Roxana Dumitrescu

C++ in Financial Mathematics

What have we learnt?

- Flow of control
  - Conditional flow of control: if, switch
  - Loops: for, while, do-while

What have we learnt?

- Functions
  - Predefined (built-in) functions - examples: pow, sqrt, floor (in order to use them, you have to include **cmath**)
  - User defined functions:
    - Value returning function

```
double max (double a, double b);
```

    - Void function

```
void print (double a);
```

What have we learnt?

- Functions
  - Oveloading functions - you can write functions with the same name, but different signatures!

```
double max (double a, double b);
int max (int a, int b);
```

  - Scope of the variable: global and local
  - Static variables (with local scope)

Plan

- Pointers to variables
- References
- Pointers to variables /references
- Passing arguments to functions using pointers/references
- Returning references/pointers
- Pointers to pointers
- Pointers to functions
- Static arrays/dynamic arrays
- C-style strings

Pointers, References, Arrays, Strings

Pointers

What is a pointer?

A pointer variable is a variable which stores a memory address. This address can be a location of one of the following in memory:

- Variable
- Pointer
- Function

Pointers are a tool for directly manipulating computer memory.

Pointers

Declaring pointer variable

Pointers must be declared before they can be used, like a normal variable. A pointer is associated with a type (such as an *int* or *double*).

SYNTAX

```
data_type * ptr;
```

This means: we declare a pointer variable called **ptr** as a pointer of *data_type*.

Pointers

```
int * p_1; // Declare a pointer variable
 called p_1 pointing to an int (or int
 pointer).
double *d; // Declare a double pointer
int * p_1, *p_2, j; // p_1 and p_2 are int
 pointers, j is an int.
```

Pointers

Initializing Pointers via the Address-of Operator (&)

When we declare a pointer, its content is not initialized. It can be initiliazed be assigning it a valid address. This could be done using the *address-of operator(&)*.

The *address-of operator(&)* operates on a variable, and returns the address of the variable.

```
 int * pNumber; // Declare a pointer variable
  called pNumber pointing to an int.

int number; // Declare an int variable and
  associate it a value
pNumber=&number; // Assign the address of the
  variable number to pointer pNumber
```

Pointers

Dereferencing Operator (*)

The *indirection operator* (or *dereferencing operator*) (*) operates on a pointer and returns the value stored at the address kept in the pointer variable.

```
int * pNumber; // Declare a pointer variable
   called pNumber pointing to an int.
int number=20; // Declare an int variable and
   associate it a value
pNumber=&number; // Assign the address of the
   variable number to pointer pNumber
cout<<*pNumber<<endl; // Print the value
   "pointed to" by the pointer, which is the
   int 88.
*pNumber=99; // Assign a new value to where
   the pointer points to, NOT to the pointer.
```

Pointers

Remarks

- We have modified the value of the variable *number*, through the pointer *pNumber*.
- The *indirection operator* $(*)$ can be used in both the RHS - right hand side - (t=*pNumber) and the LHS - left hand side - (*pNumber=99) of an assignment statement.
- Note that the symbol $(*)$ has different meaning in a declaration statement and in an expression.
  - Used in a declaration (int *p), it denotes that p is a pointer variable.
  - When it is used in an expression (*p=99, cout«*p), it refers to the value pointed by *p*.

Pointers

Pointer has a Type!

A pointer is associated with a type (the one of the value it points to), which is specified during declaration. **A pointer can hold an address of the declared type; it can't hold an address of a different type.**

```
int i=88;
double d=55;
int * iPtr=&i;
double *dPtr=&d;
iPtr=&d; //ERROR
dPtr=&i; //ERROR
iPtr=i;      //ERROR
int j=99;
iPtr=&j; // Change the address stored by
iPtr
```

Dynamic Memory Allocation

One can allocate memory at run time for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

When you do not need dynamically allocated memory anymore, you have to use **delete** operator, which de-allocates memory previously allocated by new operator.

Dynamic Memory Allocation

The example we have seen:

```
int number=5;
int *p=& number;  // Assign a "valid" address
   into a pointer
```

New and Delete Operators

```
// Dynamic allocation
int * p2; // Not initialized
p2=new int; // Dynamically allocate an int and
   assign its address to a pointer
* p2=99;

delete p2; // Remove the dynamically allocated
   storage
```

Dynamic Memory Allocation

New and Delete Operators

Note that we can write:

```
int * p1=new int(88);
...
```

At the address indicated by $p1 \mapsto 88$.

In the case of **dynamic allocation**: **the programmer handles the memory allocation** and deallocation via **new** and **delete** operators.

Pointers and const

Pointers and const

Pointing to const variables

Consider the program:

```
int x=7;
int *ptr=&x;
*ptr=6; //change value to 6
```

What happens if *x* is const?

```
const int x=7; // x is constant
int *ptr=&x; // compiler error: cannot convert
   const int* to int*
*ptr=6; //change value to 6
```

Pointers and const

Pointing to const variables

The above code doesn't compile - we can't set a non-const pointer to a const variable.

Explanation

A const variable is one whose value cannot be changed. If we could set a non-const pointer to a const value, then we would be able to dereference the non-const pointer and change the value. This would violate the intention of const.

Pointers and const

Pointer to const variables

A *pointer to a const value* is a (non-const) pointer that points to a const value. To declare a pointer to a const value, use the *const* keyword before the data type:

```
const int x=7; // x is constant
const int *ptr=&x; // OK, ptr is pointing to a
   "const int"
*ptr=6; //not OK, we cannot change a const
   value
```

Pointers and const

Pointer to const variables

Consider now the following example.

```
int x=7; // x is not constant
const int *ptr=&x; // it is OK
```

A pointer to a const variable can point to a non-const variable (such as variable *x*).

Explanation

A pointer to a constant variable treats the variable as constant when it is accessed through the pointer, regardless of whether the variable was initially defined as const or not.

Pointers and const

Pointer to const variables

Consequently, the following example is OK:

```
int x=7; //
const int *ptr=&x; // ptr points to a "const
    int"
x=6;
```

but the following is **not OK**:

```
int x=7; // x is not constant
const int *ptr=&x; // ptr points to a const int
*ptr=6;// ptr treats its value as const, so
    changing the value through ptr is not legal
```

Pointers and const

Pointer to const variables

Remark: The pointer points to a const value, but it is not const itself! In this case, the pointer can be redirected to point at other values:

```
int x_1=7; //
const int *ptr=&x_1; // ptr points to a const
    int
int x_2=6;
ptr=&x_2; // OK, ptr points now at some other
    const int
```

Pointers and const

Const pointers

A const pointer is a pointer whose value can not be changed after initialization. To declare a const pointer, use the *const* keyword between the asterisk and the pointer name:

```
int x=5;
int *const ptr=&x;
```

Like a normal const variable, a const pointer must be initialized to a value upon declaration. In other words, a const pointer will always point to the same address. In our example, ptr will always point to the address of *x*, until ptr goes out of scope and is destroyed.

Pointers and const

Const pointers

```
int x_1=5;
int x_2=6;

int * const ptr=&x_1; //OK, the const pointer
   is initialized to the address of x_1
ptr=&x_2; // not OK, once initialized a const
   pointer can not be changed
```

Pointers and const

Const pointers

Because the *value* being pointed to is still non-const, it is possible to change the value pointed to via dereferencing the const pointer:

```
int x_1=5;
int * const ptr=&x_1; //ptr will always point
   to value
*ptr=6; // OK, since ptr points to a non-const
   int
```

Pointers and const

Const pointers to a const value

It is possible to declare a const pointer to a const value by using the *const* keyword both before the type and before the variable name:

```
int x=5;
const int * const ptr=&x;
```

A const pointer to a const value can not be set to point to another address, nor can the value it is pointing to be changed through the pointer.

Pointers and const

Const pointers to a const value

It is possible to declare a const pointer to a const value by using the *const* keyword both before the type and before the variable name:

```
int x=5;
const int * const ptr=&x;
```

A const pointer to a const value can not be set to point to another address, nor can the value it is pointing to be changed through the pointer.

References

References

What is a reference?

A reference variable is an alias = another name for an already existing variable.
Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

References

Recall that we have denoted the *address-of* operator by &. C++ assigns an additional meaning to the operator & in the declaration of references variables.
In conclusion:

- When it is used in an expression, & denotes the address-of operator and is used to return the address of a variable
- When & is used in a declaration, it is part of the type identifier and is used to declare a *reference variable*.

SYNTAX

```
type & newName= existingName;
```

References

```
int main() { int number=88;
int & refNumber=number; // Declare a reference
   (alias) to the variable number
        int *ptr=&number;
cout<<number<<endl; // Print value of variable
   number (88)
cout<<refNumber<<endl; // Print value of
   reference (88)
refNumber=99;
cout<<refNumber<<endl;
cout<<number<<endl; // Value of number also
   changed.}
```

References

number and **refnumber** both refer to the same location. Unlike the refnumber reference, the ptr pointer requires a storage space to store the address of number variable to which it points. To get the value of number variable, either the ptr pointer or refNumber reference can be used!

References

Another example

```
void g()
{
int ii=0;
int& r=ii;
r++;
int* pp=&rr;
}
```

- *r* increments by 1 the value of *ii*
- *pp* points to the object referenced by the reference rr (note that we can write &rr).

References versus Pointers

Pointers and references are equivalent, but there are some differences:

1. You need to initialize the reference at the declaration. Once a reference is established to a variable, **you cannot change the reference to reference another variable**. This is not the case for pointers.

```
int & iRef; // Error: 'iRef' declared as
   reference but not initialized.
```

   The correct code is:

```
int x;
int & iRef=x;
```

2. A valid reference must refer to an object; a pointer does not need. A pointer, even a const pointer, can have a null value. A null pointer doesn't point to anything.

To retain:

- We have seen pointers which hold the address of a variable.

```
int x;
int * p; // declaration
p=&x; // p stores the address of x
```

- References represent an alias (another name) for a variable.

```
int x;
int & p=x; // declaration: p refers to x
p++; // it is the same as x++
```

Remark: doing p++ for a reference changes the value of x. Doing p++ in the case of a pointer doesn't affect the value of *x*.

Passing arguments to function C++
Returning references/pointers

How can we pass arguments to functions C++?

There are three ways of passing the arguments to functions:

- By value;
- By Reference - with pointer arguments
- By Reference - with reference arguments

How can we pass arguments to functions C++?

I. By value

When the argument is passed into functions *by value*, a clone copy of the argument is made and passed into the function. Changes to the clone copy inside the function have **NO EFFECT** on the original argument in the caller.

How can we pass arguments to functions C++?

I. By value

```
void swap(int a, int b){
     int temp;
     temp=a;
     a=b;
     b=temp;}
int main() {
     int x=5;
     int y=10;
     swap(x,y);
     cout<<x<<" "<<y;
     return 0;}
```

The program prints out: x=5; y=10!

How can we pass arguments to functions C++?

II. By pointers

C++ allows to pass a pointer to a function. To do this, you have only to declare the function parameter as a pointer type.
We give an example where we pass two int pointers to a function which interchange their values: **the result reflects back in the calling function**:

```
void swap(int* a, int* b){
     int temp;
     temp=*a;
     *a=*b;
     *b=temp;}
```

How can we pass arguments to functions C++?

II. By pointers

```
int main() {
    int x=5;
    int y=10;
    swap(&x,&y);
    cout<<x<<" "<<y;
    return 0;}
```

The program prints out: x=10; y=5.
The changes are "operated" at the addreses of *x* and *y* $\Rightarrow$ *x* and *y* **are modified**!

How can we pass arguments to functions C++?

II. By reference

```
void swap(int & a, int & b){
     int temp;
     temp=a;
      a=b;
     b=temp;}
int main() {
     int x=5;
     int y=10;
     swap(x,y);
     cout<<x<<" "<<y;
     return 0;}
```

The program prints out: x=10; y=5.
The changes are "operated" at the addreses of *x* and *y* $\Rightarrow$ *x* and *y* **are modified**!

How can we pass arguments to functions C++?

Remark: There is very little practical difference between passing data using a pointer and passing data using a reference. In the C language, you have to use pass by pointer because the concept of reference does not exist. In the C++ language, using pass by reference is the preferred approach.

Can we return references or pointers?

You cannot return a reference to a local variable!

```
int & squarePtr (int number) {
int Result=number*number;
return Result;}
```

I have a warning message: "Reference to stack memory associated with local variable 'Result' returned". Take warning messages as errors!

Can we return references or pointers?

Exercise

```
int& f(int & a)
{    a=a+5;
     return a;}
int main(){
 int a=5;
 for (int i=0;i<2;i++)
     { f(a)++;}
     cout<<f(a);
     return 0;}
```

Which is the value printed out by the program? Answer:22.

- Function *f* returns a reference; in this case it is OK because it does not return a reference to a local variable.
- We can write $f(a)++$.

Can we return references or pointers?

You cannot return a pointer to a local variable!

```
int * squarePtr (int number) {
int Result=number*number;
return & Result;}
```

I have a warning message: "Address of stack memory associated with local variable 'Result' returned".

Pointer to pointers

Pointers to pointers

Pointer to pointers

- A pointer to a pointer is a form of multiple indirection or a chain of pointers.
- The first pointer contains the address of the second pointer, which points to the location that contains the actual value.

Declaration:

```
int **p;
```

When a value is indirectly pointed to by a pointer to a pointer, in order to access the value the asterisk operator should be applied twice.

Pointer to pointers

```
int main()
{ int var;
int *ptr;
int **pptr;
var=3000;
// take the address of var
ptr=&val;
// take the address of ptr using the address
   of opeartor &
pptr=&ptr;
```

Pointer to pointers

```
// 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;
}
```

Results:

```
Value of var: 3000
Value available at *ptr: 3000
Value available at **ptr: 3000
```

Function pointers

Function pointers

We have seen that a pointer can hold the address of a variable or of a pointer! Function pointers are similar, except that instead of pointing to variables, they point to functions!
Consider the function

```
int g()
{
return 2;
}
```

- Identifier *g* is the function's name.
- The function returns an integer and has no parameters.

Function pointers

How to create a pointer to a function?

```
int (*h)();
```

h is a pointer to a function that has no parameters and returns an integer. *h* can point to any function that matches this declaration!

In order to make a const function pointer, the const goes after the asterisk!

```
int (*const h)();
```

Function pointers

How to assign a function to a pointer function?

```
int f()
{ return 2;}
int g()
{return 4;}
int main()
{       int (*h)()=f; // h points to f
h=g;// h now points to g
return 0;}
```

h is a pointer to a function that has no parameters and returns an integer. *h* can point to *f* and *g*!

Function pointers

Be careful: the type of the function pointer (parameters and return type) must match the one of the fonction.

```
// function prototypes
int f();
double g();
int h(int x);
// function pointers
int (*fPtr1)()=f; // OK
int (*fPtr2)()=g; // NOT OK - return types
   don't match
double (*fPtr3)()=g; //OK
fPtr1=h; // NOT OK - fPtr1 has no parameters,
   h has parameters
int (*fPtr4)(int)=h // OK
```

Function pointers

How to call a fonction using a function pointer?

```
int f(int x){return x;}
int main()
{ int (*fPtr)(int)=f;
  fPtr(5);  // call function f through thr
   pointer fPtr;
  return 0;
}
```

The fonction name is a pointer to the function, so you don't need to dereference using $*$. For the same reason, you don't have to use the symbol & in order to get the address of the function, as in the case of variables!

Arrays

Arrays

An array is a series of elements of the same type placed in contigous memory locations that can be invidually referenced by adding an index to a unique identifier.
For example, five values of type int can be declared as an array without habing to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

Like any variable, an array must be declared before it is used.

```
type name[number_elements];
```

Arrays

```
// Create an unintialized of length 5
int myArray[5];
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";}
```

- Create an array of 5 integers, without initialising it.
- Run through the entries and print them out.
- The entries start at 0.
- We use [] to access entries.
- There is no size function.

Arrays

```
// Create an initialised array
int myArray[] = {1, 1, 2, 3, 5};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We can initialise an array by specifying the values.
- Simply place the values in a comma separated list between curly brackets.
- Notice that we no longer have to specify the length of the array when we create it.

Arrays

```
// Create an initialised array
int myArray[] = {1, 1, 2, 3, 5};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We can initialise an array by specifying the values.
- Simply place the values in a comma separated list between curly brackets.
- Notice that we no longer have to specify the length of the array when we create it.

Arrays

```
// Create an initialised array to 0
int myArray[5] = {0};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We specify the size of the array.
- We assign it the value {0}.
- This gives an array of the desired length full of zeros.

Arrays

```
// Create a general initialised array
int myArray[5] = {1,2,3};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- This prints out the values 1, 2, 3, 0, 0.
- The length of the array is specified.
- Some of the values are specified; the rest is padded with zero.

Arrays

Passing arrays to functions

```
    int sumArray( int toSum[], int length ) {
    int sum = 0;
    for (int i=0; i<length; i++) {
        sum+=toSum[i];
    }
    return sum;
}
```

- One problem with arrays is that because we don't have a function which gives automatically their **size**, we must pass their length to a function. So, the functions receive as parameters the array and its length.

Arrays

How to call sumArray function in the main program?

```
int main()
{ int n=5;
int a[5]={1,2,3,4,5};
cout<<sumArray(a,n);
}
```

- The function call is **sumArray(a,n)**.

Arrays

Don't return arrays!

- Do NOT return arrays from functions.
- The caller receives a pointer to *where the array used to be*. The computer may have reused that memory for almost anything.
- If you attempt to return an array, the behaviour is undefined.

Arrays

```
int* thisFunctionReturnsAnArray(int &length) {
    /* This produces a compiler warning */
    length=5;
    int array[5] = {1,2,3,4,5};
    return array;
}

void testDontReturnArrays() {
    int length=0;
    int* b =
   thisFunctionReturnsAnArray(length);
    for (int i=0;i<length;i++)
    cout << b[i]<<" ";
    cout << "\n";}
}
```

Arrays

Don’t return arrays!

```
int main()
{ testDontReturnArrays();
return 0;
}
```

I have a warning message: "Address of stack memory associated with local variable array returned".

Execute the program $\mapsto$ strange values!

Arrays

You can't vary the length of an array!

- You cannot change the length of an array.
- You cannot insert a new item or add some at the end.
- In fact the size is fixed AT COMPILE TIME!

Arrays

Multi-dimensional arrays

```
// Create an initialised 3x5 array
int myArray[][5] = {{1, 2, 3, 4, 5},
                    {2, 0, 0, 0, 0},
                    {3, 0, 0, 0, 0}};
for (int i=0; i<3; i++) {
    for (int j=0; j<5; j++) {
        cout<<"Entry ("<<i<<","<<j<<")=";
        cout<<myArray[i][j];
        cout<<"\n";
    }
}
```

- RULE: You have to write explicitly the last dimension!

Arrays

Remark: In addition to accessing array elements using subscripts, array elements can also be accessed using pointers. Because an array name returns the starting address of the array (the address of the first element of the array), *an array name can also be used as a pointer to the array.*

```
int array[5]={0,1,2,3,4}
```

How to access the elements?

- Array-indexing:

```
   array[0] // first element
   array[1] // second element
   array[2] // third element
...
```

Arrays

- Pointer notation:

```
   *array; // first element
   *(array+1) // second element
   *(array+2) // third element
...
```

Arrays

Arrays and pointers

Because arrays are not flexible enough, we can work with pointers! This allows to work with sequences of data of varying lengths.

```
int n = 5;
int* myArray = new int[n];
for (int i=0; i<n; i++) {
    cout<<"Entry "<<i<<"=";
    cout << myArray[i];
    cout << "\n";
}
delete[] myArray;
```

Arrays

Arrays and pointers

- **int * myArray** contains the memory address where the array starts.
- We use the **new ...[]** operator to allocate a chunk of memory. We are creating a sequence of int data types data in memory, but you can use other types of data instead.
- You can choose the size at runtime.
- The memory crated will **NOT** be automatically deleted when the function exits.

Arrays

Arrays and pointers

- You must use **delete []** operator to manually delete everything you create with the new[] operator. As we'll see, this is good and bad.
  - With arrays we couldn't return arrays because the memory was deleted automatically, but we don't have to remember to call **delete[]**.
  - With memory created using **new[]** we have to remember to delete the memory by hand, but you can safely return the data.

Arrays

Arrays and pointers

```
int sumUsingPointer( int* toSum, int length ) {
    int sum = 0;
    for (int i=0; i<length; i++) {
        sum+=toSum[i];
    }
    return sum;
}
```

- We specify the type of the parameter as int * .
- The code here is identical to that with arrays except that we declare the type using * rather than [] .
- Note that you have to pass the number of elements as well as the pointer.

Arrays

Arrays and pointers

Returning arrays dynamically allocated

- We have seen that you should never return arrays from functions! If you do it, the code will behave unpredictably. It probably will print some junk if you run it.
- You are allowed to return a pointer created with `new []` , but then you'll have to make sure the caller knows whether or not they will be expected to call `delete[]` at some point.
- By convention in C and C++, if a function returns a pointer, the caller is **NOT** expected to call `delete[]` .

Arrays

```
int* thisFunctionReturnsAPointer(int& n) {
    int* ret = new int[n];
    for (int i=0;i<n;i++) ret[i]=i;
    return ret;
}

void usingReturnPointerFunction() {
    int n=5;
    int* b= thisFunctionReturnsAPointer(n);
    for (int i=0;i<n;i++) cout<<b[i];
   // free the memory
    delete[] text;
}
```

This violates the convention on NOT deleting the return value of a function, so it is considered to be confusing code.

Arrays

Looping with pointers

```
int sumUsingForAndPlusPlus( int* begin, int n)
    {
     int sum = 0;
     int* end = begin + n;
     for (int* ptr=begin; ptr!=end; ptr++) {
          sum += *ptr;
     }
     return sum;
}
```

- You can use `++` to move a pointer on to the next item.
- You can use `==` to compare pointers.
- This code is equivalent to the last one, we just use `++` instead of arithmetic.

Arrays

Looping with pointers

The previous code is equivalent to:

```
int sumUsingForAndPlusPlus( int* toSum, int n)
    {
     int sum = 0;
     for (int i=0; i<n; i++)
{sum+=toSum[i];}
     return sum;
}
```

The above code is identical to the previous one, excepting that we declare the type using * rather than [].

C Style Strings

C Style Strings

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

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 then the number of characters in the word "Hello".

C Style Strings

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

You can also write the above statement as follows:

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

C Style Strings

```
#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 the following result:

```
Greeting message: Hello
```

C Style Strings

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

- **strcpy(s1,s2)**
  Copies string *s*2 into string *s*1
- **strcat(s1,s2)**; Concatenates string *s*2 onto the end of string *s*1.
- **strlen(s1)**; Return the length of *s*1.
- **strcmp(s1,s2)**; Returns 0 if *s*1 and *s*2 are the same; less than 0 if $s1 < s2$; greater than 0 if $s1 > s2$.
- **strchr(s1,ch)**; Returns a pointer to the first occurrence of character *ch* in the string *s*1.
- **strstr(s1,s2)**;
  Returns a pointer to the first occurence of string *s*2 in string *s*1.

C Style Strings

```
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;
// concatenate str1 and str2
strcat(str1, str2);
cout<<"strcat(str1,str2):"<<str1<<endl;
// total length of str1 after concatenation
len=strlen(str1);
cout<<"strlen(str1):"<<len<<endl;
return 0;
}
```

C Style Strings

When the code is compiled and executed, it produces the following result:

```
strcpy(str3, str1): Hello
strcat(str1, str2): HelloWorld
strlen(str1): 10
```

C Style Strings

How to use pointers in order to write your own strlen function?

```
int computeLenghtOfString(const char* s )
{  int lenght=0;
   while ((*s)!=0) {
     s++;
     lenght++;
   }
          return lenghth;
}
```

C Style Strings

How to use pointers in order to write your own strlen function?

```
int computeLenghtOfString(const char* s )
{  int lenght=0;
   while ((*s)!=0) {
     s++;
     lenght++;
   }
          return lenghth;
}
```

Summing up

- Pointers/references
- Pointers to pointers
- Pointers to functions
- Static/dynamic arrays
- C-style strings