University of Gaziantep
Department of Engineering Physics

Advantages of C++ over C (for Teaching)

By

Dr. Ahmet BİNGÜL

May 2009

Introduction

In these notes, it is attempted

- to introduce advantages of C++ over C w.r.t teaching programming courses
- to show some features of C++ Programming Language

Note:

C and C++ are quite different from each other, even though they share some common syntax.

Resources

Web resources

http://www.gantep.edu.tr/~bingul/c (Turkish)
http://www.cplusplus.com

Compilers

GCC	(for Linux & Windows)
Dev-C++	(for Windows)
Borland C++	(for Windows)
Visual C++	(for Windows)

General Observations

	C	C++
File Extensions	`.c .C`	`.cc .cpp .cxx`
Comment Operator	`/* this is a comment */`	`/* this is a comment */` `// this is a comment`
Language Type	**Structured Programming Language**	**Object Oriented Programming Language**

“Hello World” Examples

```
/* hello.c */
#include <stdio.h>

int main(){
 printf("Hello world\n");
return 0;
}
```

Compile/run with gcc

```
$ gcc hello.c -o hello
$ ./hello
Hello world
```

```
// hello.cc
#include <iostream>
using namespace std;

int main(){
 cout << "Hello world" << endl;
return 0;
}
```

Compile/run with gcc

```
$ c++ hello.cc -o hello
$ ./hello
Hello world
```

Note:

Any C program can be compiled by a C++ compiler.

Header Files

In C

```
#include <stdio.h>
#include <math.h>
#include "mine.h"
```

In C++

obsolete usage !

```
#include <iostream>        <iostream.h>
#include <cstdio>          <cstdio.h> or <stdio.h>
#include <cmath>           <cmath.h> or <math.h>
#include "mine.h"
```

Basic Input/Output

Standard Output

```
printf("Hello World");
printf("i = %d", i);
printf("%f",(a+b)/2);
```

```
cout << "Hello World";
cout << "i = " << i;
cout << (a+b)/2;
```

Like **`printf`**, **`cout`** does not add a line break.
This is done by inserting a **`'\n'`** or a using a **`endl`** manipulator.

```
printf("Hello World\n");
```

```
cout << "Hello World" << endl;
```

```
cout << "Hello World" << '\n';
```

Basic Input/Output

Standard Input

Handling the standard input in C++ is done by applying the overloaded operator of extraction (**>>**) on the **cin** stream.

```
scanf("%c", &c);

scanf("%d %f",&i, &r);

scanf("%s", str);
```

```
cin >> c;

cin >> i >> r;

cin >> str;
```

cout and cin are defined in <iostream> in namespace std

Namespaces

Namespaces allow to group entities like classes,objects and functions under a name.

Format:

```
namespace identifier
{
    entities
}
```

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


namespace myNamespace
{
  double V = 12.0;
  int R = 10;
}


main(){
 using namesapce myNamespace;
 double i = V/R;
 cout << i << endl;
}
```

Note that:

All the files in the C++ standard library declare all of its entities within the **std** namespace defined in **<iostream>** like **cin** and **cout** streams.

Reserved Keywords

- **Reserved Keywords in C** (you can't use as an identifier)
  ```
  auto, double, int, struct, break, else, long, switch,
  case, enum, register  typedef, char, extern, return,
  union, const, float, short,  unsigned, continue   for,
  signed, void, default, goto, sizeof, volatile, do, if,
  static, while
  ```

- **Reserved Keywords in C++**
  ```
  asm, auto, bool, break, case, catch, char, class, const,
  const_cast, continue, default, delete, do, double,
  dynamic_cast, else, enum, explicit, export, extern,
  false, float, for, friend, goto, if, inline, int, long,
  mutable, namespace, new, operator, private, protected,
  public, register, reinterpret_cast, return, short,
  signed, sizeof, static, static_cast, struct, switch,
  template, this, throw, true, try, typedef, typeid,
  typename, union, unsigned, using, virtual, void,
  volatile, wchar_t, while
  ```

Fundamental Data Types

C / C++	C++

`char`
`short int`
`int`
`long int`
	`bool`
	`string`

`float`
`double`
`long double`	

Fundamental Data Types

Size in byte of data types for different platforms:

Data type	Windows 32 bit	Linux 32 bit	Linux 64 bit
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	4	8
`float`	4	4	4
`double`	8	8	8
`long double`	10	12	16

Scope of Variables

In C++, you can declare variables anywhere you want

```c
/* Globals */
int x = 10;
float f = 1.0;

main()
{
   int i, n = x;
   double y;

   for(i=2; i<n; i++)
   {
     f *= i;
     y = log(f);
   }

   printf("%f %lf\n",f, y);
}
```

```cpp
// Globals
int x = 10;
float f = 1.0;

main()
{
  int n = x;
  for(int i=2; i<n; i++)
  {
     f *= i;
     double y = log(f);
  }
  cout << f << " " << y << endl;
}
```

Scope of Variables

The scope of local variables is limited to the block enclosed in braces ({ }) where they are declared.

```
int x = 11; // this is global

int main()
{
   int x = 22;
   cout << "In main: x = " << x << endl;

   {
     int x = 33;
     cout << "In block inside main: x = " << x << endl;
   }

   // access to the global x
   cout << "In main: ::x = " << ::x << endl;

  return 0;
}
```

```
In main: x = 22
In block inside main: x = 33
In main: ::x = 11
```

Basic Strings

There are three ways to define a string:

```
char   *str1    = "This is string1"; // in C
char    str2[] = "This is string2"; // in C
string str3    = "This is string3"; // in C++
```

```
#include <stdio.h>
#include <string.h>


main()
{
 char s[14];
 strcpy(s,"This is first");
 puts(s);

 strcpy(s, "This is second");
 puts(s);
}
```

```
#include <iostream>
#include <string>
using namespace std;

main()
{
   string s;
   s = "This is first";
   cout << s << endl;

   s = "This is second";
   cout << s << endl;
}
```

```
This is first
This is second
```

Initialization of Variables

There are two ways:

- using an equal sign:

```
int a = 0;
float f = 1.0;
string str = "a string content";
```

- using a constructor initialization

```
int a(0);
float f(1.0);
string str("a string content");
```

Operators

Assignment (=)

Following assignments are valid in C++:

```
a = 5;
a = b;
a = 2 + (b = 5); // equivalent to: b = 5 and a = 7
x = y = z = 5;   // equivalent to: x = 5, y = 5 and z = 5
```

Operators

Explicit Type Casting Operator

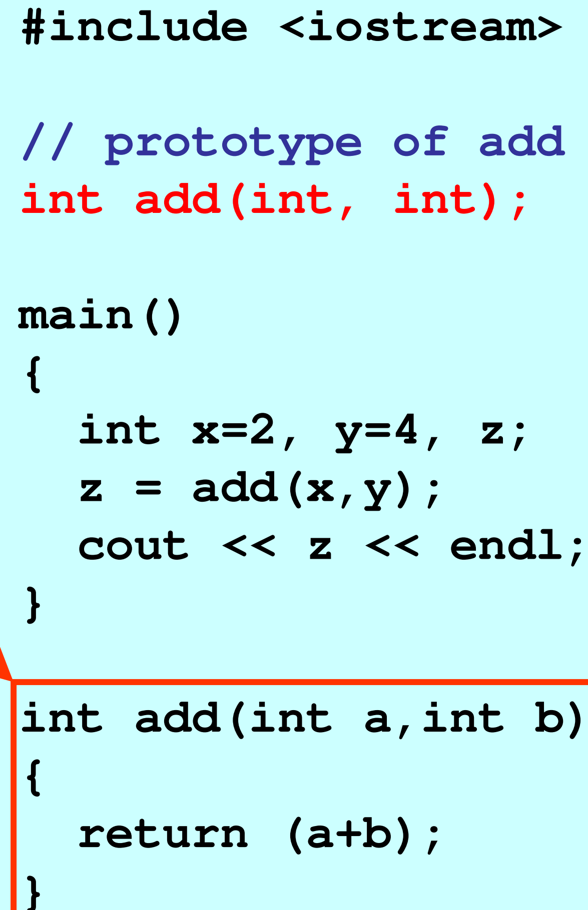
This allows you to convert a data of a given type to another.

```
int i,j;
float f;

i = 3;

f = (float) i;  // in C

f = float(i);   // in C++

j = int(4.8);   // in C++
```

Functions

The use of functions in C and C++ is the same.

```
#include <iostream>

int add(int a,int b)
{
  return (a+b);
}

main()
{
  int x=2, y=4, z;
  z = add(x,y);
  cout << z << endl;
}
```



```
#include <iostream>

// prototype of add
int add(int, int);

main()
{
  int x=2, y=4, z;
  z = add(x,y);
  cout << z << endl;
}

int add(int a,int b)
{
  return (a+b);
}
```

Functions

Overloading Functions: This is allowed by C++

```
#include <iostream>

int max(int x, int y){
   return (x>y ? x:y);
}

int max(int x, int y, int z){
  int m = (x>y ? x:y);
  return  (z>m ? z:m);
}

double max(double x, double y){
 return (x>y ? x:y);
}

main(){
  cout <<"max(9,7)     = " << max(9,7)     << endl;
  cout <<"max(3,6,2)   = " << max(3,6,2)   << endl;
  cout <<"max(3.1,4.7)= " << max(3.1,4.7) << endl;
}
```

```
max(9,7)     =  9
max(3,6,2)   =  6
max(3.1,4.7)=  4.7
```

Functions

Variable number of arguments (Default arguments)

C/C++ allows a function to have a variable number of arguments.

Consider the implementation of the second order polynomial function:

$$P(x) = a + bx + cx^2$$

```
double p(double x, ...){
  double a, t = 0.0;
  int i;
  va_list  ag;

   va_start(ag, 2);  /* allocate memory */


   for(i=0; i<2; i++)
     t += va_arg(ag, double)*pow(x,i);

   va_end(ag);      /* free the memory */

  return t;
}
```

Functions

The use of default arguments is more simple in C++

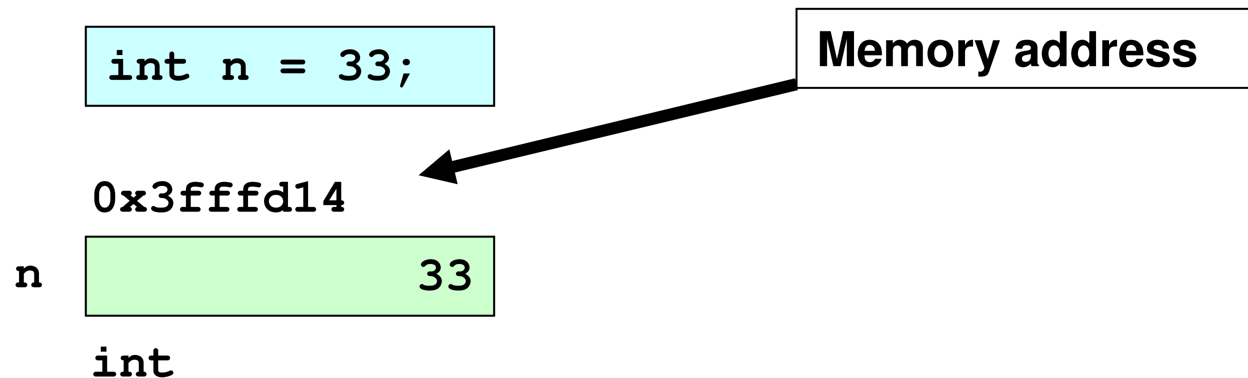
```
// -- optional parameters must all be listed last
double p(double, double=0, double =0, double =0);

main()
{
  double x = 1.0;

  cout << "p(x,7)     = " << p(x,7)     << endl;
  cout << "p(x,7,6)   = " << p(x,7,6)   << endl;
  cout << "p(x,7,6,3)= " << p(x,7,6,3) << endl;
}

double p(double x, double a, double b, double c){
  return a + b*x + c*x*x;
}
```

```
p(x,7)     =  7.
p(x,7,6)   =  13.
p(x,7,6,3)=  16.
```

Pointers and References

When a variable is declared and assigned to a value four fundamental attributes associated with it:

- its *name*
- its *type*
- its *value* (content)
- its *address*



Pointers and References

Address Operator

* The **value** of a variable is accessed via its ***name***.

* The **address** of a variable is accessed via the ***address operator*** **`&`**.

```
#include <iostream>
// printing both the value and address
// valid for both C and C++

main()
{
  int n = 33;
  cout << " n = " <<  n << endl;
  cout << "&n = " << &n << endl;
}
```

```
 n = 33
&n = 0xbfdd8ad4
```

Pointers and References

Pointer

The address operator returns the memory adress of a variable.

We can store the address in another variable, called ***pointer***.

```
main()
{
   int  n = 33;
   int* p = &n; // p holds the address of n

   cout << " n = " <<  n << endl;
   cout << "&n = " << &n << endl;

   cout << " p = " <<  p << endl;
   cout << "&p = " << &p << endl;
}
```

```
 n = 33
&n = 0xbfdd8ad4
 p = 0xbfdd8ad4
&p = 0xbffafad0
```

	address	value	type
n	0xbfdd8ad4	33	int
p	0xbfdd8ad0	0xbfdd8ad4	int*

Pointers and References

Reference

* The **reference** is an ***alias,*** a ***synonym*** for a variable.
* It is declerated by using the ***address operator*** **&**.

```
main(){
   int  n = 33;
   int& r = n; // r is a reference for n

   cout << n << r << endl;
   --n;
   cout << n << r << endl;
   r *= 2;
   cout << n << r << endl;

   cout << &n << &r << endl;
}
```

```
33 33
32 32
64 64
0xbfdd8ad4 0xbfdd8ad4
```

0xbfdd8ad4

n,r | 33

int

Pointers and References

Arguments passed by value and by reference

```
// arg. Pass by value
void Decrease(int a, int b){
   a--;
   b--;
}
```

```
// arg. Pass by address
void Decrease(int *a, int *b){
   (*a)--;
   (*b)--;
}
```

```
// arg. Pass by reference
void Decrease(int& a, int& b){
   a--;
   b--;
}
```

Pointers and References

A function may return more than ONE value using references:

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

void Convert(float, int& , float&);

main(){
  float rx, x = 3.2;
  int   ix;

  Convert(x, ix, rx);
  cout << " x = " << x  << endl;
  cout << " ix= " << ix << endl;
  cout << " rx= " << rx << endl;
}

void Convert(float num, int& ip, float& rp)
{
  ip = num;
  rp = num - int(num);
}
```

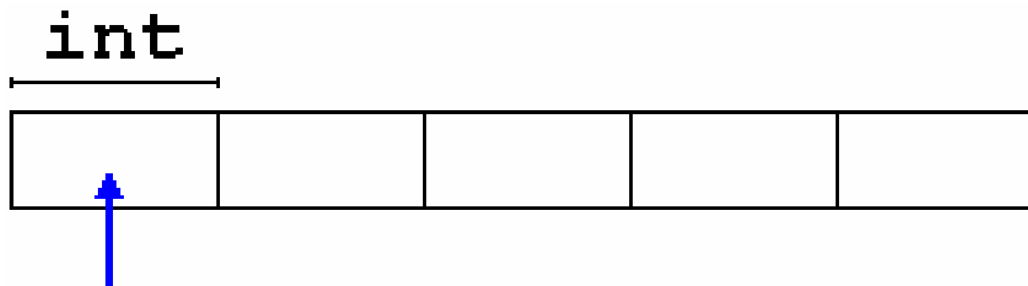
```
x  = 3.2
ix = 3
rx = 0.2
```

Dynamic Memory

ANSI C uses following functions defined in **<stdlib.h>**

malloc(), **calloc()**, **realloc()** and **free()**

```
/* decleration */
int *a;

/* allocate the memory */
a = (int *) malloc(sizeof(int)*5);

/* ... use array here ... */

/* free the memory */
free(a);
```



Dynamic Memory

In C++, it easier than C

- In order to request dynamic memory we use the operator **`new`**.
- **`delete`** operator reverses the action of the **`new`** operator, that is it frees the allocated memory by the **`new`** operator.

```
// declearation
int *a;


// allocate the memory
a = new int [5];

// ... use the array here ...


// free the memory
delete [] a;
```

Dynamic Memory

```c
/* mean of n numbers */
main (){
  float *x, mean, s;
  int i, n;

  printf("How many elements: ");
  scanf("%d",&n);

  if(n>0)
    x = (float *)
        malloc(sizeof(float)*n);

  puts ("Input elements: ");
  for(i=0, s=0.0; i<n; i++){
      scanf("%f", &x[i]);
      s += x[i];
  }
  mean = s/n;
  printf("Mean = ",mean);
  if(n>0) free(x);
}
```

```cpp
// mean of n numbers
main (){
  float *x, mean, s;
  int i,n;

  cout << "How many elements: ";
  cin >> n;

  if(n>0)
     x = new float[n];


  cout << "Input elements: ";
  for(i=0, s=0.0; i<n; i++){
     cin >> x[i];
     s += x[i];
  }
  mean = s/n;
  cout << "Mean = " << mean;
  if(n>0) delete [] x;
}
```

Vector (Linked lists)

```
// see: http://www.cplusplus.com/reference/stl/vector/
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<short> v;
   v.push_back( 23 );
   v.push_back( -1 );
   v.push_back( 9999 );
   v.push_back( 0 );
   v.push_back( 4 );

   cout << "Before sorting: ";
   for( unsigned int i = 0; i < v.size(); i++ ) cout << v[i];
   cout << endl;

   sort( v.begin(), v.end() );

   cout << "After sorting: ";
   for( unsigned int i = 0; i < v.size(); i++ ) cout << v[i];
   cout << endl;


return 0;
}
```

Data Structures

```
struct Student
{
  int mt1, mt2, fin;
};


main()
{
   Student s1, s2, *p;

   p = &s2;  // p points to s2

   s1.fin   // member fin of object s1
   p->fin   // member fin of object pointed by p
}
```

Classes

- A ***class*** is an expanded concept of a data structure: instead of holding only data, **it can hold both data and functions**.

- Classes are declared by using `class` keyword.

```
class class_name
{
    access_specifier_1:
      member1;
    access_specifier_2:
      member2;
    ...
} object_names;
```

Classes

An access specifier is one of the followings:

- **`private`**
  members of a class are accessible only from within other members of the same class

- **`public`**
  members are accessible from anywhere where the object is visible

- **`protected`**
  members are accessible from members of their same class but also from members of their derived classes

By default, all members of a class declared with the **`class`** keyword have **`private`** access for all its members.

Classes

An example class:

```
class RCCircuit{
        double R, C, V;
    public:
        double tau;
        void set_values(double, double);
        double I(double);
}my_rc;
```

- declares a class (i.e. a type) called **`RCCircuit`** and an object (i.e. a variable) of this class called **`my_rc`**.
- The functions: **`set_values()`** and **`I()`** are called *member functions* or *methods*.

```
class RCCircuit{
      double R, C, V;
   public:
      double tau;
      void set_values(double, double);
      double I(double);
};

main(){

       RCCircuit x;
       x.set_values(10., 32.);

       for(double t=0.0; t<x.tau; t += 0.1)
           cout << x.I(t) << endl;

}

// Member functions ------------------------------------
void RCCircuit::set_values(double res, double cap){
  V = 24.0;          // volt
  R = res * 1.0e+3; // kiloOhm
  C = cap * 1.0e-6; // microFarad
  tau = R*C;
}
double RCCircuit::I(double t){
       return V*exp(-t/tau)/R;
}
```

Classes

Self Contained Implementation

```
class RCCircuit{
      double R, C, V;
   public:
      double tau;
      void set_values(double res, double cap){
        V = 24.0;          // volt
        R = res * 1.0e+3; // kiloOhm
        C = cap * 1.0e-6; // microFarad
        tau = R*C;
      }
      double I(double t){ return V*exp(-t/tau)/R; };
};

main(){

       RCCircuit x;
       x.set_values(10., 32.);

       for(double t=0.0; t<x.tau; t += 0.1)
           cout << x.I(t) << endl;
}
```

Classes

Constructors

```
class RCCircuit{
      double R, C, V;
   public:
      double tau;
      RCCircuit(double res, double cap){
        V = 24.0;          // volt
        R = res * 1.0e+3; // kiloOhm
        C = cap * 1.0e-6; // microFarad
        tau = R*C;
      }
      double I(double t){ return V*exp(-t/tau)/R; };
};

main(){

       RCCircuit x(10.0, 32.0);
       for(double t=0.0; t<x.tau; t += 0.1)
           cout << x.I(t) << endl;
}
```

File Management

C++ provides the following classes to perform output and input of characters to/from files:

- **`ofstream`**: Stream class to write on files
- **`ifstream`**: Stream class to read from files
- **`fstream`**: Stream class to both read and write from/to files.

You only need to include the standard header **`<fstream>`** to your c++ code.

Files Management

Open modes	
C	C++
`FILE *fopen(*file, *mode);`	`object.open(file,mode);`

`const char *mode`
`"r", "w", "b", "a"`
	`mode (optional)`
	`ios::out, ios::in`
	`ios::binary, ios::app`

```
#include <stdio.h>
...
FILE *f;

f = fopen("data.txt", "w");

fprintf(f,"Line to the file");

fclose(f);
...
```

```
#include <fstream>
...
ofstream f;

f.open("data.txt", ios::out);

f << "Line to the file";

f.close();
...
```

Topics Not Covered

- **Classes**
  - Overloading operators
  - Friendship and Inheritance
  - Polymorphism
- **Exceptions**
- **Templates**
- **Advanced Type Casting**
- **Advanced String Operations**

Conclusions

Basic level C++ has following advantages over C

- **Strings**
  Use of strings are very simple
- **Function**
  - default argument functions are more clear
  - argument can be passed by reference
- **Dynamic Memory Management**
  It is done by two statements: **`new`** and **`delete`**.
- **File Managament**
  C++ does not require a file pointer
- **Namespaces**
  provide modular programming
- **Classes & Object Oriented Programming**
  provide more flexible programming w.r.t. data structures

END OF SEMINAR

BACKUP SLIDES

Classes

Overloading Operators

C++ incorporates the option to use standard operators to perform operations with classes in addition to with fundamental types. For example we can perform the simple operation:

```
int a, b=22, c=44;
a = b + c;
```

However following operation is not valid:

```
class Product{
  int weight;
  float price;
} a, b, c;
a = b + c;
```

We can design classes able to perform operations using standard operators. Thanks to C++ ☺

Classes

```
#include <iostream.h>

class Vector {
  public:
    int x,y;
    Vector () {x=0; y=0;} // default constructor
    Vector (int a,int b){x=a; y=b;}
    Vector operator + (Vector);
};

Vector Vector::operator+ (Vector param) {
  Vector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}

main () {
  Vector a (3,1);
  Vector b (1,2);
  Vector c;
  c = a + b;
  cout << "c= (" << c.x << "," << c.y << ")";
}
```

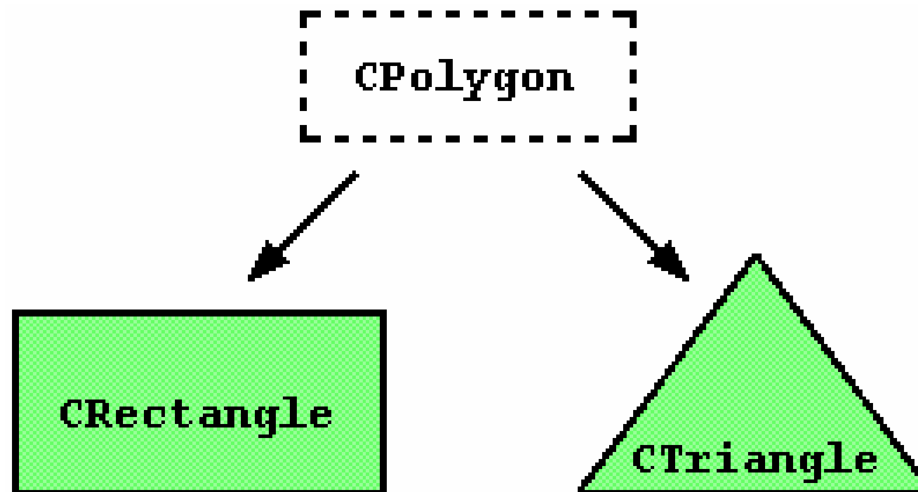
```
c = (4,3)
```

Classes

Inheritance Between Classes

Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own.

Suppose that we want to declare a series of classes which have certain common properties.



Classes

```
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b){
       width=a;
       height=b;
    }
};

class CRectangle: public CPolygon {
  public:
    int area (){
       return (width * height);
    }
};

class CTriangle: public CPolygon{
  public:
  int area (){
      return (width * height / 2);
  }
};
```

```
main()
{
  CRectangle rect;
  CTriangle trgl;

  rect.set_values (4,5);
  trgl.set_values (4,5);

  cout << rect.area() << endl;
  cout << trgl.area() << endl;
}
```

```
20
10
```

Classes

Polymorphism

C++ allows objects of different types to respond differently to the same function call.

This is called ***polymorphism*** and
it is achived by means of virtual functions.

Classes

```
#include <iostream.h>
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b){
       width=a; height=b;
    }
    virtual int area(){
       return (0);
    }
};

class CRectangle: public CPolygon {
  public:
    int area (){
       return (width * height);
    }
};
class CTriangle: public CPolygon{
  public:
  int area (){
      return (width * height / 2);
  }
};
```

```
main()
{
CRectangle rect;
CTriangle trgl;
CPolygon poly;
CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
CPolygon * ppoly3 = &poly;

ppoly1->set_values(4,5);
ppoly2->set_values(4,5);
ppoly3->set_values(4,5);

cout << ppoly1->area() <<'\n';
cout << ppoly2->area() <<'\n';
cout << ppoly3->area() <<'\n';
}
```

```
20
10
0
```

Linked Lists in Fortran and C/C++

Pointers in classes (derived data types) may even point to the class (derived data type) being defined.

This feature is useful, since it permits construction of various types of dynamic structures linked together by successive pointers during the execution of a program.

The simplest such structure is a ***linked list***, which is a list of values linked together by pointers.

Following derived data type contains a real number and a pointer:

```
TYPE Node
 INTEGER :: data
 TYPE(Node),POINTER :: next
END TYPE Node
```

```
class Node{
 public:
  int data;
  Node *next;
};
```

Linked Lists in Fortran and C/C++

The following programs (given next page) allow the user to create a linked list in reverse.It traverses the list printing each data value.

An example output:

```
Enter a list of numbers:
22
66
77
99
-8
Reverse order list:
99
77
66
22
```

```fortran
PROGRAM Linked_List

  TYPE Node
    INTEGER :: Data
    TYPE (Node), POINTER :: Next
  END TYPE Node

  INTEGER :: Num, N=0
  TYPE (Node), POINTER :: P, Q
  NULLIFY(P)

  PRINT *,"Input a list of numbers:"

  DO
    READ *, Num
    IF ( Num < 0 ) EXIT
    N=N+1
    ALLOCATE(Q)
    Q%Data = Num
    Q%Next => P
    P => Q
  END DO
  Q => P
  PRINT *, "Reversee order list: "
  DO
    IF ( .NOT.ASSOCIATED(Q) ) EXIT
    PRINT *, Q%Data
    Q => Q%Next
  END DO
END PROGRAM
```

```cpp
#include <iostream>

class Node{
  public:
    int data;
    Node *next;
};

main(){
  int n=0,num;
  Node *q, *p = NULL;

  cout << "Input a list of numbers"<<endl;

  while(1){
      cin >> num;
      if(num<0) break;
      n++;
      q = new Node;
      q->data = num;
      q->next = p;
      p = q;
  }
  q = p;
  cout << "Reverse order list: ";
  while(1){
    if(q==NULL) break;
    cout << q->data << ", ";
    q = q->next;
  }
}
```