Pointers

PART 2

Pointers and Functions

A pointer variable can be passed as a parameter to a function either by value or by reference. To declare a pointer as a value parameter in a function heading, you use the same mechanism as you use to declare a variable. To make a formal parameter be a reference parameter, you use & when you declare the formal parameter in the function heading. Therefore, to declare a formal parameter as a reference pointer parameter, between the data type name and the identifier name, you must include * to make the identifier a pointer and & to make it a reference parameter. The obvious question is: In what order should & and * appear between the data type name and the identifier to declare a pointer as a reference parameter? In C++, to make a pointer a reference parameter in a function heading, * appears before the & between the data type name and the identifier.

Pass value to a pointer

```
int global_Var = 42;
// function to change pointer value
void changePointerValue(int* pp)
{
   pp = &global_Var;
}
int main()
{
   int var = 23;
   int* ptr_to_var = &var;
   cout << "Passing Pointer to function:" << endl;
   cout << "Before :" << *ptr_to_var << endl; // display 23
   changePointerValue(ptr_to_var);
   cout << "After :" << *ptr_to_var << endl; // display 23
   return 0;
}
```

Reference to a pointer

```
int global_Var = 42;
// function to change pointer value
void changePointerValue(int* &pp)
{
   pp = &global_Var;
}
int main()
{
   int var = 23;
   int* ptr_to_var = &var;
   cout << "Passing Pointer to function:" << endl;
   cout << "Before :" << *ptr_to_var << endl; // display 23
   changePointerValue(ptr_to_var);
   cout << "After :" << *ptr_to_var << endl; // display 23
   return 0;
}
```

Returning a pointer from a function

```
#include <iostream>
using namespace std;
int global_var = 42;
// function to return a pointer
int* returnPointerValue()
{
   return &global_var;
}
int main()
{
   int var = 23;
   int* ptr_to_var = &var;
   cout << "Return a pointer from a function " << endl;
   cout << "Before :" << *ptr_to_var << endl; // display 23
   ptr_to_var = returnPointerValue();
   cout << "After :" << *ptr_to_var << endl; // display 42
   return 0;
}
```

Returning reference from function

```
#include <iostream>
using namespace std;
int global_var = 42;
// function to return reference value
int& ReturnReference()
{
   return global_var;
}
int main()
{
   int var = 23;
   int* ptr_to_var = &var;
   cout << "Returning a Reference " << endl;
   cout << "Before :" << *ptr_to_var << endl; // display 23
   ptr_to_var = &ReturnReference();
   cout << "After :" << *ptr_to_var << endl; // display 42
   return 0;
}
```

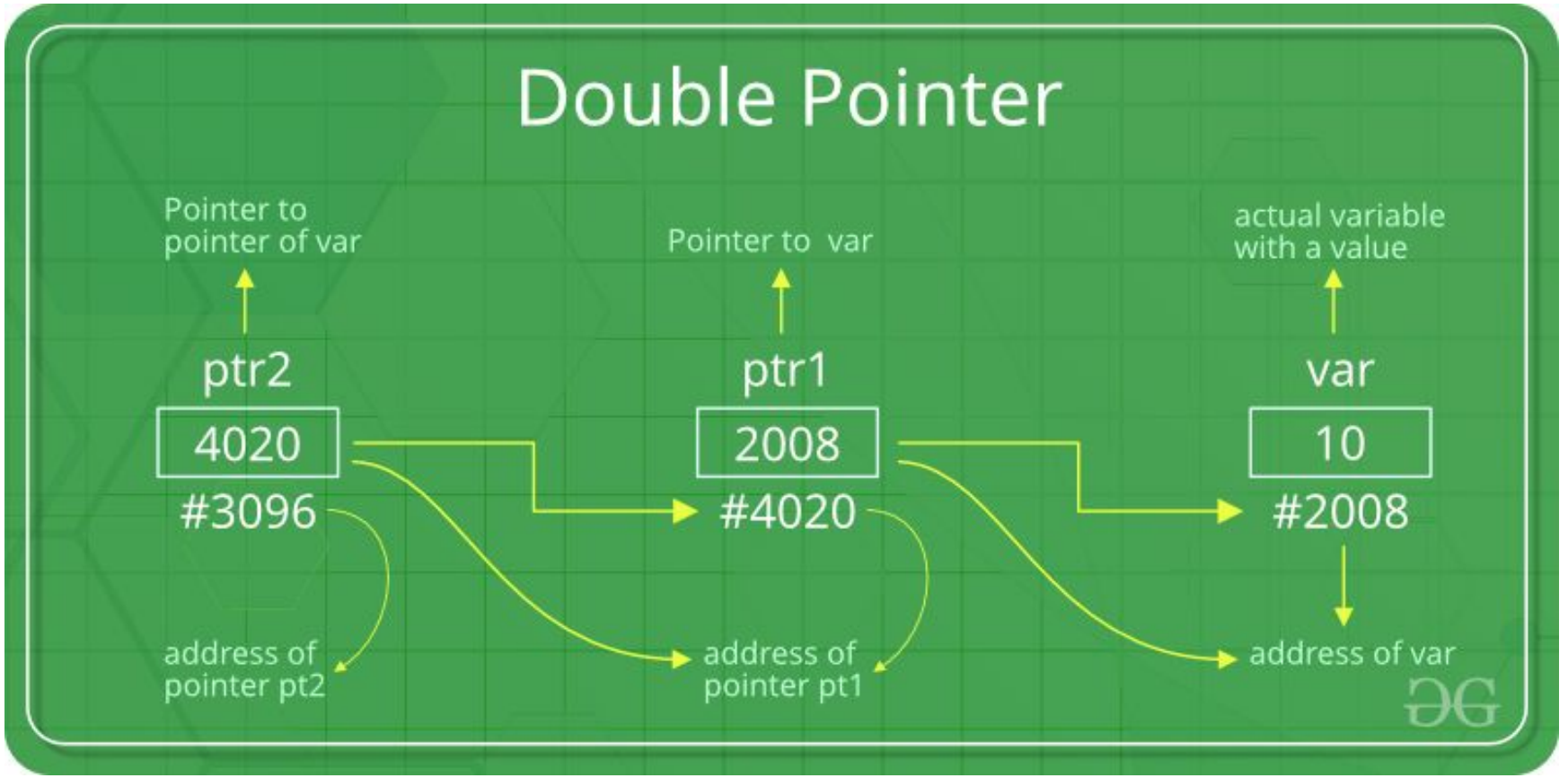
C++ Pointer to Pointer (Multiple Indirection)

A pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

Declaration to declare a pointer to a pointer of type int –

```
int **var;
```

C++ Pointer to Pointer (Multiple Indirection)



Code

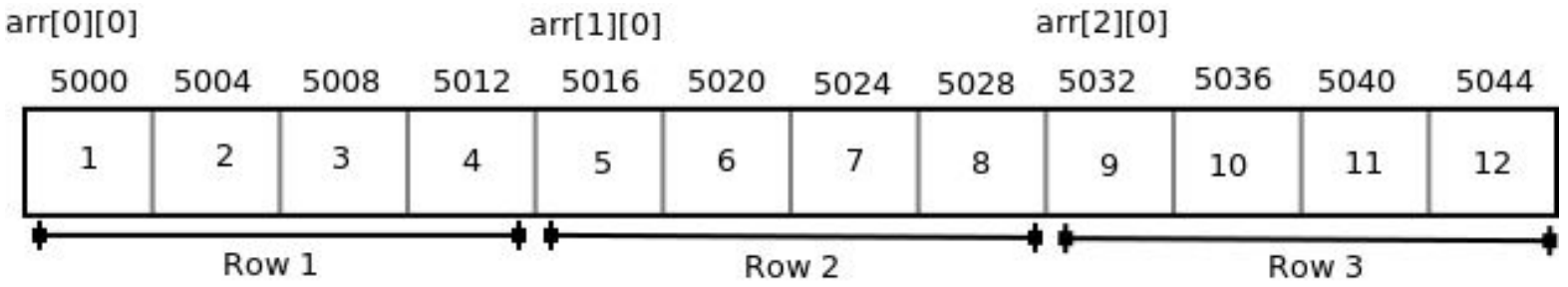
```
#include <iostream>
using namespace std;
int main() {
int  var;
int* ptr;
int** pptr;
var = 3000;
// take the address of var
ptr = &var;
// take the address of ptr using address of operator &
pptr = &ptr;
// take the value using pptr
cout << "Value of var :" << var << endl;
cout << "Value available at *ptr :" << *ptr << endl;
cout << "Value available at **pptr :" << **pptr << endl;
return 0;
}
```

Pointers to 2D array

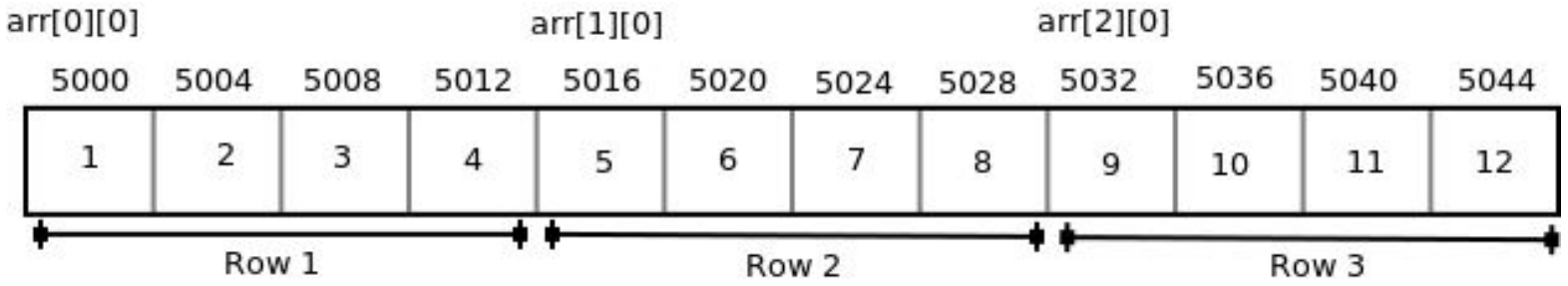
- int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };

	Col 1	Col 2	Col 3	Col 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

Since memory in a computer is organized linearly it is not possible to store the 2-D array in rows and columns. The concept of rows and columns is only theoretical, actually, a 2-D array is stored in row-major order i.e rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory.



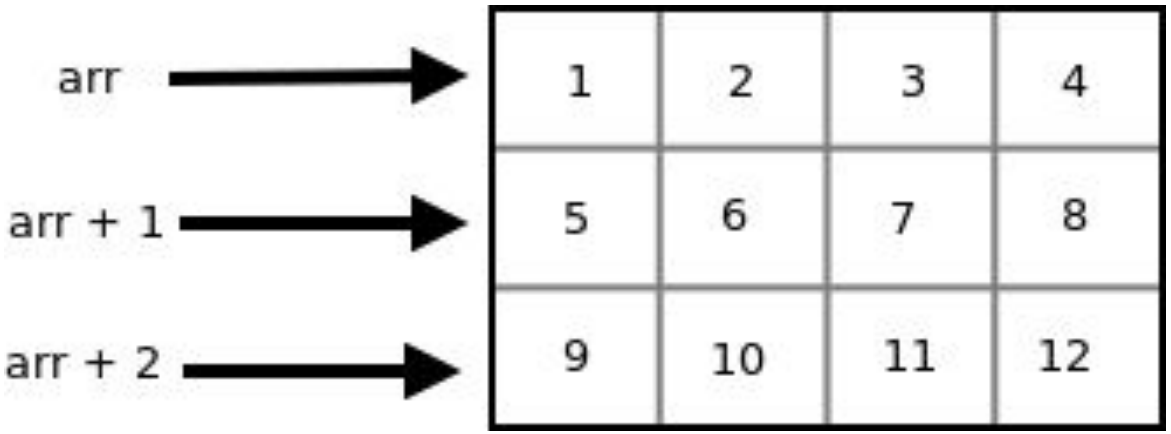
Pointers to 2D array



Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another.
So here *arr* is an array of 3 elements where each element is a 1-D array of 4 integers.

Pointers to 2D array

Since *arr* is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression arr + 1 will represent the address 5016 and expression arr + 2 will represent address 5032.
So we can say that *arr* points to the 0^{th} 1-D array, *arr + 1* points to the 1^{st} 1-D array and *arr + 2* points to the 2^{nd} 1-D array.



arr - Points to 0^{th} element of arr - Points to 0^{th} 1-D array - 5000
arr + 1 - Points to 1^{th} element of arr - Points to 1^{nd} 1-D array - 5016
arr + 2 - Points to 2^{th} element of arr - Points to 2^{nd} 1-D array - 5032

arr	Points to 0^{th} 1-D array
*arr	Points to 0^{th} element of 0^{th} 1-D array
(arr + i)	Points to i^{th} 1-D array
*(arr + i)	Points to 0^{th} element of i^{th} 1-D array
*(arr + i) + j)	Points to j^{th} element of i^{th} 1-D array
((arr + i) + j)	Reprents the value of j^{th} element of i^{th} 1-D array

- To access an individual element of our 2-D array, we should be able to access any jth element of ith 1-D array.
- Since the base type of *(arr + i) is int and it contains the address of 0th element of ith 1-D array, we can get the addresses of subsequent elements in the ith 1-D array by adding integer values to *(arr + i).
- For example *(arr + i) + 1 will represent the address of 1^{st} element of 1^{st} element of ith 1-D array and *(arr+i)+2 will represent the address of 2nd element of ith 1-D array.
- Similarly *(arr + i) + j will represent the address of jth element of ith 1-D array. On dereferencing this expression we can get the jth element of the ith 1-D array.

```
#include <iostream>
using namespace std;
int main()
{
int arr[3][4] = {
  { 10, 11, 12, 13 },
  { 20, 21, 22, 23 },
  { 30, 31, 32, 33 }
};
int i, j;
for (i = 0; i < 3; i++)
{
cout << i<<" "<<arr[i]<<" "<< * (arr + i) << endl;

for (j = 0; j < 4; j++)
cout << arr[i][j] <<" " << *(*(arr + i) + j) << endl;
cout << "\n";
}
return 0;
}
```

Dynamic two Dimensional Array

1. **Create a pointer to a pointer variable.**

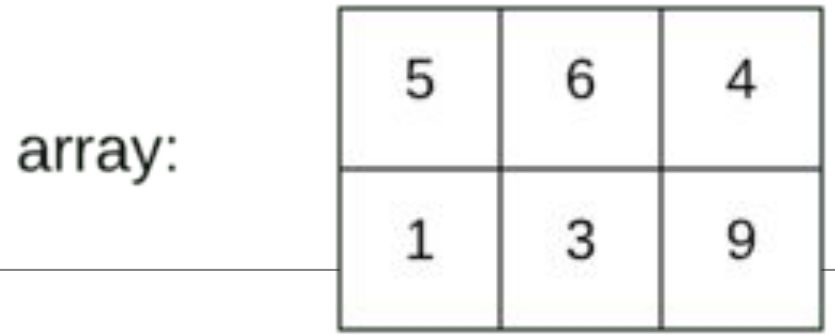
```
1.  int** arry;
```

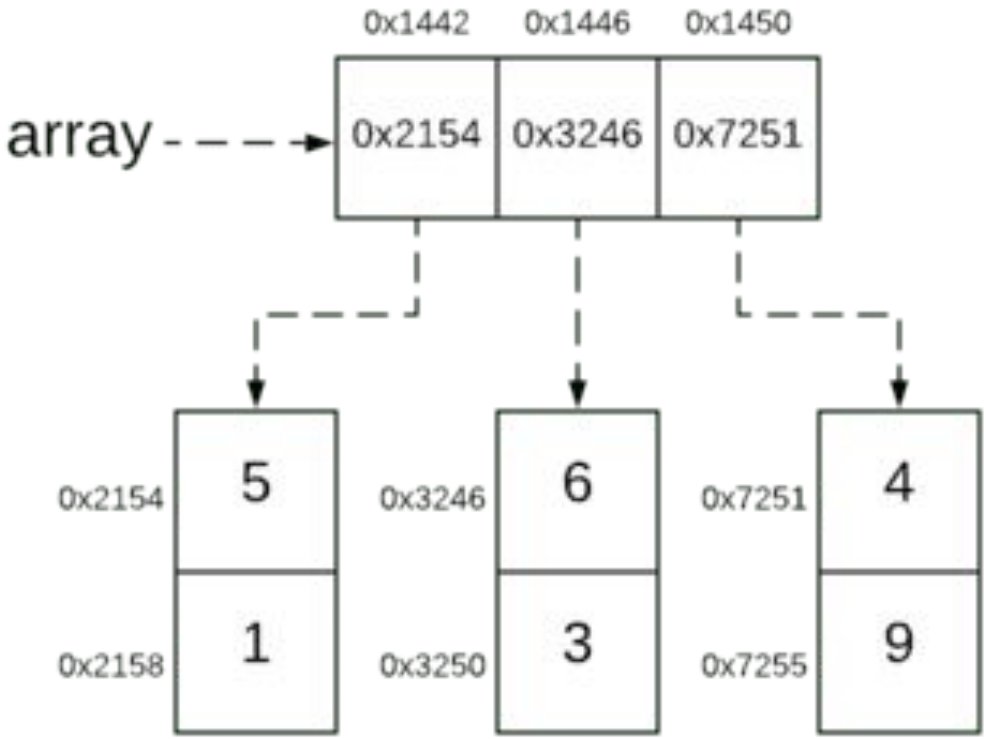
2. **Allocate memory using the `new` operator for the array of pointers that will store the reference to arrays.**

```
1.  arry = new int*[row];
```

3. **By using a loop, we will allocate memory to each row of the 2D array.**

```
1.  for(int i = 0;i < row;i++) {
2.      arry[i] = new int[col];
3.  }
```


array:
5
6
4
1
3
9


0x1442
0x1446
0x1450
array
0x2154
0x3246
0x7251
0x2154
5
1
0x2158
0x3246
6
3
0x3250
0x7251
4
9
0x7255

```
#include <iostream>
using namespace std;
int main() {
   int** arry;
   int row, col, i, j;
   cout << "Number of Rows:" << endl;
   cin >> row;
   cout << "Number of Columns:" << endl;
   cin >> col;
   //Allocating the row space in heap dynamically
   arry = new int* [row];
   //Allocating the column space in heap dynamically
   for (i = 0; i < row; i++) {
      arry[i] = new int[col];
   }
   //Giving inputs to the array
   cout << "Enter " << (row * col) << " numbers to the Array\n";
   for (i = 0; i < row; i++) {
      for (j = 0; j < col; j++) {
         cout << "Enter the elements at Row " << i + 1 << " Column " << j +
1 << endl;
         cin >> arry[i][j];
      }
   }
   //Display the array
   cout << "Here is your 2D Array:" << endl;
   for (i = 0; i < row; i++) {
      for (j = 0; j < col; j++) {
         cout << arry[i][j] << ' ';
      }
      cout << endl;
   }
   //Free the memory after the use of array
   for (i = 0; i < row; i++) {
      delete[] arry[i];
   }
   delete[] arry;
   return 0;
}
```

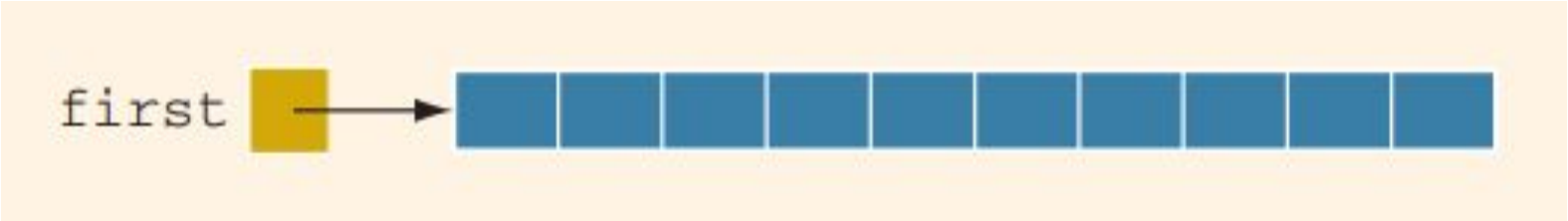
Shallow vs Deep copy and Pointers

Consider the following statements:

int *first;

int *second;

first = new int[10];



The first two statements declare first and second pointer variables of type int. The

third statement creates an array of 10 components, and the base address of the array is

stored into first

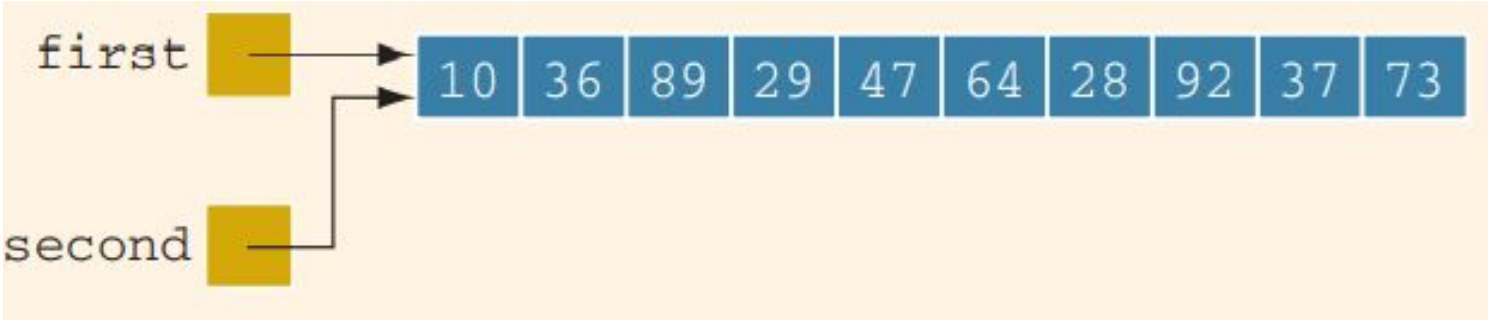
Shallow vs Deep copy and Pointers

Next, consider the following statement:

second = first; //Line A

This statement copies the value of first into second. After this statement executes,

both first and second point to the same array



Shallow vs Deep copy and Pointers

Let us next execute the following statement:

delete [] second;

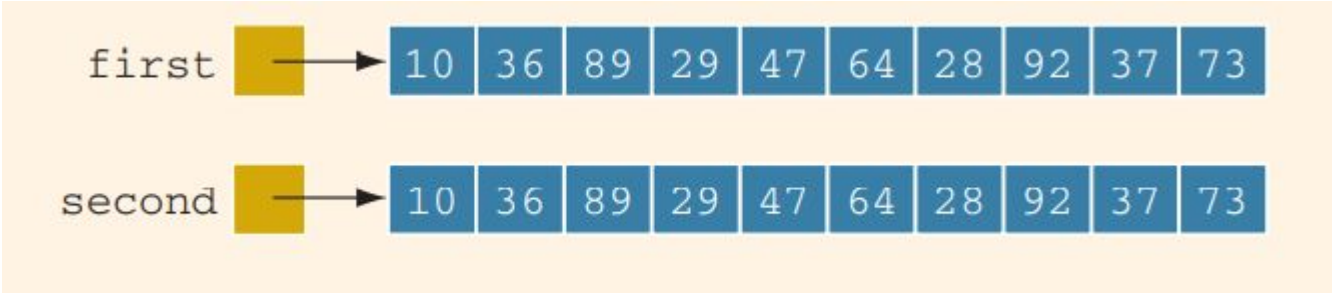
After this statement executes, the array pointed to by second is deleted.

Because first and second point to the same array, after the statement: delete [] second; executes, first becomes invalid, that is, first (as well as second) are now dangling pointers. Therefore, if the program later tries to access the memory pointed to by first, either the program will access the wrong memory or it will terminate in an error. This case is an example of a shallow copy. More formally, in a shallow copy, two or more pointers of the same type point to the same memory; that is, they point to the same data.

Deep Copy

```
second = new int[10];

for (int j = 0; j < 10; j++)
        second[j] = first[j];
```



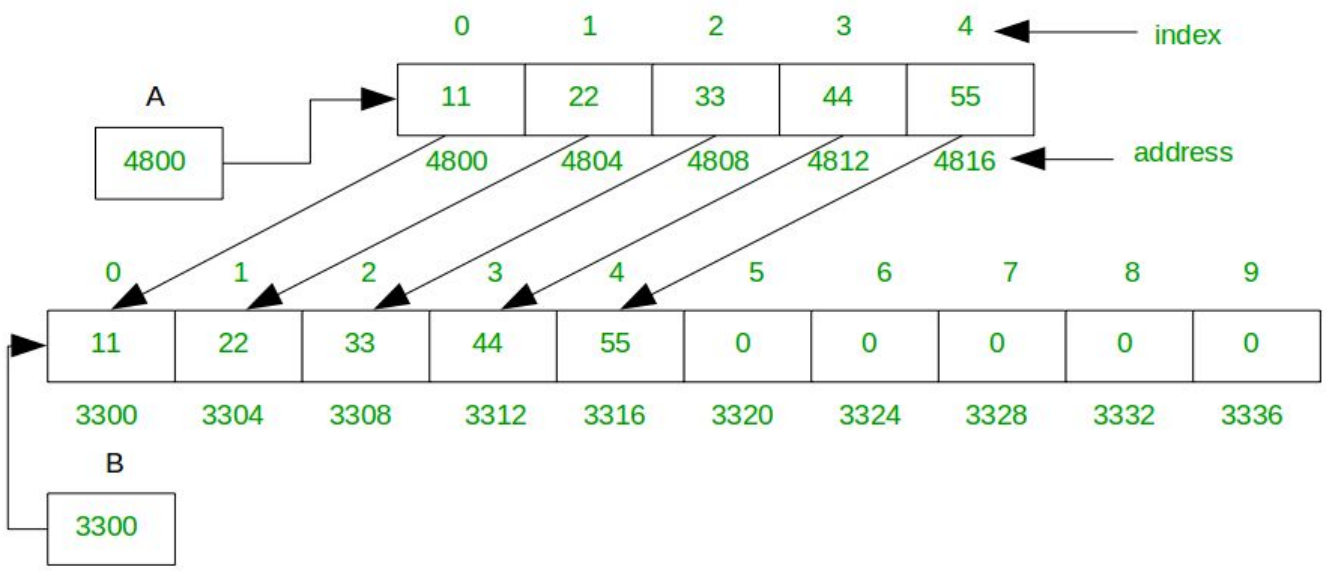
Both first and second now point to their own data. If second deletes its memory, there is no effect on first. This case is an example of a deep copy. More formally, in a deep copy, two or more pointers have their own data

How do Dynamic arrays work?

Approach: When we enter an element in array but array is full then you create a function, this function creates a new array double size or as you wish and copy all element from the previous array to a new array and return this new array. Also, we can reduce the size of the array. and add an element at a given position, remove the element at the end default and at the position also.

Grow Array

Add Element: Add element at the end if the array size is not enough then extend the size of the array and add an element at the end of the original array as well as given index.

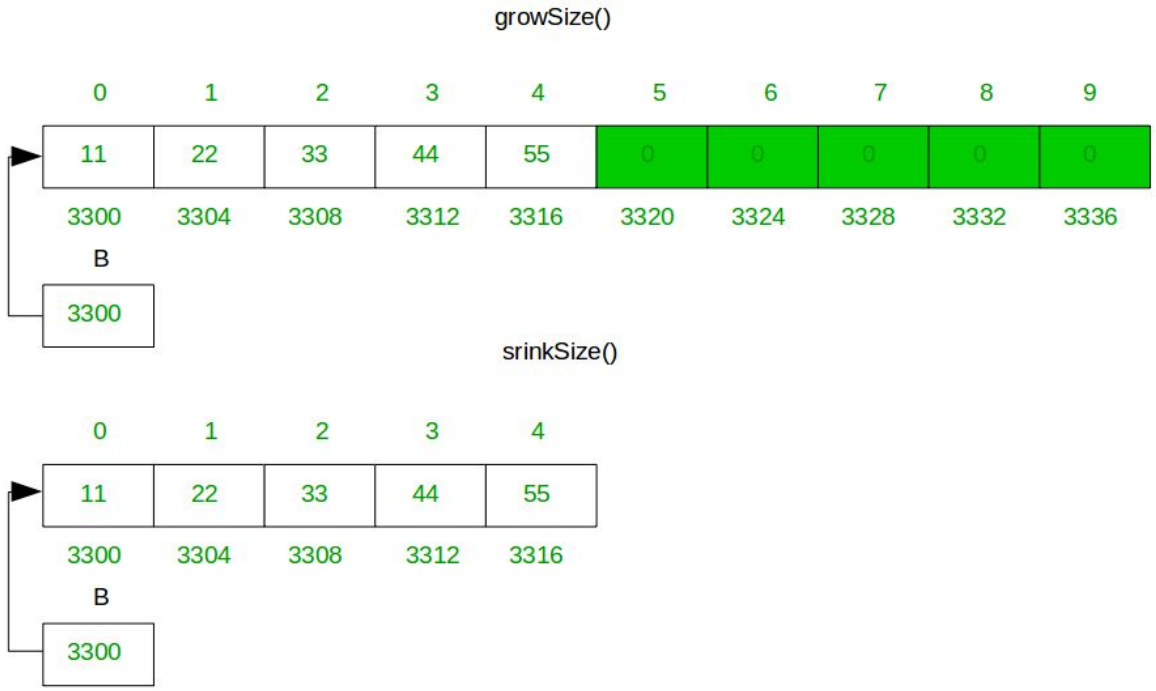


Resize of Array Size:

When the array has null/zero data (aside from an element added by you) at the right side of the array, meaning it has unused memory, the method shrinkSize() can free up the extra memory.

When all space is consumed, and an additional element is to be added, then the underlying fixed-size array needs to increase in size.

Typically resizing is expensive because you have to allocate a bigger array and copy over all of the elements from the array you have outgrown before we can finally



Dynamically resizing an array

If you have an existing array, and you want to make it bigger (add array cells to it), you cannot simply append new cells to the old ones. Remember that arrays are stored in consecutive memory, and you never know whether or not the memory immediately after the array is already allocated for something else. For that reason, the process takes a few more steps. Here is an example using an integer array. Let's say this is the original array:

```
int * list = new int[size];
```

I want to resize this so that the array called list has space for 5 more numbers (presumably because the old one is full). There are four main steps.

Create an entirely new array of the appropriate type and of the new size. (You'll need another pointer for this).

```
int * temp = new int[size + 5];
```

Copy the data from the old array into the new array (keeping them in the same positions). This is easy with a for-loop.

```
for (int i = 0; i < size; i++)
   temp[i] = list[i];
```

Delete the old array -- you don't need it anymore! (Do as your Mom says, and take out the garbage!)

```
delete [] list;  // this deletes the array pointed to by "list"
```

Change the pointer. You still want the array to be called "list" (its original name), so change the list pointer to the new address.

```
list = temp;
```

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
   int max = 5;
   int* a = new int[max];                    // allocated on heap
   int n = 0;
   //--- Read into the array
   while (cin >> a[n]) {
      n++;
      if (n >= max) {
         max = max * 2;                      // double the previous size
         int* temp = new int[max];           // create new bigger array.
         for (int i = 0; i < n; i++) {
            temp[i] = a[i];                  // copy values to new array.
         }
         delete[] a;                         // free old array memory.
         a = temp;                           // now a points to new array.
      }
   }
   for (int i = 0; i < n; i++)
   {
      cout << a[i] << endl;
   }
}
```