University of Gaziantep
Department of Engineering Physics

A C++ tutorial for Fortran 95 Users

By

Dr. Ahmet BİNGÜL

June 2007

Introduction

In these notes, we will attempt to list and introduce some programming features of C++ Programming Language for Fortran 90/95 users.

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

Resources

Web resources:

http://www.fortran.gantep.edu.tr/
http://www.cplusplus.com/

Books:

An Introduction to Fortran 95
Kanber, Beddall (2006) Gazi Kitapevi
Programming with C++
Hubbard (1996) McGraw Hill – Shaum's Outlines

General Observations

	C/C++	Fortran 90/95
Case sensitivity	Case sensitive **`result`** and **`Result`** are different identifiers	Case insensitive **`result`** and **`Result`** are the same identifiers
Each line of the code	must end with a semicolon (**`;`**)	may end with/without a semicolon (**`;`**)
File extensions:	`.c` `.cpp` `.c++`	`.f` `.f90` `.f95`
Comment operators:	`// this is a comment` `/* this is a comment */`	`! this is a comment`
Compilers	`gcc or g++` `DevC++, Borland C++`	`g95 , ifc` `Microsoft VF, Salford`

"Hello World" Examples

```
! hello.f95
PROGRAM MyFirstProgram

 PRINT *, "Hello World."

END PROGRAM
```

Compile and run with g95

```
$ g95 hello.f95 -o hello
$ ./hello
Hello World.
$
```

```
// hello.c
#include <iostream.h>

main(){
   cout << "Hello world."
}
```

Compile and run with gcc

```
$ g++ hello.c -o hello
$ ./hello
Hello World.
$
```

Identifiers

- Both in Fortran and C++
  a valid identifier is a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier.

- Reserved Keywords in C++ that you can't use as an identifier
  `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`

- In Fortran you can use any of the keywords such as

```
INTEGER :: Integer
```

Fundamental Data Types

Fortran	C/C++	Size (byte)	Range (signed)

`INTEGER K=1`
`INTEGER K=2`
`INTEGER K=4`
`INTEGER K=4`
	`char`
	`short int`
	`int`
	`long int`
		1
		2
		4
		4
			-128,127
			-32768,32767
			-2147483648,2147483647
			-2147483648,2147483647

`REAL K=4`
`REAL K=8`
`REAL K=16`
	`float`
	`double`
	`long double`
		4
		8
		8
			$3.4\times10^{\pm 38}$ (7 digits)
			$1.7\times10^{\pm 308}$ (15 digits)
			$1.7\times10^{\pm 308}$ (15 digits)

`LOGICAL`	`bool`	1	true or false
`CHARACTER`	`string`	-	-
`COMPLEX`	-	4	-

Integer Ranges

```
#include <iostream.h>
#include <limits.h>
// Prints the constants strored in limits.h
void main(void)
{
  cout << "minimum char           = " <<  CHAR_MIN  << endl;
  cout << "maximum char           = " <<  CHAR_MAX  << endl;
  cout << "minimum short          = " <<  SHRT_MIN  << endl;
  cout << "maximum short          = " <<  SHRT_MAX  << endl;
  cout << "minimum int            = " <<  INT_MIN   << endl;
  cout << "maximum int            = " <<  INT_MAX   << endl;
  cout << "minimum long           = " <<  LONG_MIN  << endl;
  cout << "maximum long           = " <<  LONG_MAX  << endl;
  cout << '\n';
  cout << "minimum signed char    = " <<  SCHAR_MIN << endl;
  cout << "maximum signed char    = " <<  SCHAR_MAX << endl;
  cout << "maximum unsigned char  = " <<  UCHAR_MAX << endl;
  cout << "maximum unsigned short = " <<  USHRT_MAX << endl;
  cout << "maximum unsigned int   = " <<  UINT_MAX  << endl;
  cout << "maximum unsigned long  = " <<  ULONG_MAX << endl;
}
```

Decleration of Variables

In order to use a *variable* in Fortran and C++,

we must first **declare** it specifying its data type .

```
INTEGER :: K,L
REAL :: Speed
```

```
int k,l;
float speed;
```

C++ prefixes for the data types

```
signed int i;       // i.e. int i;
unsigned int u;     // change range 0 to 4294967295
unsigned u;         // i.e. unsigned int u;
short s;            // i.e. short int s;
long l;             // long int l;
```

Scope of Variables

A variable can be either of global or local scope.

```
PROGRAM Scope
IMPLICIT NONE
 INTEGER :: Age
 REAL :: Real
 CHARACTER (5) :: Str

 PRINT *,"Enter your age:"
 READ *, Age
   ...

CONTAINS

   INTEGER FUNCTION Foo(x)
   REAL , INTENT(IN) :: x
   INTEGER :: K = 5
   REAL :: R

    Foo = x + K/R

   END FUNCTION

 END PROGRAM
```

Global variables — Instructions — Local variables

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

int Integer;
char aCharacter;
char string [20];
unsigned int NumberOfSons;

int main ()
{
   unsigned short Age;
   float ANumber, AnotherOne;

   cout << "Enter your age:"
   cin >> Age;
   ...
}
```

Global variables — Local variables — Instructions

A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.

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

Scope of Variables – Example

```
#include <iostream.h>
// program to demonstrate the variable scopes
int x = 11; // this x is global
main()
{
   int x = 22;
   cout << "In main: x = " << x << endl;
   {
      int x = 33;
      cout << "In block inside main: x = " << x << endl;
   }
   /* access to the gloabal x */
   cout << "In main: ::x = " << ::x << endl;
}
```

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

Introduction to Strings

There are three ways to define a string:

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

```
PROGRAM String_Example
CHARACTER (LEN=20) :: MyString

 MyString = "This is a string"

 PRINT *, MyString

END PROGRAM String_Example
```

```
This is a string
```

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

int main (){
  string mystring;

  mystring = "This is a string";
  cout << mystring << endl;
}
```

```
This is a string
```

Initialization of Variables

There are two ways to do this in C++:

- 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");
```

Constants

Literals

Literals are used to express particular values within the source code.

```
j = 25; // here 25 is a literal constant
```

Integer Numerals

Valid integer literals

```
0
1299
-542
```

octal and hexadesimal notation:

```
75      // decimal
0113    // octal
0x4b    // hexadecimal
```

By default each integer literals are of type `int`. We can force them to `unsigned` and/or `long`:

```
75    // int
75u   // unsigned int
75l   // long int
75ul  // unsigned long
```

Constants

Floating Point (`REAL`) Numbers

Valid floating point literals

```
3.14159  // 3.14159
6.02e23  // 6.02 x 10^23
1.6e-19  // 1.6 x 10^-19
-3.      // -3.0
```

By default each real literals are of type **`double`**.

We can force them to **`float`** and/or **`long double`**:

```
3.14159f   // float
3.14159l   // long double
```

Note that:

Any of the letters in a numerical literals

`u, l, e, f`

can be replaced with its uppercase letters

`U, L, E, F`

without any difference in their meanings.

Constants

Character and string literals

There also exist non-numerical constants, like:

```
'a'           // a character literal
"Hello World" // a string literal
```

Character and string literals have certain peculiarities, like the escape codes →

For example:

```
'\n'
"Left \t Right"
```

String literals can extend to more than a single line

```
"string expressed in \
two lines"
```

`\n`	Newline
`\r`	carriage return
`\t`	Tab
`\v`	Vertical tab
`\b`	Backspace
`\f`	Form feed (page feed)
`\a`	Alert (beep)
`\'`	Single quote
`\"`	Double quote
`\?`	Question mark
`\\`	Backslash

Constants

Boolean (LOGICAL) Literals

```fortran
PROGRAM Boolean
LOGICAL :: B1 = .TRUE.
LOGICAL :: B2 = .FALSE.

 PRINT *,"B1 = ",B1
 PRINT *,"B2 = ",B2

END PROGRAM Boolean
```

```
 B1 =   T
 B2 =   F
```

```cpp
#include <iostream.h>

main()
{
  bool b1 = true;
  bool b2 = false;

  cout << "b1 = " << b1 << endl;
  cout << "b2 = " << b2 << endl;
}
```

```
 b1 =  1
 b2 =  0
```

Constants

Defined Constants

You can define your own names for constants without having to resort to memory-consuming variables, simply by using the **#define** preprocessors directive.

```
#include <iostream>

#define PI 3.14159
#define NEWLINE '\n'

main(){

 double r=5.0;   // radius
 double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;
}
```

Declared Constants

```
REAL, PARAMETER :: c = 3.0E8
INTEGER, PARAMETER :: Max = 100
```

```
 const float c = 3.0e8;
 const int max = 100;
```

Operators

Simple Arithmetic Operations

Operation	*Fortran*	*Example*	*C/C++*	*Example*
addtion	`+`	`X = 12+5`	`+`	`x = 12+5`
subtraction	`-`	`X = 12-5`	`-`	`x = 12-5`
multiplication	`*`	`X = 12*5`	`*`	`x = 12*5`
division	`/`	`X = 12/5`	`/`	`x = 12/5`
power	`**`	`X = 12**5`	`pow`	`x = pow(12,5)`
modulus	`MOD`	`X =MOD(12,5)`	`%`	`x = 12%5`

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
```

Compound Assignment (+=, -=, *=, /=, ...)

```
a += 5;           // equivalent to: a = a + 5;
f *= i;           // equivalent to: f = f * i;
f *= i+1;         // equivalent to: f = f * (i+1);
z /= 1 + x;       // equivalent to: z = z / (1+x);
```

Operators

Increase or decrease by 1 (++, −−)

Following assignments are equivalent:

```
i++;
++i;
i += 1;
i = i + 1;
```

Be careful when using these operators:

```
a = 5;     // a = 5
b = a++;   // b = 5 and a = 6
```

```
a = 5;     // a = 5
b = ++a;   // b = 6 and a = 6
```

Operators

Relational and Logical Operations

Operation	*Fortran*	*Example*	*C/C++*	*Example*
Greater than	`>`	`X > Y`	`>`	`x > y`
greater than or equal to	`>=`	`X >= Y`	`>=`	`x >= y`
Less than	`<`	`X < Y`	`<`	`x < y`
Less than or equal to	`<=`	`X <= Y`	`<=`	`x <= y`
Equal to	`==`	`X == Y`	`==`	`x == y`
Not equal to	`/=`	`X /= Y`	`!=`	`x != y`
Logical or	`.OR.`	`X>1 .OR. Y<=9`	`||`	`x>1 || y<=9`
Logical and	`.AND.`	`X<Y .AND. Y>=2`	`&&`	`x<y && y>=2`
Logical not	`.NOT.`	`.NOT.(X==Y)`	`!`	`!(x==y)`

Operators

Bitwise Operations (modify variables considering bit patterns)

Operation	*Fortran*	*Example*	*C/C++*	*Example*
or	`IOR`	`IOR(10,25) = 27`	`|`	`10 | 25`
and	`IAND`	`IAND(10,25) = 8`	`&`	`10 & 25`
exclusive or	`IEOR`	`IEOR(10,25) = 19`	`^`	`10 ^ 25`
1's complement	`NOT`	`NOT(10) =245=-11`	`~`	`~10`
left shift	`ISHIFT`	`ISHIFT(12,3)= 96`	`<<`	`12 << 3`
right shift	`ISHIFT`	`ISHIFT(12,-3)= 1`	`>>`	`12 >> 3`

```
10 & 25 =  8  ➔  00001010  & 00011001 = 00001000
10 | 25 = 27  ➔  00001010  & 00011001 = 00011011
12 >> 3 =  1  ➔  00001100 >> 3        = 00000001
```

Operators

Conditional operator (?)

The conditional operator evaluates an expression returning a value if that expression is *true* and a different one if the expression is evaluated as *false*.

General form:

```
condition ? result1 : result2
```

If ***condition*** is true the expression will return ***result1***, if it is not it will return ***result2***.

```
2==1 ? 5 : 9;  // returns 9, since 2 is not equal to 1
 5>3 ? a : b;  // returns the value of a
 a>b ? a : b;  // returns whichever is greater, a or b
```

Operators

Explicit Type Casting Operator

Type casting allow you to convert a data of a given type to another.

```
INTEGER I,J
REAL F

I = 3
F = REAL(I)   ! F = 3.0
J = INT(4.8)  ! J = 4
```

```
int i,j;
float f;

i = 3;
f = (float) i;   // in C/C++
f = float(i);    // in C++
j = int(4.8);
```

Operators

`sizeof()` Operator

This operator accepts one parameter , which can be either a type or a variable itself and returns the size in bytes of that type or object

```
#include <iostream.h>

main (){
  int    i;
  float  f;
  double d;
  cout << "sizeof(i)      = " << sizeof(i)      << endl;
  cout << "sizeof(int)    = " << sizeof(int)    << endl;
  cout << "sizeof(f)      = " << sizeof(f)      << endl;
  cout << "sizeof(float)  = " << sizeof(float)  << endl;
  cout << "sizeof(d)      = " << sizeof(d)      << endl;
  cout << "sizeof(double)= " << sizeof(double)<< endl;
}
```

```
sizeof(i)      = 4
sizeof(int)    = 4
sizeof(f)      = 4
sizeof(float)  = 4
sizeof(d)      = 8
sizeof(double) = 8
```

Operators

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

Basic Input/Output

Standard Input

```
PRINT *,"Hello World"
PRINT *,"Hello ","World"
PRINT *,123
PRINT *,"A =", A
PRINT *,(A+B)/2.0
```

```
cout << "Hello World";
cout << "Hello " << "World"
cout << 123;
cout << "a =" << a;
cout << (a+b)/2;
```

Notice that (unlike the **PRINT** statement), **cout** does not add a line break after its output unless we explicitly indicate it.
This is done by inserting a **'\n'** or a using a **endl** manipulator.

```
cout << "First sentence.";
cout << "Second sentence.";
```

```
First sentence.Second sentence.
```

```
cout << "First sentence.\n";
cout << "Second sentence.";
```

```
First sentence.
Second sentence.
```

Basic Input/Output

Standard Output

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

```
INTEGER :: A,B,C
CHARACTER (20) :: Str
READ *,A
READ *,B,C
READ *,Str
```

```
int a,b,c;
string str;
cin >> a;
cin >> b >> c;
cin >> str;
```

Some Mathematical Functions

In C++, you need to include the header: **<math.h>**

```
#include <iostream.h>
#include <math.h>

main ()
{
  double x = 0.5;

  cout << "sin(x)   = " << sin(x) << endl;
  cout << "cos(x)   = " << cos(x) << endl;
  cout << "tan(x)   = " << tan(x) << endl;

  cout << "log(x)   = " << log(x) << endl;
  cout << "log10(x) = " << log10(x) << endl;
}
```

Control Stuctures

Conditional structures: `if else`

```
IF(condition) statement
```

```
if(condition) statement;
```

```
if(condition)
   statement;
```

```
IF(condition) THEN
  statement squence 1
  statement squence 2
END IF
```

```
if(condition){
   statement 1;
   statement 2;
}
```

```
IF(condition) THEN
  statement 1
ELSE
  statement 2
END IF
```

```
if(condition)
  statement 1;
else
  statement 2;
```

Control Stuctures

```
PROGRAM RootFinding
REAL :: A,B,C,D

PRINT *,"Input A,B,C"
READ *,A,B,C
D = B**2-4*A*C

IF(D<0) THEN
 PRINT *,"No real root."
ELSE
  X1 = -B + SQRT(D)/A/2.
  X2 = -B - SQRT(D)/A/2.
  PRINT *,X1,X2
END IF

END PROGRAM
```

```
#include <iostream>

main(){
  float a,b,c,d;

  cout << "input a,b,c: ";
  cin >> a >> b >> c;
  d = b*b-4*a*c;

  if(d<0)
   cout << "No real root.";
  else{
    x1 = -b + sqrt(d)/a/2.;
    x2 = -b - sqrt(d)/a/2.;
    cout << x1 << x2;
  }
}
```

Control Stuctures

The selective structure : `switch`

This is an alternative for the **`if else`** structure.
The aim is to check several possible constant values for an ***`expression`***.

```
SELECT CASE(expression)

  CASE(label list 1)
    statement squence 1
  CASE(label list 2)
    statement squence 2
  ...
  CASE DEFAULT
    default squence;

END SELECT
```

```
switch(expression)
{
  case constant1:
    statement squence 1;
    break;
  case constant2:
    statement squence 2;
    break;
  ...
  default:
    default squence;
}
```

Control Stuctures

```
SELECT CASE(ClassCode)

 CASE(1)
   PRINT *,"Freshman"
 CASE(2)
   PRINT *,"Sophmore"
 CASE(3)
   PRINT *,"Junior"
 CASE(4)
   PRINT *,"Graduate"

 CASE DEFAULT
   PRINT *,"Illegal class"

END SELECT
```

```
switch(ClassCode)
{
 case 1:
   cout << "Freshman" << endl;
   break;
 case 2:
   cout << "Sophmore" << endl;
   break;
 case 3:
   cout << "Junior" << endl;
   break;
 case 4:
   cout << "Graduate" << endl;
   break;

 default:
   cout << "Illegal class\n";
}
```

Control Stuctures

Iterative structures (loops)

Loops have as purpose to repeat a statement a certain number of times.
In C++ there are three basic loop types:

- **counter controlled loops (`for` loops)**
- **`while`**
- **`do-while`**

You can also use the following jump statements:

- **`break`**
- **`continue`**
- **`goto`**

Control Stuctures

I – counter controlled loops

```
DO counter = initial value, limit, step size
   .
   . statement sequence
   .
END DO
```

```
for(initialization; condition; step size)
   statement sequence;
```

```
DO I=1,5,1
   PRINT *,I,I*I
END DO
```

```
  1   1
  2   4
  3   9
  4  16
  5  25
```

```
for (i=1; i<=5; i++)
   cout << i << i*i << endl;
```

```
  1   1
  2   4
  3   9
  4  16
  5  25
```

Control Stuctures

```
#include <iostream>
// evaluates the factorial

main()
{
 int k,n,f;

  cout << "Input n: ";
  cin >> n;

  for(f=1, k=1; k<=n; k++)
    f *= k;

  cout << n << "! = " << f << endl;
}
```

```
Input n: 5
5! = 120
```

Control Stuctures

`while` loops

The ***statement sequence*** is executed as long as the ***condition*** is true, otherwise the loop is skipped.

```
DO WHILE(condition)
   statement sequence
END DO
```

```
while(condition)
   statement sequence;
```

```
J = 0
H = 4.0
DO WHILE(J<5)
  J = J + 1
  H = H/2.0
  PRINT *,J,H
END DO
```

```
j = 0;
h = 4.0;
while(j<5){
 j++;
 h /= 2.0;
 cout << j << h << endl;
}
```

Control Stuctures

`do-while` loops

Its functionality is exactly the same as the **`while`** loop, except that condition in the **`do-while`** loop is evaluated after the execution of statement instead of before.

```
do
   statement squence;
while(condition);
```

```
n = 5;

do
  cout << n << ", ";
while(--n>0);

cout << "FIRE!" << endl;
```

→

```
5, 4, 3, 2, 1, FIRE!
```

Control Stuctures

Jump Statements

```
DO
   ...
   IF(condition) EXIT
   ...
END DO
```

```
for(...){
   ...
   if(condition) break;
   ...
}
```

```
DO
   ...
   IF(condition) CYCLE
   ...
END DO
```

```
for(...){
   ...
   if(condition) continue;
   ...
}
```

```
10 CONTINUE
  ...
IF(condition) GOTO 10
```

```
loop: // a label
   ...
if(condition) goto loop;
```

Functions (subprograms)

General Form:

```
type FUNCTION name(p1,p2,...)
   ...
   name = an expression
   ...
END FUNCTION
```

```
type name(p1,p2,...)
{
   ...
}
```

```
INTEGER Add(A,B)
INTEGER, INTENT(IN) :: A,B
   Add = A+B
END FUNCTION Add
```

```
int add(a,b)
int a,b;{  // obsolete !
   int c;
   c = a+b;
   return c;
}
```

more compact form →

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

Functions

Example Usage of a function:

```
PROGRAM Main
INTEGER :: X=2, Y=4, Z, Add
 Z = Add(X,Y)
 PRINT *,Z
END PROGRAM Main

! External function
INTEGER Add(A,B)
INTEGER, INTENT(IN) :: A,B
  Add = A+B
END FUNCTION Add
```

```
#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;
}
```

Functions

Function prototype:

```
#include <iostream.h>

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.h>

// 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

Functions with no type

```
#include <iostream.h>

// no value is returned
void printDouble(int a)
{
  cout << "Double of a:" << 2*a;
}

main()
{
  printDouble(5);
}
```

```
Double of a: 10
```

```
#include <iostream.h>

// no value is returned
void Message(void)
{
 cout << "I am a function";
}

main()
{
  Message();
}
```

```
I am a function
```

Functions

Arguments passed by value and by reference

```
#include <iostream.h>

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

main(){
  int x=3, y=8;

  cout << " x= " << x ;
  cout << " y= " << y << endl;
  Decrease(x,y);
  cout << "x= " << x ;
  cout << "y= " << y << endl;
}
```

```
x=3 y=8
x=3 y=8
```

```
#include <iostream.h>

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

main(){
  int x=3, y=8;

  cout << " x= " << x ;
  cout << " y= " << y << endl;
  Decrease(x,y);
  cout << "x= " << x ;
  cout << "y= " << y << endl;
}
```

```
x=3 y=8
x=2 y=7
```

Functions

A function may return more than ONE value using references:

```fortran
PROGRAM Main
REAL    :: Rx , X = 3.2
INTEGER :: Ix

  CALL Convert(X,Ix,Rx)
  PRINT *,"X  = ",X
  PRINT *,"Ix = ",Ix
  PRINT *,"Rx = ",Rx

END PROGRAM

SUBROUTINE Convert(Num,Ip,Rp)
REAL,     INTENT(IN)   :: Num
INTEGER,  INTENT(OUT)  :: Ip
REAL,     INTENT(OUT)  :: Rp
  Ip = Num
  Rp = Num - INT(Num)
END SUBROUTINE
```

```
X  = 3.2
Ix = 3
Rx = 0.2
```

```cpp
#include <iostream.h>

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);
}
```

Functions

Variable number of arguments (Default arguments)

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

Consider the second order polynomial function: $a + bx + cx^2$

```
PROGRAM Main
REAL :: x = 1.0

 PRINT *,"p(x,7)    = ",p(x,7.0)
 PRINT *,"p(x,7,6)  = ",p(x,7.0,6.0)
 PRINT *,"p(x,7,6,3)= ",p(x,7.0,6.0,3.0)

CONTAINS

 REAL FUNCTION P(X,A,B,C)
 REAL, INTENT(IN) :: X,A
 REAL, INTENT(IN), OPTIONAL :: B,C
   P = A
   IF( PRESENT(B) ) P = P + B*X
   IF( PRESENT(C) ) P = P + C*X**2
 END FUNCTION P

END PROGRAM Main
```

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

Functions

```
#include <iostream.h>

// -- optional parameters must all be listed last --
double p(double, double, 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.
```

Functions

Overloading Functions

```
#include <iostream.h>

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
```

Arrays

Decleartion of an Array

An *array* is a squence of objects all of which have the same type.

A four-element array:

```
INTEGER :: A(4)
```

```
int a[4];
```

Index values: 1, 2, 3, …,N

A(1), A(2), A(3), A(4)

0, 1, 2, …,N-1

a[0], a[1], a[2], a[3]

Reading and Printing an array:

```
PROGRAM Array
INTEGER :: A(4)

 READ *,A
 PRINT *,A

END PROGRAM
```

```
main(){
  int a[4];

  for(int i=0; i<4; i++)
    cin >> a[i];

 for(int i=0;i<4;i++)
    cout << a[i];
}
```

Arrays

Initializing Arrays

```
INTEGER :: A(4)

A(1) = 22
A(2) = 33
A(3) = 44
A(4) = 77
```

```
int a[4];

a[0] = 22;
a[1] = 33;
a[2] = 44;
a[3] = 77;
```

or

```
INTEGER :: A(4)=(/22,33,44,77/)
```

```
int a[4] = {22,33,44,77};
```

```
// compiler will assume
// size of the array is 4
int a[] = {22,33,44,77};
```

Assigning all elements to zero:

```
INTEGER :: A(4)
A = 0
```

```
int a[4] = {0};
```

Arrays

Multidimensional Arrays

```fortran
REAL :: A(4)       ! vector
REAL :: B(2,3)     ! Matrix
REAL :: C(5,2,4)
```

```cpp
double a[4];         // vector
double a[2][3];      // matrix
double c[5][2][4];
```

```fortran
PROGRAM Arrays
INTEGER, PARAMETER :: N=5, M=4
INTEGER :: I,J, A(N,M)

  DO I=1,N
  DO J=1,M
    A(I,J) = I*J
  END DO
  END DO

  DO I=1,N
    PRINT *,A(I,:)
  END DO

END PROGRAM
```

1	2	3	4
2	4	6	8
3	6	9	12
4	8	12	16
5	10	15	20

```cpp
#include <iostream.h>

main(){
  const int n=5, m=4;
  int i,j, a[n][m];

  for(i=0; i<n; i++)
  for(j=0; j<m; j++)
    a[i][j] = (i+1)*(j+1);

  for(i=0; i<n; i++){
    for(j=0; j<m; j++){
     cout << a[i][j] << " ";
    }
    cout << '\n';
  }
}
```

Arrays

Passing an Array to a Function

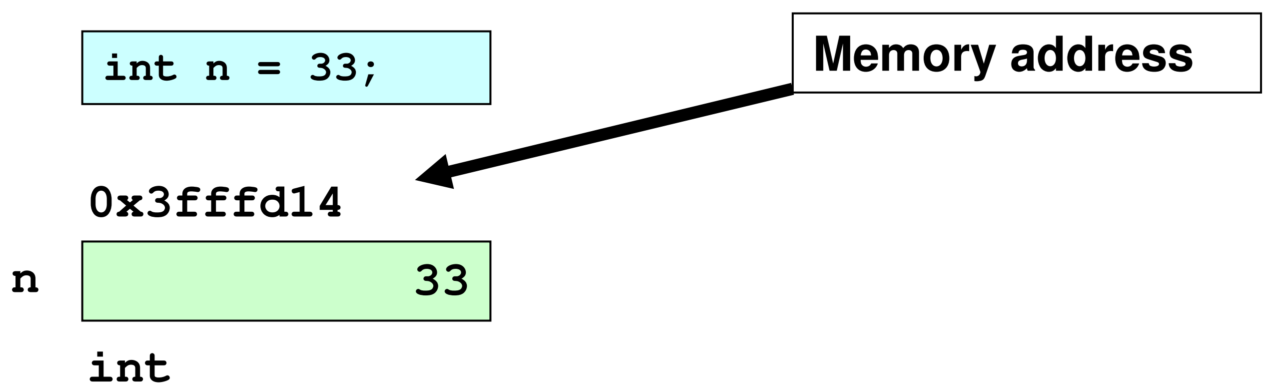
```
PROGRAM ArrayFunc
REAL :: A(4),Eb, Max

  A  = (/1.0, 6.1, 3.4 ,5.8/)
  Eb = Max(A)

  PRINT *,"Biggest is ",Eb

END PROGRAM

REAL FUNCTION Max(A)
REAL, INTENT(IN) :: A(:)
INTEGER :: I

  Max = A(1)
  DO I=2,SIZE(A)
    IF(A(I)>Max) Max = A(I)
  END DO

END FUNCTION
```

```
#include <iostream.h>

float Max(float x[],int);

main(){
 float a[4] = {1.0,6.1,3.4,5.8};
 float eb;

   eb = Max(a,4);
   cout << "Biggest is " << eb;
}

float Max(float x[],int size){
  float max = a[0];

  for(int i=1; i<size; i++)
    if(a[i]>max) max = a[i];

  return max;
}
```

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.h>

// printing both the value and address

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

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

Pointers and References

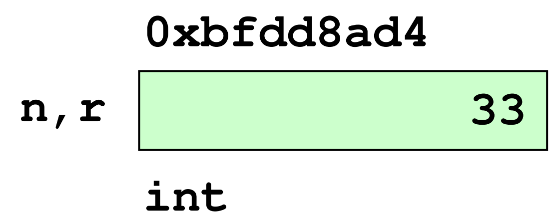
References

The **reference** is an *alias,* a *synonym* for a variable.

It is declerated by using the *address operator* **&**.

```
#include <iostream.h>
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
```



Pointers and References

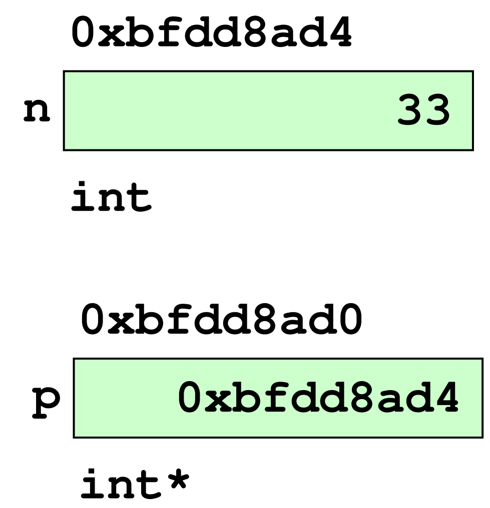
Pointers

The address operator returns the memory adress of a variable.

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

```
#include <iostream.h>

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
```



Pointers and References

In Fortran pointer variable is decelerated by **`POINTER`** attribute, to point a variable whose attribute must be **`TARGET`**.

In C/C++ you can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*) called *dereference operator*.

```
PROGRAM PointerExample

INTEGER, TARGET  :: N = 33
INTEGER, POINTER :: P

  P => N  ! P points to N
  PRINT *,"N P: ",N,P

  P = 66
  PRINT *,"N P: ",N,P

END PROGRAM
```

```
#include <iostream.h>

main(){
 int  n = 33;
 int *p;

 p = &n; // p points to n
 cout << "n *p: " << n << *p <<endl;

 *p = 66;
 cout << "n *p: " << n << *p <<endl;
}
```

```
N P: 33 33
N P: 66 66
```

```
n *p: 33 33
n *p: 66 66
```

Pointers and References

Use of Pointers in Functions

```
PROGRAM Swapping
REAL, POINTER :: PA,PB
REAL, TARGET  :: A = 11.0
REAL, TARGET  :: B = 22.0

  PA => A
  PB => B

  PRINT *,"A B: ",A,B
  CALL Swap(PA,PB)
  PRINT *,"A B: ",A,B

END PROGRAM

SUBROUTINE Swap(X,Y)
REAL, POINTER :: X,Y
REAL, POINTER :: Z
  Z => X
  X => Y
  Y => Z
END SUBROUTINE
```

```
#include <iostream.h>

void Swap(float *, float *);

main(){
  float *pa, *pb;
  float a = 11.0, b =22.0;

  pa = &a;
  pb = &b;

  cout << "a b : " << a << b << endl;
  Swap(pa,pb);
  cout << "a b : " << a << b << endl;
}

void Swap(float *x, float *y){
  float z;
  // z equal to value pointed by x
   z = *x;
  *x = *y;
  *y =  z;
}
```

Pointers and References

The **Swap** function can be re-written without using a pointer.

```
PROGRAM Swapping
REAL :: A = 11.0, B = 22.0

  PRINT *,"A B: ",A,B
  CALL Swap(A,B)
  PRINT *,"A B: ",A,B

END PROGRAM

SUBROUTINE Swap(X,Y)
REAL, INTENT(INOUT) :: X,Y
REAL :: Z
  Z = X
  X = Y
  Y = Z
END SUBROUTINE
```

```
A B: 11.0 22.0
A B: 22.0 11.0
```

```
#include <iostream.h>

void Swap(float &, float &);

main(){
  float a = 11, b =22;

  cout << "a b : " << a << b << endl;
  Swap(a,b);
  cout << "a b : " << a << b << endl;
}

void Swap(float& x, float& y)
{
   float z;
   z = x;
   x = y;
   y = z;
}
```

Pointers and References

Pointers and Arrays

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element.

Therefore the array name is a constant pointer.

Consider the declaration:

```
int numbers[20];
int *p;
```

Following assignment is valid (since array name is a constant pointer):

```
p = numbers;
```

The following assignments are equivalent:

```
numbers[4] = 25;
*(p+4) = 25;
```

Pointers and References

Pointer Arithmetics

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer data types.

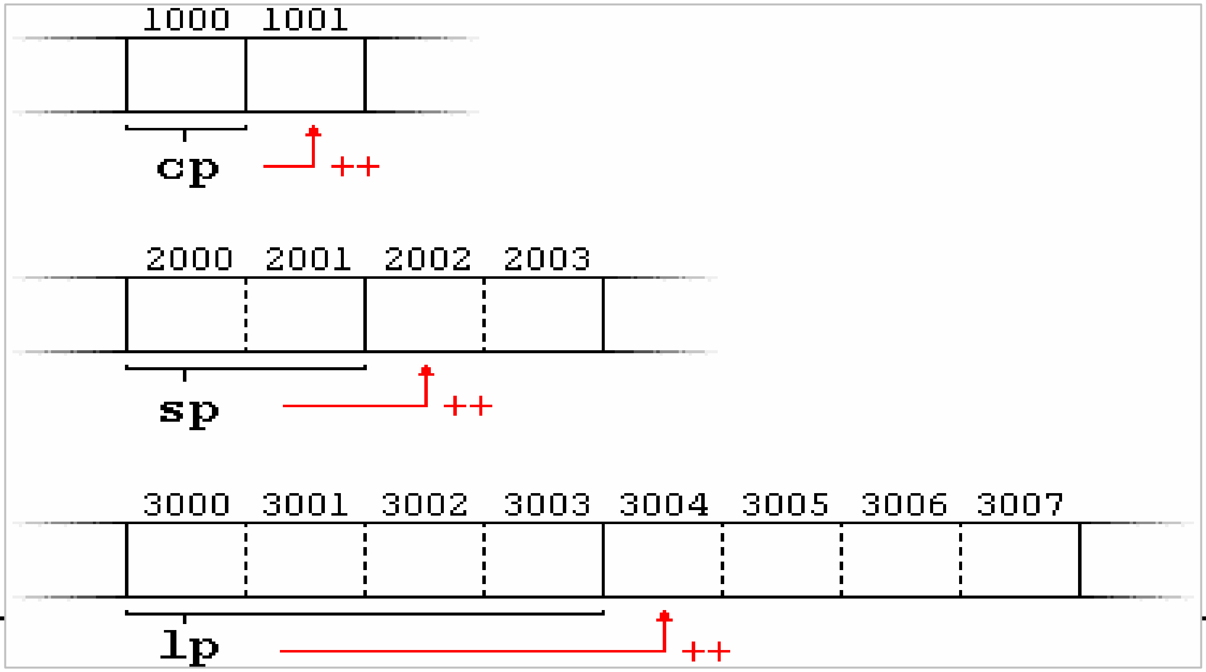
Suppose that we define three pointers in this compiler:

```
char  *cp;
short *sp;
long  *lp;
```

Let they point to memory locations 1000, 2000 and 3000 respectively.

If we write:

```
cp++;
sp++;
lp++;
```



Pointers and References

Both the increase (**++**) and decrease (**--**) operators have greater operator precedence than the dereference operator (*).

Following expressions may lead to confusion:

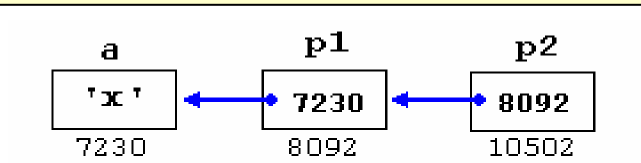
```
*p++; // equivalent to *(p++);
```

Pointers and References

Pointers to Pointers

C++ allows the use of pointers that point to pointers.

```
PROGRAM TwoPointers
CHARACTER,TARGET :: A = 'x'
CHARACTER,POINTER :: P1,P2

  P1 => A
  P2 => P1

  PRINT *,A,P1,P2

  P1 = 'y'
  PRINT *,A,P1,P2

  P2 = 'z'
  PRINT *,A,P1,P2

END PROGRAM
```




```
#include <iostream.h>

main(){
  char   a = 'x';
  char*  p1;
  char** p2;

  p1 = &a;
  p2 = &p1;

  cout << a << *p1 << **p2 << endl;

  *p1 = 'y';
  cout << a << *p1 << **p2 << endl;

 **p2 = 'z';
  cout << a << *p1 << **p2 << endl;
}
```

```
x x x
y y y
z z z
```

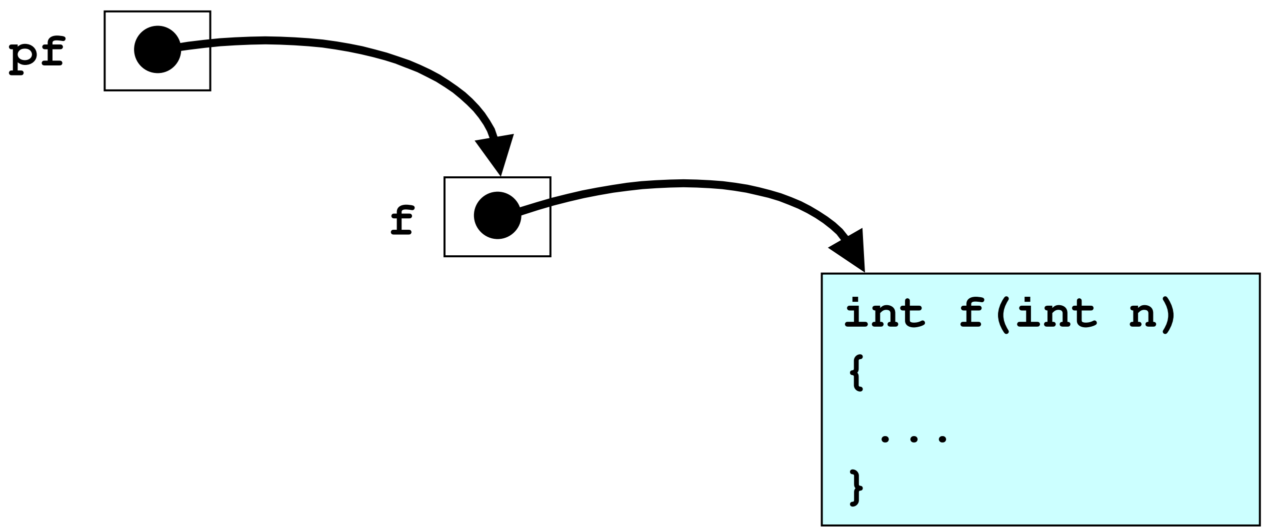
Pointers and References

Pointers to Functions

Like an array name, a function name is actually a constant pointer.

A pointer to a function is a pointer whose value is the address of the function name. Consider the declaration:

```
int f(int);       // decleres func. f
int (*pf)(int);   // decleres func. pointer pf
pf = &f;          // assigns address of f to pf
```



Pointers and References

```
// pointer to functions
#include <iostream.h>

int square(int);
int cube(int);
int sum(int (*)(int), int);

main ()
{
 cout << sum(square,4) << endl;
 cout << sum(cube,4) << endl;
}

int square(int x){
  return x*x;
}

int cube(int x){
  return x*x*x;
}
```

```
// returns f(1)+f(2)+ ... +f(n)
int sum(int (*pf)(int x), int n)
{
  int i,s=0;
  for(i = 1; i <= n; i++)
    s += (*pf)(i);
  return s;
}
```

```
30
100
```

Dynamic Memory

The `new` Operator

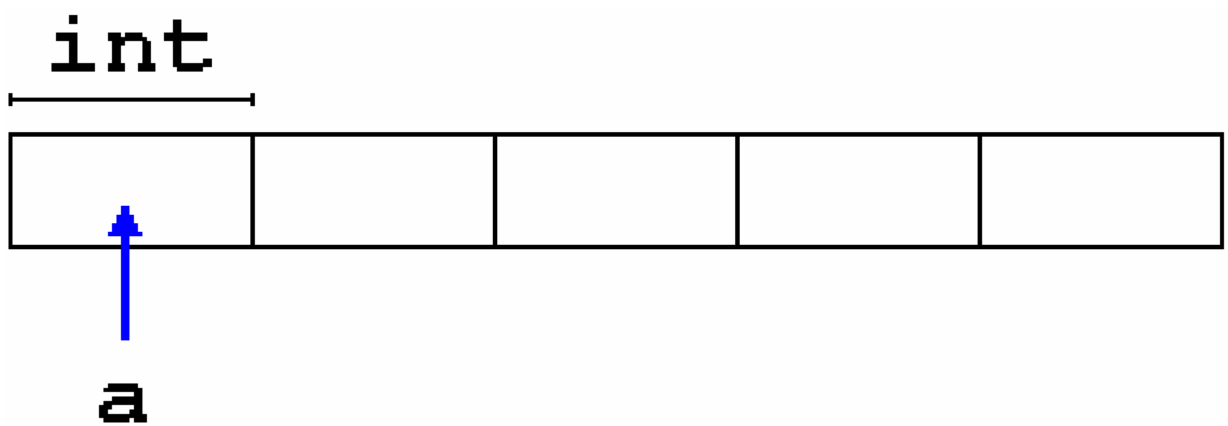
In order to request dynamic memory we use the operator **`new`**.

General form:

```
pointer = new type // single
pointer = new type [number_of_elements];
```

For example:

```
int * a;
a = new int [5];
```



Dynamic Memory

The `delete` Operator

`delete` operator reverses the action of the **`new`** operator, that is it frees the allocated memory by the **`new`** operator.

General form:

```
delete pointer     // for a single pointer
delete [] pointer
```

For example:

```
delete [] a;
```

Dynamic Memory

```fortran
PROGRAM DynamicMemory
! mean of n numbers
REAL, ALLOCATABLE :: X(:)
REAL      :: Mean
INTEGER :: N

DO
  PRINT *,"How many elements:"
  READ *,N

  IF (N<=0) EXIT

  ALLOCATE(X(N))
  PRINT *,"Input elements:"
  READ *,X

  Mean = SUM(X)/N
  PRINT *,"Mean = ",Mean

  DEALLOCATE(X)
END DO

END PROGRAM
```

```cpp
#include <iostream.h>
// mean of n numbers
main (){
  float *x, mean,s;
  int i,n;

  while(1){
    cout << "How many elements: ";
    cin >> n;

    if(n<=0) break;

    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 << endl;
    delete [] x;
  }
}
```

Dynamic Memory

Here is a sample output of the previous program(s):

```
How many elements: 3
Input elements: 1 2 3
Mean = 2.0
How many elements: 6
Input elements: 2 4 5 9 1 0
Mean = 3.5
How many elements: 0
```

Dynamic Memory

Dynamic Memory in ANSI C

Operators **`new`** and **`delete`** are exclusive of C++.

They are not available in the C language. But using pure C language, dynamic memory can also be used through the functions

`malloc`, **`calloc`**, **`realloc`** and **`free`**, defined in **`<cstdlib.h>`**

An example usage: (this is not recommended in C++)

```
double *array; /* decleration */
int n;

scanf("%d",&n); /* read number of elements */

/* allocate the memory */
array = (double *) malloc(sizeof(double)*n);

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

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

Data Structures

Fortran and C/C++ allow you to define your own data types.

- A data structure (or derived data types) is a group of data elements grouped together under one name.
- These data elements, known as *members*, can have different types and different lengths.

General forms:

```
TYPE name
type1 member_name1;
type2 member_name2;
.
.
END TYPE name
```

```
struct name {
type1 member_name1;
type2 member_name2;
.
.
} object_names;
```

```
TYPE Student
  CHARACTER (15) :: Name
  INTEGER :: MT1,MT2,FIN
END TYPE Student
```

```
struct Student{
  string name;
  int mt1, mt2, fin;
} std1, std2;
```

Data Structures

```fortran
PROGRAM Structure
IMPLICIT NONE

 TYPE Product
   INTEGER :: Weight
   REAL :: Price
 END TYPE Product

 TYPE(Product) :: Apple, Banana;
 REAL :: TA,TB

 Apple%Weight  = 10
 Apple%Price   = 1.50
 Banana%Weight = 12
 Banana%Price  = 3.75

 TA=  Apple%Weight * Apple%Price
 TB= Banana%Weight * Banana%Price

 PRINT *,"Total Prices",
 PRINT *,"Apple : ",TA
 PRINT *,"Banana: ",TB

END PROGRAM
```

```cpp
#include <iostream.h>

struct product{
  int weight;
  float price;
};

main ()
{
  product  apple, banana;
  float ta,tb;

  apple.weight  = 10;
  apple.price   = 1.50;
  banana.weight = 12;
  banana.price  = 3.75;

  ta=  apple.weight *  apple.price;
  tb= banana.weight * banana.price;

  cout << "Total Prices" << endl;
  cout << "Apple : " << ta << endl;
  cout << "Banana: " << tb << endl;
}
```

Other Data Types

Defined Data Types

C++ allows the definition of our own types based on other existing data types. This is done by **`typedef`** keyword having general form:

```
typedef existing_type new_type
```

```
#include <iostream.h>

typedef int     INTEGER;
typedef float   REAL;

main (){
  INTEGER i = 33;
  REAL    r = 45.0;

  cout << i << r << endl;
}
```

```
#include <iostream.h>

#define PROGRAM_Main      main()
#define IMPLICIT_NONE     {
#define END_PROGRAM       }
#define PRINT             cout

typedef int     INTEGER;
typedef float   REAL;

PROGRAM_Main
IMPLICIT_NONE
INTEGER i = 33;
REAL    r = 45.0;

  PRINT << i << r;

END_PROGRAM
```

Other Data Types

Enumerations

Enumerations create new data types to contain something different that is not limited to the values fundamental data types may take.

```
enum type_name{enumerator _list}
```

For example, we could create a new type of variable called color to store colors with the following declaration:

```
enum Color_t {black, blue, green, red, gray};
```

We can then declare variables of this type:

```
Color_t c1,c2;
c1 = black; // c1 = 0;
c2 = green; // c2 = 2;
if(c1==c2) cout << "same color.\n";
```

Other Data Types

```
#include <iostream.h>

enum Mount{Jan=1, Feb, Mar, Apr, May,
           Jun, Aug, Sep, Oct, Nov, Dec};

enum Base{Binary=2, Octal=8, Decimal=10,
          Hexadecimal=16};

main(){
  Mount m = Apr;
  Base  b = Hexadecimal;

  cout << "Mount : " << m << ", ";
  cout << "Base  : " << b << endl;

  m = Jun;
  b = Decimal;

  cout << "Mount : " << m << ", ";
  cout << "Base  : " << b << endl;
}
```

```
Mount = 4, Base = 16
Mount = 6, Base = 10
```

Classes

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

- An ***object*** is an instantiation of a class. In terms of variables, a class would be the *type*, and
an object would be the *variable*.

- Classes are declerated 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 Cylinder {
    double pi;
    double r,h;
  public:
    void set_values(double,double);
    double volume();
} my_cylinder;
```

- declares a class (i.e., a type) called **`Cylinder`** and an object (i.e., a variable) of this class called **`my_cylinder`**.
- The functions: **`set_values()`** and **`volume()`** are called *member functions* or *methods*.
- Member **`pi,r`** and **`h`** have (default) **`private`** access and member functions have **`public`** access.

Classes

```
#include <iostream.h>

class Cylinder{
  private:
    double pi, r, h;
  public:
    void set_values(double,double);
    double volume();
};

main(){
  Cylinder c;
  c.set_values(1.5,2);
  cout << "volume: " << c.volume();
}

void Cylinder::set_values(double R,double H){
  r = R;
  h = H;
  pi= 3.141593;
}

double Cylinder::volume(){
  return (pi*r*r*h);
}
```

```
volume: 14.137168
```

Classes

Classes in C++ can be considered to be modules in Fortran 95.

Modules in Fortran 95	*Classes in C++*
Contain member data and functions.	Contain member data and functions.
Can be used in any other programs after including **USE** statement. **USE *module_name***	Can be used in any other programs after declaring objects of the class type like other variables. ***class_name object_name;***
Members are accessed by directly calling their names.	Members are *not* accessed directly. First you should call the object: ***object_name.member;***
Default access specifier is **PUBLIC**	Default access specifier is **private**
Can be a separate file and compiled to an object or library that can be linked with a main program.	Can be a separate file and compiled to an object or library that can be linked with a main program.

Classes

```
MODULE Cylinder
 REAL, PRIVATE :: pi,r,h;

 CONTAINS

  SUBROUTINE Set_Values(x,y)
  REAL,INTENT(IN) :: x,y
    r = x
    h = y
    pi = 3.141593
  END SUBROUTINE

  REAL FUNCTION Volume()
    Volume = pi*r*r*h
  END FUNCTION

END MODULE

PROGRAM Main
 USE Cylinder

 CALL Set_Values(1.5,2.0)
 PRINT *,"Volume: ",Volume()

END PROGRAM
```

```
#include <iostream.h>

class Cylinder{
  private:
    double pi, r, h;
  public:
    void set_values(double,double);
    double volume();
};

void
Cylinder::set_values(double x,double y){
  r = x;
  h = y;
  pi= 3.141593;
}

double Cylinder::volume(){
  return (pi*r*r*h);
}

main(){
  Cylinder c;
  c.set_values(1.5,2);
  cout << "Volume: " << c.volume();
}
```

Classes

Self Contained Implementation

Here is the same **`Cylinder`** class with the definitions of its member functions included within the class decleration.

```
#include <iostream.h>

class Cylinder{
  private:
    double pi,r, h;
  public:
    void set_values(double R,double H){
      r = R;
      h = H;
      pi= 3.141593;
    }
    double volume(){
     return (pi*r*r*h);
   }
};

main(){
  Cylinder c(1.5,2.0);
  cout << "Volume: " << c.volume();
}
```

Classes

Constructors

In the **`Cylinder`** class **`set_values()`** function initialize its objects. It would be more natural to have this initialization occur when objects are declared.

A ***constructor*** is a member function that is called automatically when an object is declared.

A constructor function must have the same name as the class itself, and declared without return type.

Classes

```
#include <iostream.h>
// example: class constructor
class Cylinder{
  private:
    double pi,r, h;
  public:
    Cylinder(double,double);
    double volume(){return (pi*r*r*h); }
};

Cylinder::Cylinder(double R,double H){
  r = R;
  h = H;
  pi= 3.141593;
}

main(){
  Cylinder c(1.5,2);
  cout << "Volume: " << c.area();
}
```

```
Volume: 14.137168
```

Pointers to Classes

It is perfectly valid to create pointers that point to classes.

For example:

```
Cylinder * pc;
```

is a pointer to an object of class **Cylinder**.

In order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (**->**) of indirection.

Classes

```
#include <iostream.h>

class Cylinder{
    double pi,r,h;
  public:
    void set_values(double,double);
    double volume(){return (pi*r*r*h);}
};
void Cylinder::set_values(double R,double H){
  r = R;
  h = H;
  pi= 3.141593;
}

main () {
  Cylinder c, *p;

  c.set_values(1,2);
  cout << "c  volume: " << c.volume() << endl;

  p = &c;  // p points to c
  p->set_values(3,4);
  cout << "c  volume: " << c.volume() << endl;
  cout << "*p volume: " << p->volume()<< endl;
}
```

```
c  volume: 6.283186
c  volume: 113.097348
*p volume: 113.097348
```

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 << ")";
}
```

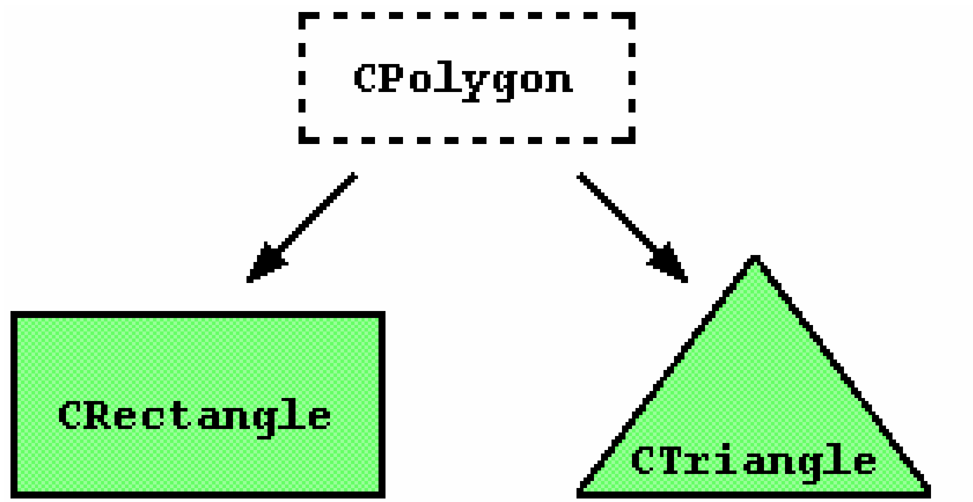
```
c = (4,3)
```

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

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:

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

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

Linked Lists

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
```

```
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
```

```
#include <iostream.h>

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;
  }
}
```

END OF SEMINAR