Arrays

- Passing two-dimensional array to a function

```
#include <iostream>
using namespace std;
double f(double values[][4], int n);

int main() {
	double beans[3][4] = {
		{ 1.0,  2.0,  3.0,  4.0 },
		{ 5.0,  6.0,  7.0,  8.0 },
		{ 9.0, 10.0, 11.0, 12.0 }
	};

	cout << f(beans, sizeof beans / sizeof beans[0]) << endl;
	return 0;
}

double f(double array[][4], int size) {
	double sum = 0.0;
	for (int i = 0; i < size; i++)
		for (int j = 0; j < 4; j++)
			sum += array[i][j];
	return sum;

}
```

Pointers

- Pointers hold a memory address
  - Memory address: long
- Value of memory address is a hexadecimal number
- Declare a pointer
  - int *ptr; //pointer
    EX)
    int x = 10;
    ptr=&x; //=200
    cout<<x; //prints 10
    cout<<&x //prints 200
    cout<<&pts; //shows 100 (address for pointer)
    cout<<ptr; //200
    cout<<*ptr; //10
    x=x+5;
    cout<<x //15
    cout<<*ptr //15

  - ptr returns memory address
  - &ptr returns memory adress for ptr
  - *ptr returns value

- Uninitialized pointer

```
int *ptr;
*ptr = 10; //corrupts the value somewhere in the program
cout<<*ptr;
```

- Initialized pointer

```
int *ptr;
int x = 20;
ptr=&x; //initialize pointer
*ptr=10 //ok
cout<<x<<endl; //10
x=x+5;
cout<<*ptr; //15
```

- Null pointer

```
int *ptr=0 //ptr points to nothing
int *ptr = null; //ptr points to nothing
int *ptr = nullptr; //only c++ 11
*ptr =10 //error
```

- Reference (variable)
  - Reference is an alias, or an alternative name, to an existing variable
  - Type & refVal = existingVariable

```
int x = 10;
int &refX = x;
cout<<x; //10
cout << &x; //100
cout<<refX; //10
cout <<&refX; //100
```

- **Reference vs pointer**
  - A reference is a name constant for an address
  - Once a reference is established to a variabe you cannon change the reference to reference another variable

```
int num1=88;
int num2=22;
int *ptrnum1 = &num1;
cout<<*ptrnum1<<endl; //88
cout<<&ptrnum1;  //300
cout<<&num1; //100
cout<<ptrnum1 //100
ptrnum1=&num2;
cout<<*ptrnum1; //22
num1=num+15 //num <---103
cout<<*ptrnum1; //22
double z =2.5;
*ptrnum1=z; //error not int
int n1=30
int &refn1=n1;
cout<<n1; //30
cout<<refn1; //30
cout<<&n1; //155
```

```
int n2=5;
refn1 = &n2 //error, references are constant
```

- **Call-by-value**

```
int square (int);
int main()
{
        int number=8;
        cout<<"In main: "<<&number<<endl; //200
        cout<<square(number)<<endl; //64
        cout << number<<endl; //8
}
int square(int n)
{
        cout<< "In Square: "<<&n<<endl; //300
        n*=n;
        return n;
}
```

- **Pass by reference with pointer argument**

```
void square(int *)
int main()
{
        int number=8;
        cout<<"In main: "<<&number<<endl; //100
        square (&number);
        cout<<number;//64
        return 0;
}
void Square (int *n)
{
        cout<<"In Square: "<<n<<endl;//8
        *n = *n  *  *n;
        return ;
}
```

- **Pass by reference with reference argument**

```
int square (int &)
int main()
{
        int number = 8;
        cout<<"In Main: "<<&number<<endl; //100
        cout<<square(number)<<endl; //implicitly
        cout<<number<<endl; //64
        return 0;
}
int square (int &n)
{
        cout<<"in Square: "<<&n<<endl;
```

```
    n *= n;
    return n;
}
```

"Const" function reference/pointer parameter

- **A const function parameter cannot be modified in a function. A const function parameter can receive both const and non const arguments**

```
int test (const int);
int main()
{
    int number=8;
    const int n1 = 3;
    cout<<test(number);
    cout<<test(n1);
    return 0;
}

int test (const int n)
{
    n = n*n; //error!
    return n*n;
}
```

- **A non- const function reference/point argument parameter can only receive non-const arguments**

```
int square (int &n)
{
    return n*n;
}
int main ()
{
    int number = 8;
    const int n1=3
    cout<<square(number); //64
    cout<<square(n1); //error, cannot use const
    return 0;
}
//OR
int square (int *n)
{
    return *n * *n;
}
int main ()
{
    int number = 8;
    const int n1 = 3;
    cout<<square(number); //64
```

```
        cout<<square(n1); // error
        return 0;
    }
```

Const function Reference/pointer parameter

```
square (const int & n)
{
    n = n*n //error
    return n*n;
}
int main ()
{
    int number = 8;
    const int n1=3;
    cout << square (number); //64
    cout <<square (n1);//9
    return 0;
}
```

Pointers and Arrays

```
// arrnote.cpp
// array accessed with array notation
#include <iostream>
using namespace std;

int main()
   {                                        //array
   int intarray[5] = { 31, 54, 77, 52, 93 };

   for(int j=0; j<5; j++)                   //for each element,
      cout << intarray[j] << endl;          //print value
   return 0;


   }

// array accessed with pointer notation
#include <iostream>
using namespace std;

int main()
   {                                        //array
   int intarray[5] = { 31, 54, 77, 52, 93 };

   for(int j=0; j<5; j++)                   //for each element,
      cout << *(intarray+j) << endl;        //print value
   return 0;


   }

// passarr.cpp
// array passed by pointer
```

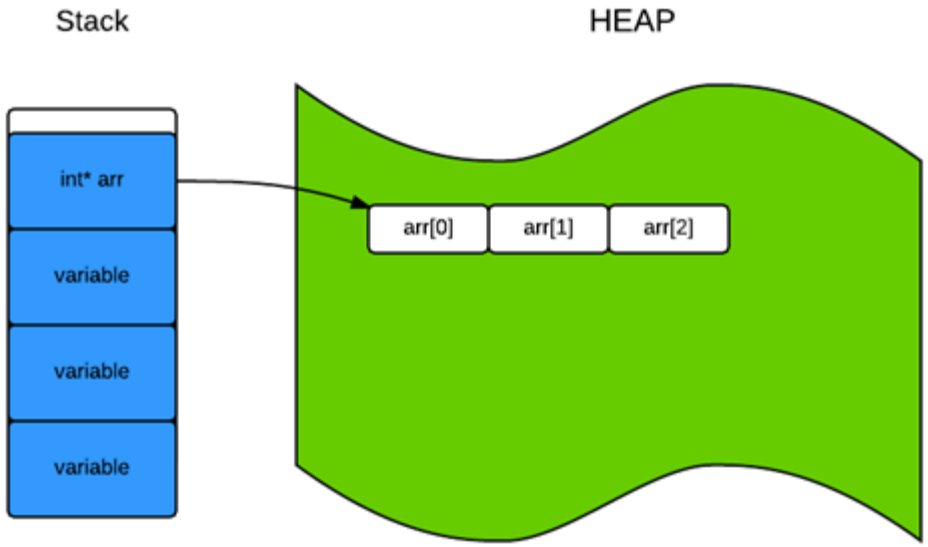
```
#include <iostream>
using namespace std;
const int MAX = 5;           //number of array elements

int main()
   {
   void centimize(double*);  //prototype

   double varray[MAX] = { 10.0, 43.1, 95.9, 59.7, 87.3 };

   centimize(varray);        //change elements of varray to cm

   for(int j=0; j<MAX; j++)  //display new array values
      cout << "varray[" << j << "]="
           << varray[j] << " centimeters" << endl;
   return 0;
   }
//-------------------------------------------------------------
void centimize(double* ptrd)
   {
   for(int j=0; j<MAX; j++)
      *ptrd++ *= 2.54;       //ptrd points to elements of varray
   }
```

Use pointer with two dimensional array

```
#include <iostream>
#include <iomanip>
#include <cctype>
using namespace std;

int main()

       const int table = 12;
       long values[table][table] = { 0 };

       for (int i = 0; i < table; i++)
              for (int j = 0; j < table; j++)
                     *(*(values + i) + j) = i*j;


       for (int i = 0; i < table; i++) {
              for (int j = 0; j < table; j++)
                     cout << " " << setw(3) << values[i][j] << " |";
              cout << endl;
       }
       return 0;

}
```

Dynamic Memory

The **heap** is a region of computer's memory, used for dynamic memory allocation. When you use dynamic allocation, all the created variables are stored into heap, Heap memory is not managed automatically. When you use dynamic memory allocation, a pointer that is located in stack points to the region of the allocated memory in heap:



For dynamic memory allocation, C++ offers operator **new.** Operator new returns the pointer to the newly allocated space. If you want to allocate memory for one single element of a specified data type (it can be a built in data type, structure or class), you will have to use operator new in the following form:

```
new data_type;
```

If you want to allocate memory for an array, you will have to use another form of operator new:

```
new data_type[size_of_array];
```

The common scheme for dynamic memory allocation consists of two parts:

Declare a pointer.

Allocate needed amount of memory.

```
int* arr;//pointer to int
int n;//number of elements

cout << "Please, enter the number of elements for input" << endl;
cin >> n; // get n
```

```
arr = new int[n];//allocate memory for array of int of size n

//get user's input
cout << "Enter " << n << " elements" << endl;

//get elements in loop
for (int i = 0; i != n; ++i)
     cin >> arr[i];

cout << "You entered :" << endl;

for (int i = 0; i != n; ++i)
     cout << "arr[" << i << "] = " << arr[i] << endl;
```

In the case when operator new fails to allocate memory, exception of type **bad_alloc** is thrown. There is a possibility to use "no throw" version of the operator new. In this case, you have to use the following syntax:

```
new (nothrow) data_type
```

Operator new will not throw exception even if it cannot allocate memory. It will simply return an empty pointer.

Once you do not need memory allocated by operator **new,** you have to release it**.** You can do it by using operator **delete:**

delete pointer; for single object and

delete[] pointer; for an array of objects

For example, we can free memory, allocated for array **arr** from the example above:

```
delete[] arr;
```

C-String manipulation

C++ provides following two types of string representations:

- The C-style character string.
- The string class type introduced with Standard C++.

The C-Style Character String:

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

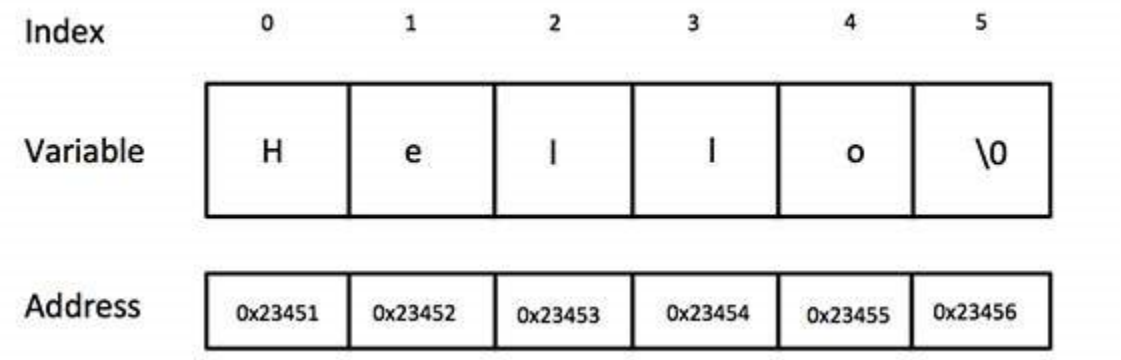
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++:



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string:

```
#include <iostream>

using namespace std;

int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

   cout << "Greeting message: ";
   cout << greeting << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Greeting message: Hello
```

C++ supports a wide range of functions that manipulate null-terminated strings:

Function & Purpose

strcpy(s1, s2);

1 Copies string s2 into string s1.

strcat(s1, s2);

2 Concatenates string s2 onto the end of string s1.

strlen(s1);

3 Returns the length of string s1.

strcmp(s1, s2);

4 Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

strchr(s1, ch);

5 Returns a pointer to the first occurrence of character ch in string s1.

strstr(s1, s2);

6 Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions:

```
#include <iostream>
#include <cstring>

using namespace std;

int main ()
{
   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;

   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;

   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;

   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld

strlen(str1) : 10
```

C string manipulation

Write a function that returns the number of digits in a given null-terminated string.

```
#include<iostream>
#include<cctype>
using namespace std;
int numAlphas(const char* s)
{
	int count = 0;
	for (int i = 0; s[i] != '\0'; i++)
	{
		if (isdigit(s[i]))
		{
			count++;
		}
	}
	return count;
}

int main()
{
	char str[] = "a12bc3d";
	cout << numAlphas(str);

}
```

C Strings and Pointers

```
// Create your own strlen function
#include <iostream>
using namespace std;
int myStrLen(char str[]);

int main()
{
	char s[15] = "Hello World";
	cout << myStrLen(s);
	return 0;
}
//------------------------------------------------------------
int myStrLen(char str[])
{
	int i = 0;
	while (str[i] != '\0')
		i++;
	return i;
}
```

Or

```
int myStrLen(char *str)
{
	char *first = str;
	while (*str != '\0')
		str++;
	return str - first;
}
```

Or

```
int myStrLen(char *str)
{
	char *first = str;
	while (*str)
		str++;
	return str - first;

}

// create your own strcpy function
#include <iostream>
using namespace std;
void myStrcpy(char str2[], char str1[]);

int main()
{
	char s1[15] = "Hello World";
	char s2[30];
	myStrcpy(s2, s1);
	cout << s2;
	return 0;
}
//-----------------------------------------------------------------
void myStrcpy(char *to, char * from)
{
	while (*to = *from)
	{
		to++;
		from++;
	}

}
```

Or

```
void myStrcpy(char *to, char * from)
{
	while (*to++ = *from++);

}
```