**friend Functions and friend Classes**

**friend Function**

// friend.cpp

// friend functions

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class beta; //needed for frifunc declaration

class alpha

 {

 private:

 int data;

 public:

 alpha() : data(3) { } //no-arg constructor

 friend int frifunc(alpha, beta); //friend function

 };

////////////////////////////////////////////////////////////////

class beta

 {

 private:

 int data;

 public:

 beta() : data(7) { } //no-arg constructor

 friend int frifunc(alpha, beta); //friend function

 };

////////////////////////////////////////////////////////////////

int frifunc(alpha a, beta b) //function definition

 {

 return( a.data + b.data );

 }

//--------------------------------------------------------------

int main()

 {

 alpha aa;

 beta bb;

 cout << frifunc(aa, bb) << endl; //call the function

 return 0;

 }

// nofri.cpp

// limitation to overloaded + operator

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public:

 Distance() : feet(0), inches(0.0) //constructor (no args)

 { } //constructor (one arg)

 Distance(float fltfeet) //convert float to Distance

 { //feet is integer part

 feet = static\_cast<int>(fltfeet);

 inches = 12\*(fltfeet-feet); //inches is what's left

 }

 Distance(int ft, float in) //constructor (two args)

 { feet = ft; inches = in; }

 void showdist() //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 Distance operator + (Distance);

 };

//--------------------------------------------------------------

 //add this distance to d2

Distance Distance::operator + (Distance d2) //return the sum

 {

 int f = feet + d2.feet; //add the feet

 float i = inches + d2.inches; //add the inches

 if(i >= 12.0) //if total exceeds 12.0,

 { i -= 12.0; f++; } //less 12 inches, plus 1 foot

 return Distance(f,i); //return new Distance with sum

 }

////////////////////////////////////////////////////////////////

int main()

 {

 Distance d1 = 2.5; //constructor converts

 Distance d2 = 1.25; //float feet to Distance

 Distance d3;

 cout << "\nd1 = "; d1.showdist();

 cout << "\nd2 = "; d2.showdist();

 d3 = d1 + 10.0; //distance + float: OK

 cout << "\nd3 = "; d3.showdist();

// d3 = 10.0 + d1; //float + Distance: ERROR

// cout << "\nd3 = "; d3.showdist();

 cout << endl;

 return 0;

 }

// frengl.cpp

// friend overloaded + operator

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public:

 Distance() //constructor (no args)

 { feet = 0; inches = 0.0; }

 Distance( float fltfeet ) //constructor (one arg)

 { //convert float to Distance

 feet = int(fltfeet); //feet is integer part

 inches = 12\*(fltfeet-feet); //inches is what's left

 }

 Distance(int ft, float in) //constructor (two args)

 { feet = ft; inches = in; }

 void showdist() //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 friend Distance operator + (Distance, Distance); //friend

 };

//--------------------------------------------------------------

Distance operator + (Distance d1, Distance d2) //add D1 to d2

 {

 int f = d1.feet + d2.feet; //add the feet

 float i = d1.inches + d2.inches; //add the inches

 if(i >= 12.0) //if inches exceeds 12.0,

 { i -= 12.0; f++; } //less 12 inches, plus 1 foot

 return Distance(f,i); //return new Distance with sum

 }

//--------------------------------------------------------------

int main()

 {

 Distance d1 = 2.5; //constructor converts

 Distance d2 = 1.25; //float-feet to Distance

 Distance d3;

 cout << "\nd1 = "; d1.showdist();

 cout << "\nd2 = "; d2.showdist();

 d3 = d1 + 10.0; //distance + float: OK

 cout << "\nd3 = "; d3.showdist();

 d3 = 10.0 + d1; //float + Distance: OK

 cout << "\nd3 = "; d3.showdist();

 cout << endl;

 return 0;

 }

**friends for functional notation**

// misq.cpp

// member square() function for Distance

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public: //constructor (no args)

 Distance() : feet(0), inches(0.0)

 { } //constructor (two args)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void showdist() //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 float square(); //member function

 };

//--------------------------------------------------------------

float Distance::square() //return square of

 { //this Distance

 float fltfeet = feet + inches/12; //convert to float

 float feetsqrd = fltfeet \* fltfeet; //find the square

 return feetsqrd; //return square feet

 }

////////////////////////////////////////////////////////////////

int main()

 {

 Distance dist(3, 6.0); //two-arg constructor (3'-6")

 float sqft;

 sqft = dist.square(); //return square of dist

 //display distance and square

 cout << "\nDistance = "; dist.showdist();

 cout << "\nSquare = " << sqft << " square feet\n";

 return 0;

 }

**friend class**// friclass.cpp

// friend classes

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class alpha

 {

 private:

 int data1;

 public:

 alpha() : data1(99) { } //constructor

 friend class beta; //beta is a friend class

 };

////////////////////////////////////////////////////////////////

class beta

 { //all member functions can

 public: //access private alpha data

 void func1(alpha a) { cout << "\ndata1=" << a.data1; }

 void func2(alpha a) { cout << "\ndata1=" << a.data1; }

 };

////////////////////////////////////////////////////////////////

int main()

 {

 alpha a;

 beta b;

 b.func1(a);

 b.func2(a);

 cout << endl;

 return 0;

 }

**Overloading operators**

**Overloading Binary Operators
Arithmetic operators**// englplus.cpp

// overloaded '+' operator adds two Distances

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public: //constructor (no args)

 Distance() : feet(0), inches(0.0)

 { } //constructor (two args)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void getdist() //get length from user

 {

 cout << "\nEnter feet: "; cin >> feet;

 cout << "Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 Distance operator + ( Distance ) const; //add 2 distances

 };

//--------------------------------------------------------------

 //add this distance to d2

Distance Distance::operator + (Distance d2) const //return sum

 {

 int f = feet + d2.feet; //add the feet

 float i = inches + d2.inches; //add the inches

 if(i >= 12.0) //if total exceeds 12.0,

 { //then decrease inches

 i -= 12.0; //by 12.0 and

 f++; //increase feet by 1

 } //return a temporary Distance

 return Distance(f,i); //initialized to sum

 }

////////////////////////////////////////////////////////////////

int main()

 {

 Distance dist1, dist3, dist4; //define distances

 dist1.getdist(); //get dist1 from user

 Distance dist2(11, 6.25); //define, initialize dist2

 dist3 = dist1 + dist2; //single '+' operator

 dist4 = dist1 + dist2 + dist3; //multiple '+' operators

 //display all lengths

 cout << "dist1 = "; dist1.showdist(); cout << endl;

 cout << "dist2 = "; dist2.showdist(); cout << endl;

 cout << "dist3 = "; dist3.showdist(); cout << endl;

 cout << "dist4 = "; dist4.showdist(); cout << endl;

 return 0;

 }

**Unary operators**

// countpp1.cpp

// increment counter variable with ++ operator

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Counter

 {

 private:

 unsigned int count; //count

 public:

 Counter() : count(0) //constructor

 { }

 unsigned int get\_count() //return count

 { return count; }

 void operator ++ () //increment (prefix)

 {

 ++count;

 }

 };

////////////////////////////////////////////////////////////////

int main()

 {

 Counter c1, c2; //define and initialize

 cout << "\nc1=" << c1.get\_count(); //display

 cout << "\nc2=" << c2.get\_count();

 ++c1; //increment c1

 ++c2; //increment c2

 ++c2; //increment c2

 cout << "\nc1=" << c1.get\_count(); //display again

 cout << "\nc2=" << c2.get\_count() << endl;

 return 0;

 }

// countpp2.cpp

// increment counter variable with ++ operator, return value

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Counter

 {

 private:

 unsigned int count; //count

 public:

 Counter() : count(0) //constructor

 { }

 unsigned int get\_count() //return count

 { return count; }

 Counter operator ++ () //increment count

 {

 ++count; //increment count

 Counter temp; //make a temporary Counter

 temp.count = count; //give it same value as this obj

 return temp; //return the copy

 }

 };

////////////////////////////////////////////////////////////////

int main()

 {

 Counter c1, c2; //c1=0, c2=0

 cout << "\nc1=" << c1.get\_count(); //display

 cout << "\nc2=" << c2.get\_count();

 ++c1; //c1=1

 c2 = ++c1; //c1=2, c2=2

 cout << "\nc1=" << c1.get\_count(); //display again

 cout << "\nc2=" << c2.get\_count() << endl;

 return 0;

 }

// countpp3.cpp

// increment counter variable with ++ operator

// uses unnamed temporary object

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Counter

 {

 private:

 unsigned int count; //count

 public:

 Counter() : count(0) //constructor no args

 { }

 Counter(int c) : count(c) //constructor, one arg

 { }

 int get\_count() //return count

 { return count; }

 Counter operator ++ () //increment count

 {

 ++count; //increment count, then return

 return Counter(count); // an unnamed temporary object

 } // initialized to this count

 };

////////////////////////////////////////////////////////////////

int main()

 {

 Counter c1, c2; //c1=0, c2=0

 cout << "\nc1=" << c1.get\_count(); //display

 cout << "\nc2=" << c2.get\_count();

 ++c1; //c1=1

 c2 = ++c1; //c1=2, c2=2

 cout << "\nc1=" << c1.get\_count(); //display again

 cout << "\nc2=" << c2.get\_count() << endl;

 return 0;

 }

**Posfix Notation**

// postfix.cpp

// overloaded ++ operator in both prefix and postfix

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Counter

 {

 private:

 unsigned int count; //count

 public:

 Counter() : count(0) //constructor no args

 { }

 Counter(int c) : count(c) //constructor, one arg

 { }

 unsigned int get\_count() const //return count

 { return count; }

 Counter operator ++ () //increment count (prefix)

 { //increment count, then return

 return Counter(++count); //an unnamed temporary object

 } //initialized to this count

 Counter operator ++ (int) //increment count (postfix)

 { //return an unnamed temporary

 return Counter(count++); //object initialized to this

 } //count, then increment count

 };

////////////////////////////////////////////////////////////////

int main()

 {

 Counter c1, c2; //c1=0, c2=0

 cout << "\nc1=" << c1.get\_count(); //display

 cout << "\nc2=" << c2.get\_count();

 ++c1; //c1=1

 c2 = ++c1; //c1=2, c2=2 (prefix)

 cout << "\nc1=" << c1.get\_count(); //display

 cout << "\nc2=" << c2.get\_count();

 c2 = c1++; //c1=3, c2=2 (postfix)

 cout << "\nc1=" << c1.get\_count(); //display again

 cout << "\nc2=" << c2.get\_count() << endl;

 return 0;

 }

**Comparison Operators**

// engless.cpp

// overloaded '<' operator compares two Distances

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public: //constructor (no args)

 Distance() : feet(0), inches(0.0)

 { } //constructor (two args)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void getdist() //get length from user

 {

 cout << "\nEnter feet: "; cin >> feet;

 cout << "Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 bool operator < (Distance) const; //compare distances

 };

//--------------------------------------------------------------

 //compare this distance with d2

bool Distance::operator < (Distance d2) const //return the sum

 {

 float bf1 = feet + inches/12;

 float bf2 = d2.feet + d2.inches/12;

 return (bf1 < bf2) ? true : false;

 }

////////////////////////////////////////////////////////////////

int main()

 {

 Distance dist1; //define Distance dist1

 dist1.getdist(); //get dist1 from user

 Distance dist2(6, 2.5); //define and initialize dist2

 //display distances

 cout << "\ndist1 = "; dist1.showdist();

 cout << "\ndist2 = "; dist2.showdist();

 if( dist1 < dist2 ) //overloaded '<' operator

 cout << "\ndist1 is less than dist2";

 else

 cout << "\ndist1 is greater than (or equal to) dist2";

 cout << endl;

 return 0;

 }

**Arithmetic Assignment operators**

// englpleq.cpp

// overloaded '+=' assignment operator

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Distance //English Distance class

 {

 private:

 int feet;

 float inches;

 public: //constructor (no args)

 Distance() : feet(0), inches(0.0)

 { } //constructor (two args)

 Distance(int ft, float in) : feet(ft), inches(in)

 { }

 void getdist() //get length from user

 {

 cout << "\nEnter feet: "; cin >> feet;

 cout << "Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 void operator += ( Distance );

 };

//--------------------------------------------------------------

 //add distance to this one

void Distance::operator += (Distance d2)

 {

 feet += d2.feet; //add the feet

 inches += d2.inches; //add the inches

 if(inches >= 12.0) //if total exceeds 12.0,

 { //then decrease inches

 inches -= 12.0; //by 12.0 and

 feet++; //increase feet

 } //by 1

 }

////////////////////////////////////////////////////////////////

int main()

 {

 Distance dist1; //define dist1

 dist1.getdist(); //get dist1 from user

 cout << "\ndist1 = "; dist1.showdist();

 Distance dist2(11, 6.25); //define, initialize dist2

 cout << "\ndist2 = "; dist2.showdist();

 dist1 += dist2; //dist1 = dist1 + dist2

 cout << "\nAfter addition,";

 cout << "\ndist1 = "; dist1.showdist();

 cout << endl;

 return 0;

 }

**Data Conversion**

Type conversion

|  |  |  |
| --- | --- | --- |
| **Conversion** | **Routine in Destination** | **Routine in source** |
| Basic to basic(float to int) | Built in | Built in |
| Basic to class(int to obj) | Constructor |   |
| Class to Basic(obj to int) |   | Operator function |
| Class to class(obj to otherObj | Constructor | Operator function |

**Conversion between Class and Basic Types**

// englconv.cpp

// conversions: Distance to meters, meters to Distance

#include <iostream>

using namespace std;

////////////////////////////////////////////////////////////////

class Distance //English Distance class

 {

 private:

 const float MTF; //meters to feet

 int feet;

 float inches;

 public: //constructor (no args)

 Distance() : feet(0), inches(0.0), MTF(3.280833F)

 { } //constructor (one arg)

 Distance(float meters) : MTF(3.280833F)

 { //convert meters to Distance

 float fltfeet = MTF \* meters; //convert to float feet

 feet = int(fltfeet); //feet is integer part

 inches = 12\*(fltfeet-feet); //inches is what's left

 } //constructor (two args)

 Distance(int ft, float in) : feet(ft),

 inches(in), MTF(3.280833F)

 { }

 void getdist() //get length from user

 {

 cout << "\nEnter feet: "; cin >> feet;

 cout << "Enter inches: "; cin >> inches;

 }

 void showdist() const //display distance

 { cout << feet << "\'-" << inches << '\"'; }

 operator float() const //conversion operator

 { //converts Distance to meters

 float fracfeet = inches/12; //convert the inches

 fracfeet += static\_cast<float>(feet); //add the feet

 return fracfeet/MTF; //convert to meters

 }

 };

////////////////////////////////////////////////////////////////

int main()

 {

 float mtrs;

 Distance dist1 = 2.35F; //uses 1-arg constructor to

 //convert meters to Distance

 cout << "\ndist1 = "; dist1.showdist();

 mtrs = static\_cast<float>(dist1); //uses conversion operator

 //for Distance to meters

 cout << "\ndist1 = " << mtrs << " meters\n";

 Distance dist2(5, 10.25); //uses 2-arg constructor

 mtrs = dist2; //also uses conversion op

 cout << "\ndist2 = " << mtrs << " meters\n";

// dist2 = mtrs; //error, = won't convert

 return 0;

 }

**Conversion between Objects of Different Classes**

**class Cartesian**

{

double x;

double y;

public:

Cartesian()

{x=0,y=0)

Cartesian(doubly x, double y)

{

this.x=x

this.y=y

}

//added constructor

Cartesian(Polar p)

{

double r=P.getRadius();

double a=p.getAngle();

x=r\*cos(a)

y=r\*cos(a)

}

};

**class Polar**

{

double radius;

double angle;

public:

Polar()

{

radius=0;

angle=0;

}

Polar (double r, double a)

{

radius=r;

angle=a;

}

operator Cartesian()

{

double x=Radius\*cos(angle);

double y=radius\*sin(angle);

return cartesian(x,y)

}

};

**In main**

Polar P(10,.5)

Cartesian c;

c=p

**Pointer**

* 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

 }

**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** |
| 1 | **strcpy(s1, s2);** Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);** Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);** Returns the length of string s1. |
| 4 | **strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| 5 | **strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1. |
| 6 | **strstr(s1, s2);** 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++);

}