Introduction to C++

Getting Started with C++

- C++ is derived from the C language.
- It is a superset of C.
- C++ was originally called “C with classes.”
- // **FIRST.CPP C++ programm**

```
#include <iostream>
using namespace std;
int main()
{
   cout << “Hello JUET\n”;
   return 0;
}
```


*

Header Files

- The preprocessor directive #include tells the compiler to add the source file IOSTREAM to the FIRST.CPP source file before compiling.
- **Why these files are required?**
- IOSTREAM is an example of a *header file*
- *It's concerned with basic input/output* operations, and contains declarations that are needed by the cout identifier and the << operator.
- Without these declarations, the compiler won't recognize cout and will think << is being used incorrectly.
- There are many such header files.

*

using namespace std

- Namespace allows organizing the elements of programs into different logical scopes to prevent **name collisions** that can occur due to inclusion of multiple libraries.
- In other words, namespaces allow us to group named entities (like function, variables etc.) into narrower scopes.
- Namespace is a feature in C++ and not available in C.
- Namespaces were added in the C++ in 1995.

*

using namespace std;

- The directive **using namespace std**; says that all the program statements that follow can use all the objects, macros, functions within the **std** namespace.
- Various program components such as **cout** are declared within this namespace.
- **Is it necessary to write "using namespace std"?**

  No! If we do not use the **using** directive, we would need to add the **std** name to each program elements.
  For example, in the program we need to write
  std::cout << "Hello JUET.";

  Therefore to avoid adding std:: several times in programs we use the **using** directive.

*

cout object

- The identifier cout (pronounced as “C out”) is actually an *object*.
- It is predefined in C++.
- The operator << is called the *insertion* operator.
- *insertion* operator sends contents of the variable on its right to the object on its left.
- In FIRST program it directs the string constant “Hello JUET!\n” to cout, which sends it to the display.

*

Variable in C++

- // demonstrates integer variables

```
#include <iostream>
using namespace std;
int main(){
   int var1; //define var1
   int var2; //define var2
   var1 = 20; //assign value to var1
   var2 = var1 + 10; //assign value to var2
   cout << “var1+10 is “; //output text
   cout << var2 << endl; //output value of var2
   return 0;
}
```


*

The endl Manipulator

- endl will insert a new line to the stream, so that subsequent text is displayed on the next line.
- It has the same effect as sending the '\n' character, but is somewhat clearer.

*

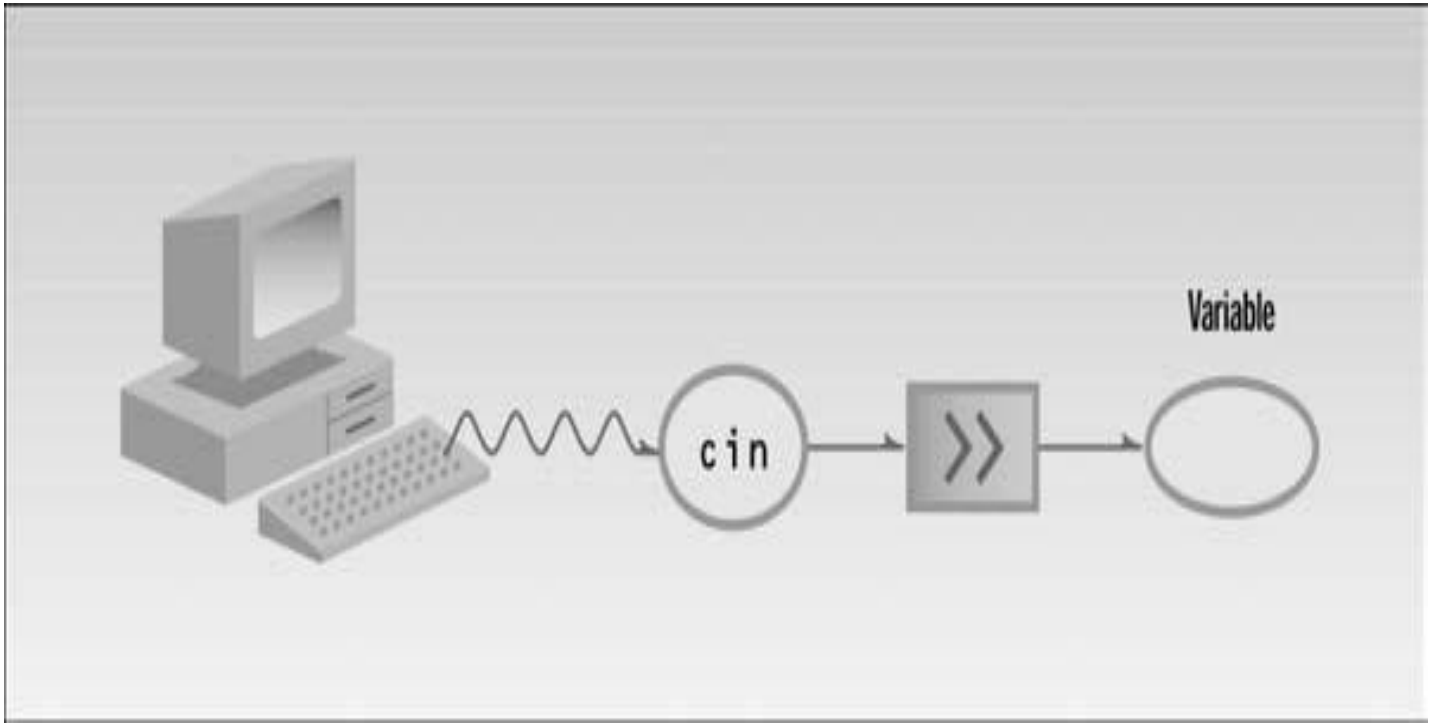
Input with cin

```
// fahren.cpp
// demonstrates cin, newline
#include <iostream>
using namespace std;
int main(){
   int ftemp; //for temperature in fahrenheit
   cout << "Enter temperature in fahrenheit: ";
   cin >> ftemp;
   int ctemp = (ftemp-32) * 5 / 9;
   cout << "Equivalent in Celsius is: " << ctemp <<
     '\n';
   return 0;
}
```


*

Input with cin (Cont...)

- The >> is the *extraction* operator. *It takes the value* from the stream object on its left and places it in the variable on its right.



*

Program to demonstrates floating point variables

```
// circarea.cpp  for demonstrates floating point
   variables
#include <iostream> //for cout, etc.
using namespace std;
int main(){
   float rad;                    //variable of type float
   const float PI = 3.14159F;     //type const float
   cout << "Enter radius of circle: ";      //prompt
   cin >> rad;                    //get radius
   float area = PI * rad * rad;              //find area
   cout << "Area is " << area << endl;  //display
   answer
    return 0;
```

O/P: Enter radius of circle: 0.5
Area is 0.785398

const Qualifier Vs #define Directive

- **The const Qualifier**
  - const float PI = 3.14159F; //type const float
  - The keyword **const** precedes the data type of a variable.
  - It specifies that the value of a variable will not change throughout the program.
  - Any attempt to alter the value of a variable defined with this qualifier will throw an error message from the compiler.
- **The #define Directive**
  - #define PI 3.14159
  - It specifies that the identifier PI will be replaced by the text 3.14159 throughout the program before compilation.
  - CONSTs are handled by the compiler, where as

Cascading the Insertion Operator and Extraction Operator

- << and >> operators can be cascaded for displaying or taking input for more than on variable.
- Example:
  - cin>>var1>>var2>>var3;
  - cin>>a>>b;
  - cout<<var1<<“ ”<<var2<<“ ”<<var3<<endl;

*

Library Functions

```
// sqrt.cpp
// demonstrates sqrt() library function
#include <iostream>   //for cout, etc.
#include <cmath> /     /for sqrt()
using namespace std;
int main()
{
      double number, answer;     //sqrt() requires type double
      cout << “Enter a number: “;
      cin >> number; //get the number
      answer = sqrt(number);      //find square root
      cout << “Square root is “<< answer << endl;        //display it
      return 0;
}
```


*

Relational Operators

Operator	*Meaning*
>	Greater than (greater than)
<	Less than
==	Equal to
!=	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

Examples:

```
jane = 44;      //assignment statement
harry = 12;     //assignment statement
(jane == harry)      //false
(harry <= 12)        //true
(jane > harry) //true
(jane >= 44)  //true
(harry != 12)   // false
(7 < harry)     //true
(0)          //false (by definition)
(44)          //true (since it's not 0)
```


*

Relational Operators

```
// relat.cpp for demonstrates relational
   operators
#include <iostream>
using namespace std;
int main() {
   int numb;
   cout << "Enter a number: ";
   cin >> numb;
   cout << "numb<10 is " << (numb < 10) <<
   endl;
   cout << "numb>10 is " << (numb > 10) <<
   endl;
   cout << "numb==10 is " << (numb == 10) <<
```

Enter a number: 20
numb<10 is 0
numb>10 is 1
numb==10 is 0

Loops

- **The for Loop**

```
// fordemo.cpp for demonstrates simple FOR loop
#include <iostream>
using namespace std;
int main()
{
    int j; //define a loop variable
    for(j=0; j<15; j++) //loop from 0 to 14,
        cout << j * j << " "; //displaying the square of j
    cout << endl;
    return 0;
}
```

Note: while Loop and do while loop have same syntax as they have in C.

*

Problem

```
#include <iostream>
using namespace std;
int main()
{
int count = 10;
cout << “count=” << count << endl; //displays 10
cout << “count=” << ++count << endl; //displays 11 (prefix)
cout << “count=” << count << endl; //displays 11
cout << “count=” << count++ << endl; //displays 11 (postfix)
cout << “count=” << count << endl; //displays 12
return 0;
}
```

```
count=10
count=11
count=11
count=11
count=12
```

- Q2:Assuming var1 starts with the value 20, what will the following code fragment print out?
  cout << var1--; cout << ++var1;
- Q3: header files are used for what purpose?
- Q4: The actual code for library functions is contained in a _____ file.

*

Functions in C++

- A function is a group of statements that together perform a task.
- The function can then be invoked from other

Function Components

Component	*Purpose*	*Example*
Declaration (prototype)	Specifies function name, argument types, and return value. Alerts compiler (and programmer) that a function is coming up later.	`void func();`
Call	Causes the function to be executed.	`func();`

Definition	The function itself. Contains the lines of code that constitute the function.	`void func()`
		`{`
		`// lines of code`
		`}`

Declarator	First line of definition.	`void func()`

*

Argument names in the function declaration

- The names of the variables in the declaration are optional.
- For example, suppose you have a function that displays a point on the screen. Then following two declarations mean exactly the same thing to the compiler.

- **void display_point(int, int); //declaration**

  or
- **void display_point(int horiz, int vert); //declaration**

*

Call by Value

```
void repchar(char, int);        //function declaration
int main(){
        char ch;
        int n;
        cout << “Enter a character: “;
        cin >> ch;
        cout << “Enter number of times to repeat it: “;
        cin >> n
        repchar(ch, n);                 //function call
        return 0;
}
void repchar(char ch1, int n1)    //function declarator
{
        for(int j=0; j<n1; j++)         //function body
                cout << ch1;
        cout << endl;
}
```


*

Reference Variables in C++

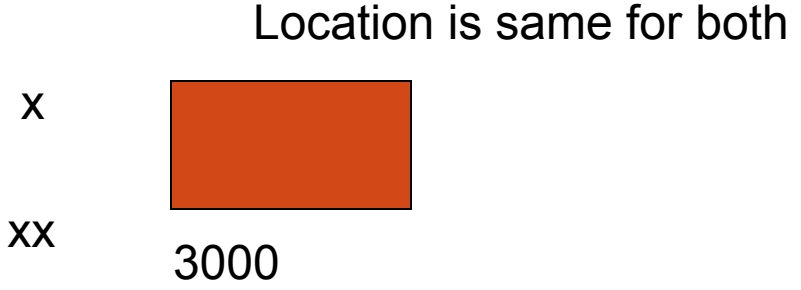
- A reference variable is an alias (*alternate name*) for an existing variable.

```
int x;      // x is a variable
int& alias_x=x;  // alias_x is a reference
```

- To create a reference variable we simply put '&' sign and equate it to an existing variable of same data type
- Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable..

Reference Variable Example

```
#include <iostream>
using namespace std;
int main()
{
  int x;
  int& xx=x; //xx is a reference of x
  x=10;
  cout<<endl<<x<<xx; //both are 10
  xx++;
  cout<<endl<<x<<xx; //both are 11
  return 0;
}
```

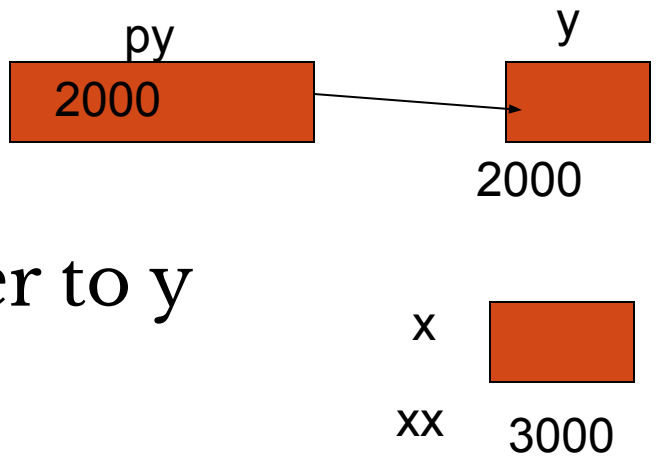


Differences between references and pointers

1. A reference may not occupy any space as it is a alternative name for a variable.; but pointer has their own memory locations.

```
int x;
int& xx=x; // a reference

int y;
int* py=&y; // py is a pointer to y
```



Differences between references and pointers

2. A reference must be initialized at place of declaration. Its declaration cannot be deferred from its initialization. The initialization of a pointer can be deferred.

```
int x;
int& xx; // Error: not possible
xx=x;
//////////////////////////////////////
int y;
int* py;
py=&y;  // used frequently
```

Differences between references and pointers

3. A reference once created and bound to a variable, cannot be reinitialized to another variable. A pointer can be reinitialized any time.

```
int x,y;
int& xy=x; // a reference
xy=y; // This doesn't mean xy has become a reference of y
////////////////////////////////
int x,y;
int * pxy=&x;    //p xy is a pointer to x
pxy=&y;          // Now pxy pointing to y
```

Differences between references and pointers

4. There is no concept of NULL reference. You must always be able to assume that a reference is connected to a legitimate piece of storage. There exist NULL pointers.

5. A function returning a reference can appear on the left hand side of the assignment operator. There is no such concept with pointers

reference

```
#include <iostream>
using namespace std;
static int a=20;
int& setValues( )
{
      return a;   // return a reference to the variable a
}
int main ()
{
   cout << "Value before change:" <<a<<endl;

   setValues() = 500; // change the value of a

   cout << "Value after change:" <<a<<endl;
   return 0;
}
```

O/p:
Value before change: 20
Value after change :500

Pass-by-Reference

```
void change(int&,int); //prototype
int main()
{
     int a=10,b=20;
    change(a,b);   //function called
    cout<<endl<<“a=”<<a<<“b=”<<b;
return 0;                                                    a=11, b=20
}
/*function to change two parameters,*/
void change(int& x, int y)
{
    cout<<endl<<x<<y;    //x=10, y=20
    x++;
    y++;
    cout<<endl<<“ x=”<<x<<“y=”<<y;  }
                                                             x=11 y=21
```

Similarities between Reference and Pointers

1. Both can be used as formal parameters in a function.

   void swap(int *, int *);

   void swap(int &, int &);

2. Both can be return from a function.

   int* fun();

   int& fun();

3. A variable can have any number of references or pointers

Advantages of Pass-by-Reference

- It saves space, because no extra space is reserved for reference parameter
- It saves time, because there is no data copying from calling function to called function
- **References are safer to use:** Since references must be initialized, wild references like wild pointers does not exist. It is still possible to have references that don't refer to a valid location (dangling reference)
- **References are E*asier to use:*** References don't need a dereferencing operator to access the value. Also, members of an object reference can be accessed with dot operator (‘.’), unlike pointers where arrow operator (->)

Function Overloading

- Function overloading is a feature in C++ where two or more functions can have the same name but different function signatures.
- It is a type of polymorphism.
- The advantage is that the user needs less function names to remember.
- **Function signature:** A function's **signature** includes the function's name and the number, order and type of its formal parameters.
- Two overloaded functions must not have the same signature.
- The return value is **not** part of a function's signature.
- These two functions have the same signature:

*

Function Overloading Example…

```
#include <iostream>
using namespace std;
void print(int i)
{
  cout << " Here is int " << i << endl;
}
void print(double  d)
{
  cout << " Here is double " << d << endl;
}
void print(char *c)
{
  cout << " Here is char* " << c << endl;
}
```

```
int main()
{
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

overloading

The compiler works through the following checklist and if it still can't reach a decision, it issues an error:

1. Gather all the functions in the current scope that have the same name as the function called.
2. Exclude those functions that don't have the right number of parameters to match the arguments in the call.
3. If no function matches, the compiler reports an error.
4. If there is more than one match, select the 'best match'.
5. If there is no clear winner of the best matches, the compiler reports an error - *ambiguous function call.*

Rules for resolving function overloading

- **Best match :** For the best match, the compiler works on a rating system for the match of actual parameters and formal parameters, in a decreasing order of goodness of match:

1. An exact match, e.g. actual parameter is a double and formal parameter is a double.

Example 1:

```
void f(int y[ ]); // call this f1
void f(int* z); // call this f2

int x[ ] = {1, 2, 3, 4};
 f(x); //function call
```

Explanation: // Both f1 and f2 are exact matches, so the call is ambiguous.

Example2:

```
void fun(const char s[]);
void fun(const char*);
fun("abc");
```

Explanation: // Both function has exact matches, so the call is ambiguous.

*

Rules for resolving function overloading

2. Type promotion:

- A bool, char, unsigned char, short or unsigned short can be promoted to an int.
- Example 1: void f(int); can be a match for f('a');
- Example 2: void f(int); can be a match for f(FALSE); In bool FALSE counts as 0, TRUE as 1.
- A float can be promoted to a double. For example void f(double); can be a match for f(5.5F);

overloading

3. A standard type conversion: All the following are described as "standard conversions":

- conversions between integral types (bool, char , signed char , unsigned char. short int , unsigned short, int , and unsigned int, log int, long long int) apart from the ones counted as promotions.
- conversions between floating types: double, float and long double, except for float to double which counts as a promotion.
- conversions between floating and integral types
- conversions of integral, floating, and pointer types to bool.
- conversion of an integer zero to the NULL pointer.
- **All of the standard conversions are treated as equivalent for scoring purposes. No standard conversion is considered better than any of the**

*

overloading

Example:

```
struct Employee; // defined somewhere else
void print(float value);
void print(Employee value);
print('a'); // 'a' converted to match print(float)
```

- In this case, because there is no print(char) (exact match), and no print(int) (promotion match), the ‘a’ is converted to a float and matched with print(float).

4. A constructor or user-defined type conversion

overloading

- **Matching for functions with multiple arguments**
- If there are multiple arguments, C++ applies the matching rules to each argument.
- The function chosen is the one for which at least one argument matching better than all the other functions.
- In other words, the function chosen must provide a better match than all the other candidate functions for at least one parameter.
- In the case that such a function is found, it is clearly and unambiguously the best choice. If no such function can be found, the call will be considered ambiguous (or a non-match).

overloading

```
#include <iostream>
void fcn(char c, int x){
std::cout << 'a';
}
void fcn(char c, double x){
std::cout << 'b';
}
void fcn(char c, float x){
std::cout << 'c';
}
int main(){
fcn('x', 4);
}
```

In the above program, all functions match the first argument exactly. However, the top function matches the second parameter exactly, whereas the other functions require a conversion. Therefore, the top function (the one that prints

More Examples

```
void print(char *value);
void print(int value);
print(0);
```

Although 0 could technically match print(char*) (as a null pointer), it exactly matches print(int) (matching char* would require an implicit conversion). Thus print(int) is the best match available.

****

```
void print(char *value);
void print(int value);
print('a');
```

// promoted to match print(int) In this case, because there is no print(char), the char 'a' is promoted to an integer, which then matches print(int).

*

Predict the output?

```
#include<iostream>
using namespace std;
void print(unsigned int value)
{
	cout<<"UI"<<endl;
}
void print(float value)
{
	cout<<"float"<<endl;
}
int main()
{
//print('a');    // Ambiguity
//print(0);      // Ambiguity
//print(3.14159);    // Ambiguity
return 0;
}
```


*

Output?

```
#include<iostream>
using namespace std;
void test(float s,float t)        // standard conversion:  double to float
{
    cout << "Function with float called ";
}
void test(int s, int t)           // standard conversion:  double to int
{
    cout << "Function with int called ";
}
int main()
{
    test(3.5, 5.6);          // Ambiguity
    return 0;
}
```


*

Output?

```
#include<iostream>
using namespace std;
void f(double )        // standard conversion:  int to double
 {
   cout << "Function with double called "
}
void f(float )         // standard conversion:  int to float
 {
 cout << "Function with float called "
 }
int main()
{
     f(42);       // Ambiguity
    return 0;
}
```


*

Default Arguments

- A default argument is a value provided in function declaration that is automatically assigned by the compiler if caller of the function doesn't provide a value for the argument with default value.

```
#include<iostream>
using namespace std;
// sum function can be called by passing 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)

{
    return (x + y + z + w);
}

int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```


*

Common mistakes when using Default argument

1. It is not possible to keep a non-defaulted argument in between two default arguments:

E.g. void add(int a, int b = 3, int c, int d = 4);
The above function will not compile. In this case, c should also be assigned a default value.

2. All the default arguments should be the trailing arguments:

E.g. void add(int a, int b = 3, int c, int d);
The above function will not compile, we must provide default values for each argument after b. In this case, c and d should also be assigned default values

Inline Functions

- To save execution time in short functions, you may elect to put the code in the function body directly inline with the code in the calling program.
- That is, each time there's a function call in the source file, the actual code from the function is inserted, instead of a jump to the function.
- Functions that are very short, say one or two statements, are candidates to be inlined.

*

Inline function example

```
#include <iostream>
using namespace std;
inline float lbstokg(float pounds) // lbstokg() converts pounds to kilograms
{
	return 0.453592 * pounds;
}
int main()
{
	float lbs;
	cout << "\nEnter your weight in pounds: ";
	cin >> lbs;
	cout << "Your weight in kilograms is " << lbstokg(lbs)<< endl;
	return 0;
}
```

Note: Inline keyword is just a *request to the compiler.* Sometimes the compiler will ignore the request and compile the function as a normal function.

*

Output?

```
#define MUL(a, b)  a*b
int main()
{
   printf("%d", MUL(2+3, 3+5));
    return 0;
}
```

// The macro is expended as 2 + 3 * 3 + 5,
not as 5*8
// Output: 16`

*

type cast or Cast in c++

- Sometimes a programmer needs to convert a value from one type to another.
- **Syntax:** aCharVar = static_cast<char>(anIntVar);

```
#include <iostream>
using namespace std;
int main()
{
        int intVar = 1500000000;                    //1,500,000,000
        intVar = (intVar * 10) / 10;                //result too large
        cout << "intVar = " << intVar << endl;      //wrong answer
        intVar = 1500000000;
        intVar = (static_cast<double>(intVar) * 10) / 10;     //cast to double
        cout << "intVar = " << intVar << endl;              //right answer
        return 0;
}
```

intVar = 211509811
intVar = 1500000000

type cast or Cast in c++

- Advantages of static_cast over C-style cast

1. static_cast<>() gives you a compile time checking ability, C-Style cast doesn't.

For example:

```
char c = 10;       // 1 byte
int *p = (int*)&c; // 4 bytes
*p=5; // run-time error
int *q = static_cast<int*>(&c); // compile-time error
```

2. static_cast<>() can be spotted easily anywhere inside a C++ source code; in contrast, C_Style cast is harder to spot.

*