pointer to pointer to int.



✓ We know that *ip will give value at address ip i.e value of i. Can you guess what value will **iip will return?

```
1 **iip
```

Pointer Arithmetic in C XI

✓ We know that the indirection operator evaluated from right to left so **iip can also be written as

```
1 *(*iip)
```

✓ *iip means value at address iip or address stored at ip. On dereferencing the address stored at ip we will get the value stored in the variable i.

```
1 *(*iip)
2 => *ip
3 => i
```

✓ Therefore **iip gives the value stored in the variable i.

Pointer Arithmetic in C XII

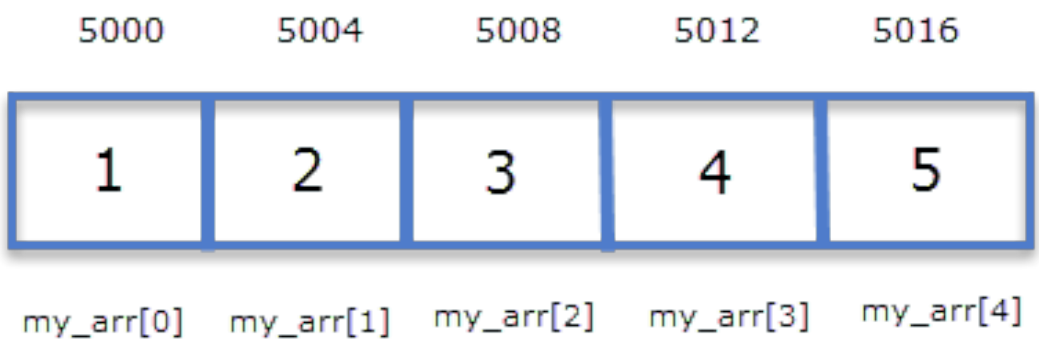
```c
#include<stdio.h>

int main()
{
    int i = 10;
    int *ip = &i;
    int **iip = &ip;

    printf("Value of i = %d\n\n", i);

    printf("Address of i = %u\n", &i);
    printf("Value of ip = %d\n\n", ip);

    printf("Address of ip = %u\n", &ip);
    printf("Value of iip = %d\n\n", iip);

    printf("Value of *iip = value of ip = %d\n", *iip);
    printf("Value of **iip = value of i = %d\n\n", **iip);

    return 0;
}
```

Pointer Arithmetic in C XIII

Pointers and 1-D arrays I

✓ In C, the elements of an array are stored in contiguous memory locations. For example: if we have the following array.

```
int my_arr[5] =
{
     1, 2, 3, 4, 5
}
;
```



Pointers and 1-D arrays II

✓ Here &my_arr[0] points to the address of the first element of the array.
✓ Since the name of the array is a constant pointer that points to the first element of the array, my_arr and &my_arr[0] represent the same address.
✓ &my_arr[1] points to the address of the second element. Similarly &my_arr[2] points to the address of the third element and so on.

Note: my_arr is of type (int *) or pointer to int.

```
#include<stdio.h>

int main()
{
     int my_arr[5] =
     {
          1, 2, 3, 4, 5
     }
     , i;

     for(i = 0; i < 5; i++)
     {
```

```
13              printf("Value of a[%d] = %d\t", i, my_arr[i]);
14              printf("Address of a[%d] = %u\n", i, &my_arr[i]);
15      }
16
17      // signal to operating system program ran fine
18      return 0;
19  }
```

Using pointers to access elements and address of elements in an array:

```
1  int my_arr[5] =
2  {
3      1, 2, 3, 4, 5
4  }
5  ;
```

Pointers and 1-D arrays IV

✓ Here arr is a pointer to the first element. But, What is the base type of pointer arr ?

✓ The answer is a pointer to int or (int *).

✓ In this case, arr points to the address of an integer number i.e address of integer 1.

So the base type of arr is a pointer to int or (int*).

Let's take some more examples:

```
1 char arr[] =
2 {
3     'A','B','C','D','E'
4 }
5 ;
```

What is the type of pointer arr ?.

✓ Here arr points to the address of the first element which is a character. So the type ofTi arr is a pointer to char or (char *).

Similarly,

Pointers and 1-D arrays V

```
double arr[] =
{
     1.03, 29.3, 3.42, 49.3, 51.2
}
;
```

Here arr is a pointer of type pointer to double or (double *).

```
int my_arr[5] =
{
     11, 22, 33, 44, 55
}
;
```

✓ Here my_arr is a constant pointer of base type pointer to int or (int *) and according to pointer arithmetic when an integer is added to a pointer we get the address of the next element of same base type.

Pointers and 1-D arrays VI

```
my_arr is same as &my_arr[0]
my_arr + 1 is same as &my_arr[1]
my_arr + 2 is same as &my_arr[2]
my_arr + 3 is same as &my_arr[3]
my_arr + 4 is same as &my_arr[4]
```

```
*(my_arr) is same as my_arr[0]
*(my_arr + 1) is same as my_arr[1]
*(my_arr + 2) is same as my_arr[2]
*(my_arr + 3) is same as my_arr[3]
*(my_arr + 4) is same as my_arr[4]
```

The following program prints value and address of array elements using pointer notation.

Pointers and 1-D arrays VII

```c
#include<stdio.h>
int main()
{
     int my_arr[5] =
     {
          1, 2, 3, 4, 5
     }
     , i;

     for(i = 0; i < 5; i++)
     {
            printf("Value of a[%d] = %d\t", i, *(my_arr + i) );
            printf("Address of a[%d] = %u\n", i, my_arr + i );
     }
     // signal to operating system program ran fine
     return 0;
}
```

Assigning 1-D array to a Pointer variable

You can assign a 1-D array to a pointer variable. Consider the following example:

```
1 int *p;
2 int my_arr[] =
3 {
4      11, 22, 33, 44, 55
5 }
6 ;
7 p = my_arr;
```

✓ Now you can use pointer p to access address and value of each element in the array.

✓ It is important to note that assignment of a 1-D array to a pointer to int is possible because my_arr and p are of the same base type i.e pointer to int.

✓ In general (p+i) denotes the address of the ith element and *(p+i) denotes the value of the ith element.

✓ There are some differences between the name of the array (i.e my_arr) and pointer variable (i.e p).

Pointers and 1-D arrays IX

✓ The name of the array is a constant pointer hence you can't alter it to point to some other memory location. You can't assign some other address to it nor you can apply increment/decrement operator like you do in a pointer variable.

```
1 my_arr++; // error
2 my_arr--; // error
3 my_arr = &i // error
```

✓ However, p is an ordinary pointer variable, so you can apply pointer arithmetic and even assign a new address to it.

```
1 p++; // ok
2 p--; // ok
3 p = &i // ok
```

Pointers and 2-D arrays I

✓ how you can declare a pointer to an array.

```
1 int (*p)[10];
```

✓ Here p is a pointer that can point to an array of 10 integers. In this case, the type or base type of p is a pointer to an array of 10 integers.

Note that parentheses around p are necessary, so you can't do this:
int *p[10];
here p is an array of 10 integer pointers.

✓ A pointer that points to the 0th element of an array and a pointer that points to the whole array are totally different. The following program demonstrates this concept.

Pointers and 2-D arrays II

```c
#include<stdio.h>
int main()
{
     int *p; // pointer to int
     int (*parr)[5]; // pointer to an array of 5 integers
     int my_arr[5]; // an array of 5 integers
     p = my_arr;
     parr = my_arr;
     printf("Address of p = %u\n", p );
     printf("Address of parr = %u\n", parr );
     p++;
     parr++;
     printf("\nAfter incrementing p and parr by 1 \n\n");
     printf("Address of p = %u\n", p );
     printf("Address of parr = %u\n", parr );
     printf("Address of parr = %u\n", *parr );
     return 0;
}
```

Pointers and 2-D arrays III

✓ Here p is a pointer which points to the 0th element of the array my_arr, while parr is a pointer which points to the whole array my_arr.

✓ The base type of p is of type (int *) or pointer to int and base type of parr is pointer to an array of 5 integers.

✓ Since the pointer arithmetic is performed relative to the base type of the pointer, that's why parr is incremented by 20 bytes i.e (5 x 4 = 20 bytes). On the other hand, p is incremented by 4 bytes only.

✓ The important point you need to remember about pointer to an array is this:

Whenever a pointer to an array is dereferenced we get the address (or base address) of the array to which it points.

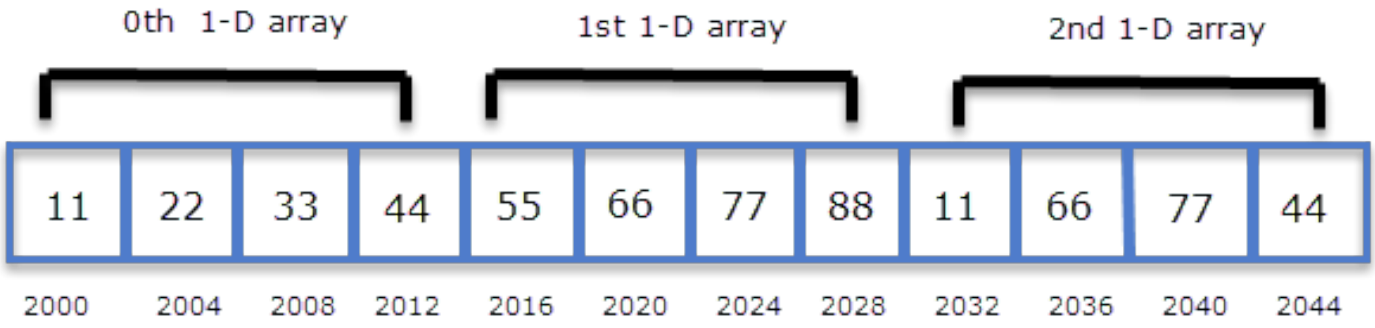
✓ So, on dereferencing parr, you will get *parr. The important thing to notice is although parr and *parr points to the same address, but parr's base type is a pointer to an array of 5 integers, while *parr base type is a pointer to int. This is an important concept and will be used to access the elements of a 2-D array.

Pointers and 2-D Array:

```
int arr[3][4] =
{

      {
            11,22,33,44
      }
      ,
      {
            55,66,77,88
      }
      ,
      {
            11,66,77,44
      }
}
;
```

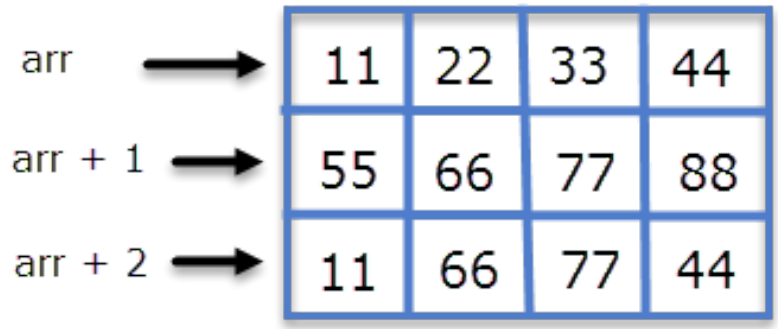
Pointers and 2-D arrays V

	Col 0	Col 1	Col 2	Col 3
Row 0	11	22	33	44
Row 1	55	66	77	88
Row 2	11	66	77	44



Pointers and 2-D arrays VI

arr points to 0th 1-D array.
(arr + 1) points to 1st 1-D array.
(arr + 2) points to 2nd 1-D array.



Pointers and 2-D arrays VII

In general, we can write:
(arr + i) points to ith 1-D array.

✓ Dereferencing a pointer to an array gives the base address of the array. So dereferencing arr we will get *arr, base type of *arr is (int*).

✓ Similarly, on dereferencing arr+1 we will get *(arr+1).

In general, we can say that: *(arr+i) points to the base address of the ith 1-D array.

✓ It is important to note that type (arr + i) and *(arr+i) points to same address but their base types are completely different. The base type of (arr + i) is a pointer to an array of 4 integers, while the base type of *(arr + i) is a pointer to int or (int*).

✓ Since *(arr + i) points to the base address of every ith 1-D array and it is of base type pointer to int, by using pointer arithmetic we should we able to access elements of ith 1-D array.

```
1 *(arr + i) points to the address of the 0th element of the 1-D array. So,\\
2 *(arr + i) + 1 points to the address of the 1st element of the 1-D array\\
3 *(arr + i) + 2 points to the address of the 2nd element of the 1-D array\\
```

✓ *(arr + i) + j points to the base address of jth element of ith 1-D array.

✓ On dereferencing *(arr + i) + j we will get the value of jth element of ith 1-D array.

Pointers and 2-D arrays VIII

```
*( *(arr + i) + j)
```

✓ The following program demonstrates how to access values and address of elements of a 2-D array using pointer notation.

```
#include<stdio.h>
int main()
{
     int arr[3][4] =
     {
          {
               11,22,33,44
          }
          ,
          {
               55,66,77,88
          }
          ,
          {
               11,66,77,44
          }
```

Pointers and 2-D arrays IX

```
17      }
18      ;
19      int i, j;
20      for(i = 0; i < 3; i++)
21      {
22           printf("Address of %d th array %u \n",i , *(arr + i));
23           for(j = 0; j < 4; j++)
24           {
25                printf("arr[%d][%d]=%d\n", i, j, *( *(arr + i) + j) );
26           }
27           printf("\n\n");
28      }
29      return 0;
30 }
```

Assigning 2-D Array to a Pointer Variable:

✓ You can assign the name of the array to a pointer variable, but unlike 1-D array you will need pointer to an array instead of pointer to int or (int *).

```
int arr[2][3] =
{
     {
          33, 44, 55
     }
     ,
     {
          11, 99, 66
     }
}
;
```

Pointers and 2-D arrays XI

✓ Always remember a 2-D array is actually a 1-D array where each element is a 1-D array. So arr as an array of 2 elements where each element is a 1-D arr of 3 integers. Hence to store the base address of arr, you will need a pointer to an array of 3 integers.

✓ Similarly, If a 2-D array has 3 rows and 4 cols i.e int arr[3][4], then you will need a pointer to an array of 4 integers.

```
1 int (*p)[3];
```

✓ Here p is a pointer to an array of 3 integers. So according to pointer arithmetic p+i points to the ith 1-D array, in other words, p+0 points to the 0th 1-D array, p+1 points to the 1st 1-D array and so on.

✓ The base type of (p+i) is a pointer to an array of 3 integers.

✓ If we dereference (p+i) then we will get the base address of ith 1-D array but now the base type of *(p + i) is a pointer to int or (int *).

✓ Again to access the address of jth element ith 1-D array, we just have to add j to *(p + i).

✓ So *(p + i) + j points to the address of jth element of ith 1-D array.

Pointers and 2-D arrays XII

✓ Therefore the expression *(*(p + i) + j) gives the value of jth element of ith 1-D array.

```
#include<stdio.h>
int main()
{
     int arr[3][4] =
     {
          {
               11,22,33,44
          }
          ,
          {
               55,66,77,88
          }
          ,
          {
               11,66,77,44
          }
     }
     ;
     int i, j;
```

Pointers and 2-D arrays XIII

```
20      int (*p)[4];
21      p = arr;
22      for(i = 0; i < 3; i++)
23      {
24           printf("Address of %d th array %u \n",i , p + i);
25           for(j = 0; j < 4; j++)
26           {
27                printf("arr[%d][%d]=%d\n", i, j, *( *(p + i) + j) );
28           }
29           printf("\n\n");
30      }
31      // signal to operating system program ran fine
32      return 0;
33 }
```

Array of Pointers in C I

Just like we can declare an array of int, float or char etc, we can also declare an array of pointers, here is the syntax to do the same.
Syntax: datatype *array_name[size];

```
int *arrop[5]
```

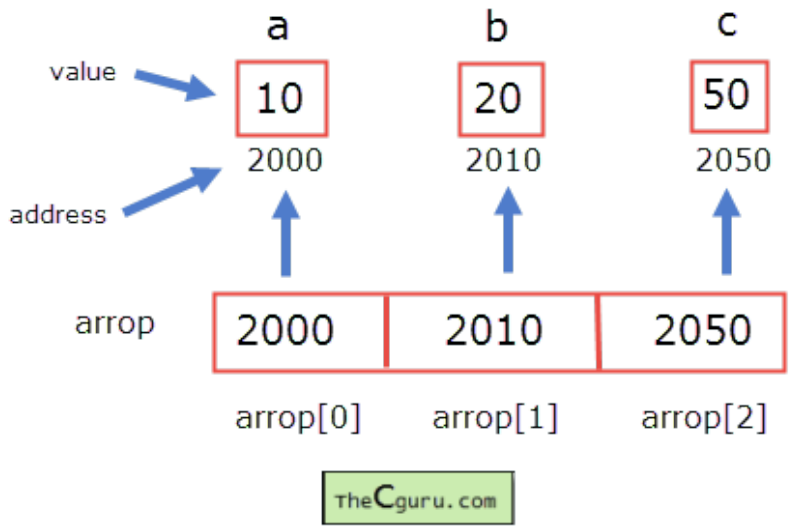
✓ Here arrop is an array of 5 integer pointers. It means that this array can hold the address of 5 integer variables. In other words, you can assign 5 pointer variables of type pointer to int to the elements of this array.

```
#include<stdio.h>
#define SIZE 10
int main()
{
     int *arrop[3];
     int a = 10, b = 20, c = 50, i;
     arrop[0] = &a;
     arrop[1] = &b;
     arrop[2] = &c;
     for(i = 0; i < 3; i++)
     {
          printf("Address = %d\t Value = %d\n", arrop[i], *arrop[i]);
```

Array of Pointers in C II

```
13      }
14      return 0;
15  }
```

✓ Notice how we are assigning the addresses of a, b and c. In line 7, we are assigning the address of variable a to the 0th element of the of the array. Similarly, the address of b and c is assigned to 1st and 2nd element respectively. At this point, the arrop looks something like this:



Array of Pointers in C III

✓ arrop[i] gives the address of ith element of the array. So arrop[0] returns address of variable a, arrop[1] returns address of b and so on. To get the value at address use indirection operator (*).

*arrop[i]

✓ So, *arrop[0] gives value at address arrop[0], Similarly *arrop[1] gives the value at address arrop[1] and so on.

Void Pointers in C I

✓ Pointer Basics in C that if a pointer is of type pointer to int or (int *) then it can hold the address of the variable of type int only.

✓ It would be incorrect, if we assign an address of a float variable to a pointer of type pointer to int.

✓ But void pointer is an exception to this rule. A void pointer can point to a variable of any data type. Here is the syntax of void pointer.

```
1 void *vp;
```

Let's take an example:

```
1 void *vp;
2
3 int a = 100, *ip;
4 float f = 12.2, *fp;
5 char ch = 'a'; </pre>
```

✓ Here vp is a void pointer, so you can assign the address of any type of variable to it.

Void Pointers in C II

```
vp = &a; // ok
vp = ip; // ok
vp = fp; // ok

ip = &f; // wrong since type of ip is pointer to int
fp = ip; // wrong since type of fp is pointer to float</pre>
```

✓ A void pointer can point to a variable of any data type and void pointer can be assigned to a pointer of any type.

Void Pointers in C III

Dereferencing a void Pointer:

We can't just dereference a void pointer using indirection (*) operator. For example:

```
1 void *vp;
2 int a = 100;
3
4 vp = &a;
5 printf("%d", *vp); // wrong
```

✓ It simply doesn't work that way!. Before you dereference a void pointer it must be typecasted to appropriate pointer type.

✓ For example: In the above snippet void pointer vp is pointing to the address of integer variable a. So in this case vp is acting as a pointer to int or (int *).

✓ Hence the proper typecast in this case is (int*).

```
1 (int *)vptr
```

✓ Now the type of vptr temporarily changes from void pointer to pointer to int or (int*), and we already know how to dereference a pointer to int, just precede it with indirection operator (*)

Void Pointers in C IV

```
*(int *)vptr
```

Note: typecasting changes type of vp temporarily until the evaluation of the expression, everywhere else in the program vp is still a void pointer.

✓ The following program demonstrates how to dereference a void pointer.

```
#include<stdio.h>
#define SIZE 10
int main()
{
     int i = 10;
     float f = 2.34;
     char ch = 'k';
     void *vptr;
     vptr = &i;
     printf("Value of i = %d\n", *(int *)vptr);
     vptr = &f;
     printf("Value of f = %.2f\n", *(float *)vptr);
     vptr = &ch;
```

Void Pointers in C V

```
14      printf("Value of ch = %c\n", *(char *)vptr);
15      return 0;
16  }
```

Void Pointers in C VI

Pointer Arithmetic in Void Pointers:

✓ Before you apply pointer arithmetic in void pointers make sure to provide a proper typecast first otherwise you may get unexpected results.

```
int one_d[5] =
{
     12, 19, 25, 34, 46
}
, i;
void *vp = one_d;
printf("%d", one_d + 1); // wrong
```

✓ Here we have assigned the name of the array one_d to the void pointer vp. Since the base type of one_d is a pointer to int or (int*), the void pointer vp is acting like a pointer to int or (int*). So the proper typecast is (int*).

Void Pointers in C VII

```
int one_d[5] =
{
     12, 19, 25, 34, 46
}
, i;
void *vp = one_d;

printf("%d", (int *)one_d + 1); // correct
```

✓ The following program demonstrates pointer arithmetic in void pointers.

```
#include<stdio.h>
#define SIZE 10
int main()
{
     int one_d[5] =
     {
          12, 19, 25, 34, 46
     }
     , i;
     void *vp = one_d;
```

Void Pointers in C VIII

```
11      for(i = 0; i < 5; i++)
12      {
13           printf("one_d[%d] = %d\n", i, *( (int *)vp + i ) );
14      }
15      return 0;
16  }
```

The void pointers are used extensively in dynamic memory allocation

The malloc() Function in C I

✓ In static memory allocation the size of the program is fixed, we can not increase or decrease size while the program is running.

✓ It is used to allocate memory at run time. The syntax of the function is:

```
Syntax: void *malloc(size_t size);
```

✓ This function accepts a single argument called size which is of type size_t.

✓ The size_t is defined as unsigned int in stdlib.h, for now, you can think of it as an alias to unsigned int.

✓ If successful, malloc() returns a void pointer to the first allocated byte of memory.

✓ Before you can use the pointer you must cast it to appropriate type. So malloc() function is generally used as follows:

```
p = (datatype *)malloc(size);
```

✓ where the p is a pointer of type (datatype *) and size is memory space in bytes you want to allocate.

Let's take a simple example:

✓ Suppose we want to allocate 20 bytes(for storing 5 integers, where the size of each integer is 4 bytes) dynamically using malloc(). Here is how we can do it:

The malloc() Function in C II

```
int *p; // p is pointer to int or (int*)
p = (int*)malloc(20); // allocate 20 bytes
```

✓ As we know the size of data types in C vary from system to system, that's why malloc() function is used in conjunction with the sizeof operator.

```
int *p; // p is pointer to int or (int*)
p = (int*)malloc(5*sizeof(int)); // allocate sufficient memory for 5 integers
```

✓ Now we have p pointing to the first byte of allocated memory, we can easily access subsequent bytes using pointer arithmetic.

✓ When the heap runs out of free space, malloc() function returns NULL.

✓ So before using the pointer variable in any way, we must first always check the value returned by malloc() function.

```
if(p == NULL)
{
     printf("Memory allocation failed");
     exit(1); // exit the program
}
```

The malloc() Function in C III

Calculate the average marks of students in a class using the malloc() function.

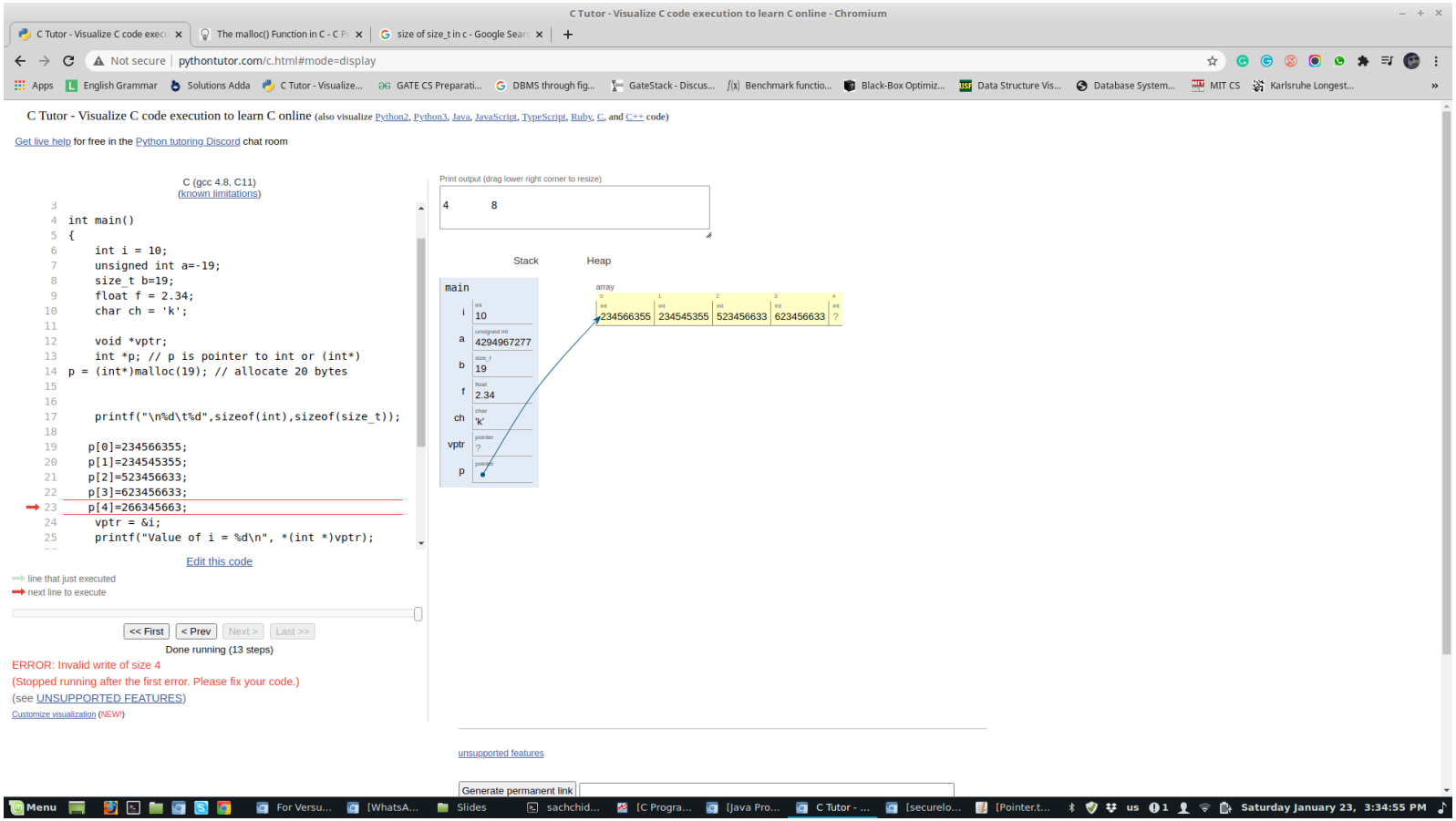
```c
#include<stdio.h>
#include<stdlib.h>

int main()
{
     float *p, sum = 0;
     int i, n;

     printf("Enter the number of students: ");
     scanf("%d", &n);

     // allocate memory to store n variables of type float
     p = (float*)malloc(n*sizeof(float));
     //Q- What if malloc() is not void type?
     // if dynamic allocation failed exit the program
     if(p==NULL)
     {
          printf("Memory allocation failed");
          exit(1); // exit the program
     }
```

The malloc() Function in C IV

```
21
22      // ask the student to enter marks
23      for(i = 0; i < n; i++)
24      {
25           printf("Enter marks for %d student: ", i+1);
26           scanf("%f", p+i);
27           //Q--> What if we use p++?
28      }
29
30      // calculate sum
31      for(i = 0; i < n; i++)
32      {
33           sum += *(p+i);
34      }
35
36      printf("\nAverage marks = %.2f\n", sum/n);
37
38      // signal to operating system program ran fine
39      return 0;
40 }
```

The malloc() Function in C V

In this figure malloc allocated only 19 bytes of memory. each block will be assigned 4 bytes except last block. Last block will get only 3 bytes. Therefore, we can not assign a number bigger than 3 bytes in the last block. So, we should avoid using direct number. Always try to use with a data type.



The calloc() Function in C I

✓ C provides another function to dynamically allocate memory which sometimes better than the malloc() function. Its syntax is:

```
Syntax: void *calloc(size_t n, size_t size);
```

✓ It accepts two arguments the first argument is the number of the element, and the second argument is the size of elements. Let's say we want to allocate memory for 5 integers, in this case, 5 is the number of elements i.e n and the size of each element is 4 bytes (may vary from system to system). Here is how you can allocate memory for 5 integers using calloc().

```
int *p;
p = (int*)calloc(5, 4);

int *p;
p = (int*)calloc(5, sizeof(int));
```

✓ The difference between calloc() and malloc() function is that memory allocated by malloc() contains garbage value while memory allocated by calloc() is always initialized to 0.

The calloc() Function in C II

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

int main()
{
     int *p, i, n;

     printf("Enter the size of the array: ");
     scanf("%d", &n);

     p = (int*)calloc(n, sizeof(int));

     if(p==NULL)
     {
          printf("Memory allocation failed");
          exit(1); // exit the program
     }

     for(i = 0; i < n; i++)
     {
          printf("Enter %d element: ", i);
```

The calloc() Function in C III

```
22            scanf("%d", p+i);
23        }
24        printf("\nprinting array of %d integers\n\n", n);
25
26        // calculate sum
27
28        for(i = 0; i < n; i++)
29        {
30            printf("%d ", *(p+i));
31        }
32        // signal to operating system program ran fine
33        return 0;
34    }
```

The realloc() Function in C I

✓ Let's say we have allocated some memory using malloc() and calloc(), but later we find that memory is too large or too small. The realloc() function is used to resize allocated memory without losing old data. It's syntax is:

Syntax: void *realloc(void *ptr, size_t newsize);

✓ The realloc() function accepts two arguments, the first argument ptr is a pointer to the first byte of memory that was previously allocated using malloc() or calloc() function.

✓ The newsize parameter specifies the new size of the block in bytes, which may be smaller or larger than the original size.

✓ And size_t is just an alias of unsigned int defined inside stdlib.h header file.

```
1 int *p;
2 p = (int*)malloc(5*sizeof(int)); // allocate memory for 5 integers</pre>
```

✓ Suppose that sometimes later we want to increase the size of the allocated memory to store 6 more integers. To do that, we have to allocate additional 6 x sizeof(int) bytes of memory. Here is how you will call realloc() function to allocate 6 x sizeof(int) bytes of memory.

The realloc() Function in C II

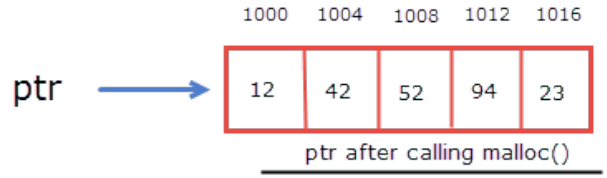
```
// allocate memory for 6 more integers integers i.e a total of 11 integers
p = (int*)realloc(p, 11*sizeof(int));
```

✓ If sufficient memory (in this case 6 * sizeof(int) bytes) is available following already used bytes then realloc() function allocates only allocates 6 * sizeof(int) bytes next to already used bytes. In this case, the memory pointed to by ptr doesn't change. It is important to note that in doing so old data is not lost but newly allocated bytes are uninitialized.

✓ On the hand, if sufficient memory (in this case 6 * sizeof(int) bytes) is not available following already used bytes then realloc() re-allocates entire 11 * sizeof(int) bytes of memory somewhere else in the heap and copies the content from the old memory block to the new memory block. In this case, address pointed to by ptr changes.

The realloc() Function in C III

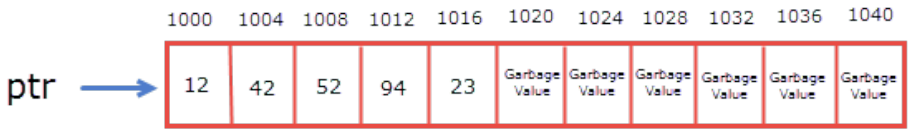
p = (int)malloc(5*sizeof(int));*



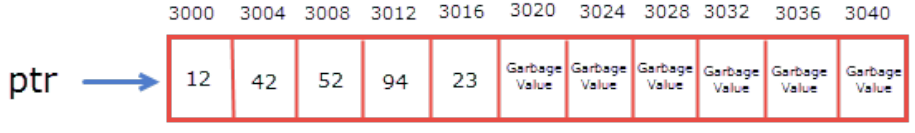
p = (int)realloc(p, 11*sizeof(int));*

Now two conditions may arise:

1st case: If sufficient memory is available after address 1016, then the address of ptr doesn't change.



2nd case: If sufficient memory is not available after address 1016, then the realloc() function allocates memory somewhere else in the heap and copies the all content from old memory block to the new memory block. In this case the address of ptr changes.



TheCguru.com

The realloc() Function in C IV

If realloc() failed to expand memory as requested then it returns NULL, the data in the old memory remains unaffected.

The following program demonstrates the realloc() function.

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
     int *p, i, n;
     printf("Initial size of the array is 4\n\n");
     p = (int*)calloc(4, sizeof(int));
     if(p==NULL)
     {
          printf("Memory allocation failed");
          exit(1); // exit the program
     }
     for(i = 0; i < 4; i++)
     {
          printf("Enter element at index %d: ", i);
          scanf("%d", p+i);
```

The realloc() Function in C V

```
17      }
18      printf("\nIncreasing the size of the array by 5 elements ...\n ");
19      p = (int*)realloc(p, 9 * sizeof(int));
20      if(p==NULL)
21      {
22           printf("Memory allocation failed");
23           exit(1); // exit the program
24      }
25      printf("\nEnter 5 more integers\n\n");
26      for(i = 4; i < 9; i++)
27      {
28           printf("Enter element at index %d: ", i);
29           scanf("%d", p+i);
30      }
31      printf("\nFinal array: \n\n");
32      for(i = 0; i < 9; i++)
33      {
34           printf("%d ", *(p+i) );
35      }
36      return 0;
37 }
```

The realloc() Function in C VI