Introduction to C++

Getting Started with C++

- C++ is derived from the C language. Strictly speaking, it is a superset of C.
- C++ was originally called "C with classes."
- // **FIRST.CPP C++ prgramme**

```
#include <iostream>
#include<cstdio>
using namespace std;
int main()
{
     cout << "Every age has a language of its own\n";
   //std::cout << "Every age has a language of its own\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 file are required?**
- IOSTREAM is an example of a *header file (sometimes called an include 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 include files.

using namespace std

- Namespaces allow us to group named entities into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.
- Namespace is a feature added in C++ and not present in C.
- A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.

using namespace std;

- A C++ program can be divided into different *namespaces.*
- *A namespace is a part of the program* in which certain names are recognized; outside of the namespace they're unknown.
- The directive using namespace std; says that all the program statements that follow within the std namespace.
- Various program components such as **cout** are declared within this namespace.
- If we didn't use the using directive, we would need to add the std name to many program elements. For example, in the This program we'd need to say
- std::cout << "Every age has a language of its own.";
- To avoid adding std:: dozens of times in programs we use the using directive instead.

Cout object

- The identifier cout (pronounced "C out") is actually an *object. It is predefined in C*++.
- The operator << is called the *insertion or put to operator. It directs the contents of the variable* on its right to the object on its left. In FIRST it directs the string constant "Every age has a language of its own\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

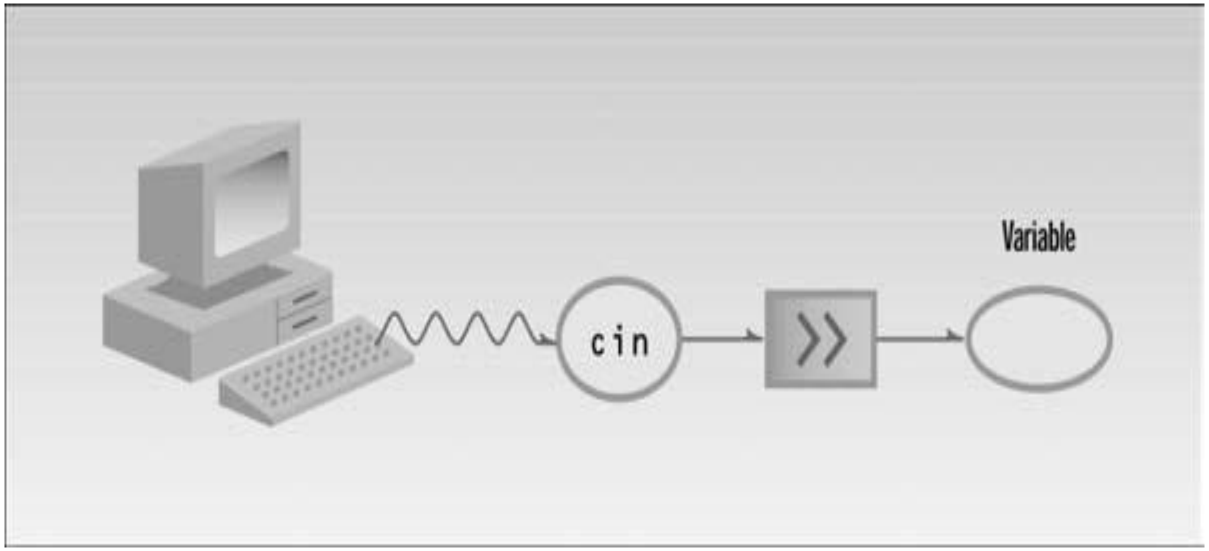
- endl causes a linefeed to be inserted into the stream, so that subsequent text is displayed on the next line. It has the same effect as sending the ‘\n’ character, but is somewhat clearer.

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 or get from 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
```

The const Qualifier Vs #define Directive

- **The const Qualifier**
  - const float PI = 3.14159F; //type const float
  - The keyword const (for constant) 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
  - appearing at the beginning of your program specifies that the identifier PI will be replaced by the 3.14159 throughout the program.

Some more on Const

```
Int main()
int a=10;
int b=20;
const int *c=&a;
*c=20; //error because c is a pointer to an constant int, i.e., c can
not be used to change the value of the variable it is pointing to.
c=&b; // not an error because c is not a constant pointer so it can
point to any variable.
return 0;
}
```

Some more on Const

```
Int main()
int a=10;
int b=20;
int f=30;
int* const g=&f;
g=&a;   // error because g is a constant pointer to int, i.e., it must
always point to the same variable.
*g=50; // not an error because g is pointing to an int and not the
constant int.

return 0;
}
```

Some more on Const

```
Int main()
{
int a=10;
int b=20;
int h=60;
const int * const i=&h;
//i=&a;  //error because i is a constant pointer to constant int.
//*i=70;  //error

return 0;
}
```

Some more on #define

```
#define ghi (2+3*5)
Int main()
{
std::cout<< ghi;
return 0;
}
Output: 17
```

Some more on #define

```
#define jkl (m)   m*m
Int main()
{
Int p=9;
std::cout<< jkl(p);
return 0;
}
Output: 81
```

Some more on #define

```
#define jkl (m)   m*m
Int main()
{
Int p=10;
std::cout<< jkl(p+1);
return 0;
}
```

Output: 21 (not 121)

Explanation: p+1*p+1=10+1*10+1=21

Some more on #define

```
#define jkl (m)   m*m
Int main()
{
Int p=10;
std::cout<< jkl(++p);
return 0;
}
```

Output: 144 (not 121)

Explanation: ++p*++p=12*12=144

Advantages of Const over #define

- #define is not visible to compiler but const variables are visible.
- #define is globally defined but const variables can be used locally or globally.
- Evaluated as many times as replaced but const are evaluated only on initialization.

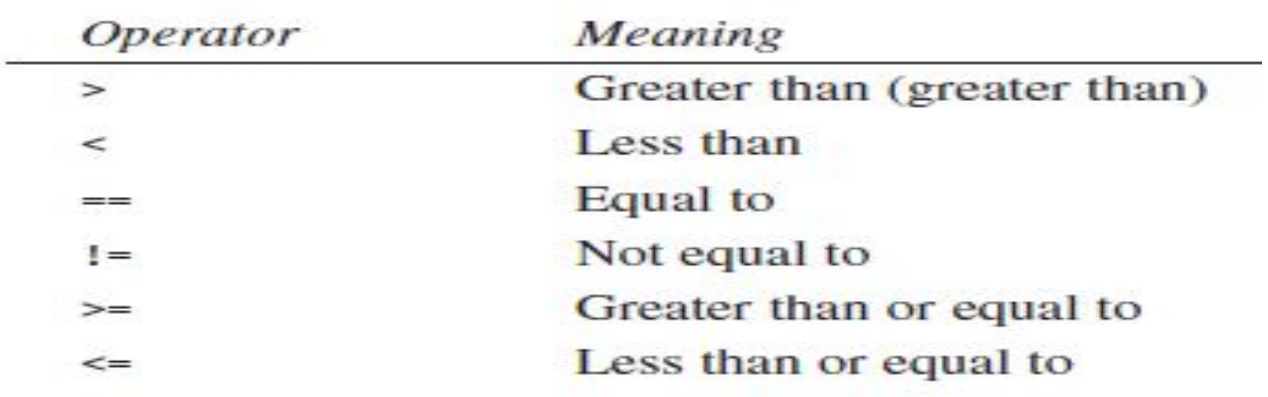
Operator and Extraction Operator

- << 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.

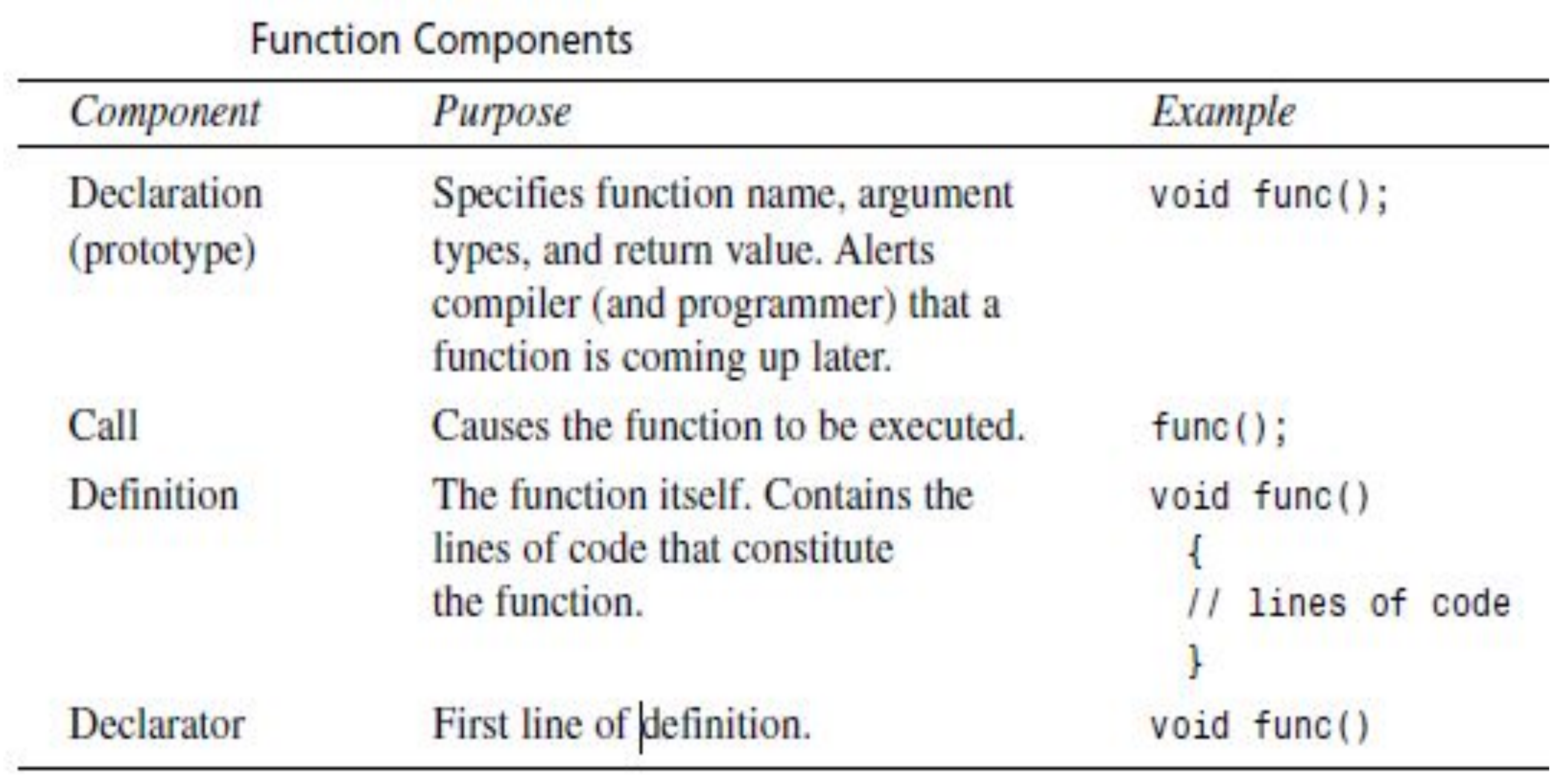
Problem

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

```
#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;
}
```

Functions in C++

- A function groups a number of program statements into a unit and gives it a name.
- This unit 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()`

Names in the Declaration

- The names of the variables in the declaration are optional.
- For example, suppose you were using a function that displayed 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 Variable

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

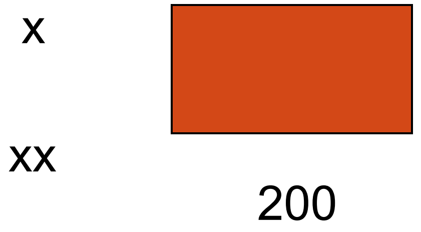
```
int x;       // x is a variable
  int& alias_x=x;  // alias_x is a reference
```

- To create a reference we simply put ‘&’ sign and equate it to an existing variable of same data type

Reference Variable

```
#include <iostream.h>
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
  cout<<endl<<&x<<&xx; //both are 200
  return 0;
}
```

Location is same for both



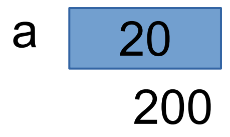
More on Reference Variables

```
Int main()
{
Int a=20;
cout<<a<<&a;
fun(a,a);}
Void fun(int &b, int c)
{
cout<<b<<&b;
cout<<c<<&c;}
```

Output: 20, 200

20, 200

20, 300



More on Reference Variables

```
Int main()
{
Int a=20, b=15;
cout<<a<<b;
swap(&a,&b);
cout<<a<<b;
}
Void swap(int *x, int *y)
{int t;
t=*x;*x=*y;*y=t;}
```

Output: 20, 15
15, 20

More on Reference Variables

```
Int main()
{
Int a=20, b=15;
cout<<a<<b;
swap(a,b);
cout<<a<<b;
}
Void swap(int &x, int &y)
{int t;
t=x;x=y;y=t;}
```

Output: 20, 15
15, 20

Call by Value vs Call by Reference

- Call by reference makes code cleaner, easy, and reliable.
- It saves efforts while passing a big structure.
- Any accidental change in the formal parameter will spoil actual parameter in case of call by reference.
- **Solution:**

```
Main()
{ int a=10, b;
b=fun(a);
cout<<a<<b;}
int fun(const int &x)
{ ++x; return x;}
```

- Output: Error as x cannot be auto incremented.

Marking Scheme

OS

End Sem: 37.5 Marks

Mid Sem: 10 to 15 Marks

Project: 10 to 15 Marks

Lab exam: 10 to 15 Marks

Assignment+ Quiz+ Class Interactions: Remaining

OOPs

End Sem: 37.5 Marks

Mid Sem: 10 to 15 Marks

Project: 10 to 15 Marks

Lab exam: 10 to 20 Marks

Assignment+ Quiz+ Class Interactions: Remaining

Reference Variables Error

- Int &c is an error // A variable must be assigned
- int &c=5 is an error // Constant can not be assigned
- Const int &c=5 is not an error //As LHS became const, so RHS can be const.
- int &c=a+b is an error //Expressions are also const at the end
- Const int &c=a+b is not an error

Return by value

```
Int fun (int &x)
{
cout<<x<<&x;
return x; }
main()
{ int a=10;       //&a=200
cout<<a<<&a;
const int &b=fun(a);// if const is not used then it will give error.
cout<<b<<&b;}
```

- Output: 10 200

```
10 200
10 300 //fun() is returning a copy of x and not actual x.
```

Return by reference

```
Int& fun (int &x)
{
cout<<x<<&x;
return x; }
main()
{ int a=10;      //&a=200
cout<<a<<&a;
const int &b=fun(a);// if const is not used then it will give error.
cout<<b<<&b;}
```

- Output: 10 200

```
10 200
10 200 //fun() is returning actual x.
```

- Good if returning large structures

Return by reference

```
Int& fun (int &x)
{
return x; }
main()
{ int a=10, b;      //&a=200
b=fun(a);
cout<<a<<b;
fun(a)=3;
cout<<a;}
```

- Output: 10 10
  3
- (x>3?y:z)=9;

Return by reference

```
Int& fun (int &x)
{
Int y=20;
return y; }
main()
{ int a=10, b;      //&a=200
b=fun(a);
cout<<a<<b;
fun(a)=3;
cout<<a;}
```

- Segmentation fault

Comparison between Reference and Pointers

- **Similarities:**

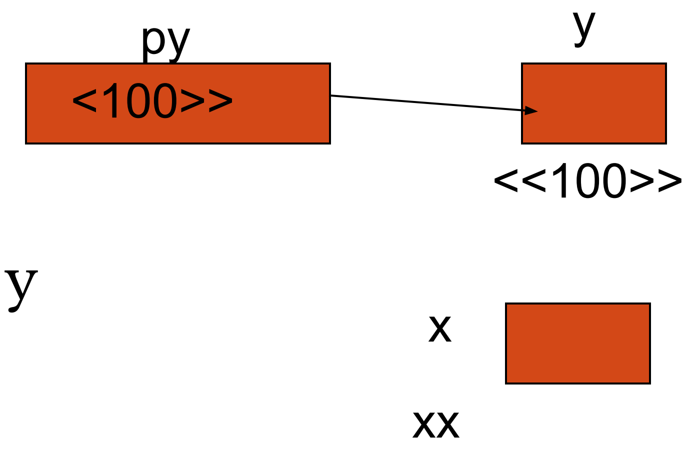
1. Reference and pointers both can be passed to and return from a function
2. An identifier can have any number of references or pointers

Comparison…

- **Differences:**

1. A reference is a logical alternative name for a variable. It does not occupy any space 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
```



Comparison…

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

```
int x;
int& xx;
xx=x;   // Error: not possible
//////////////////////////////////////
int y;
int* py;
py=&y;  // No Error: used frequently
```

Comparison…

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; // ERROR: not possible
//////////////////////////////////
int x,y;
int * pxy=&x;    //p xy is a pointer to x
pxy=&y;          // NO ERROR: used frequently
```

Comparison…

4. There is no concept of NULL reference. There exist NULL pointers
5. Null checking is not required for reference which makes the code easier and faster.
6. Independent manipulation is possible in pointers but any manipulation in reference variables boils down to manipulation of actual variable.

Rules for default parameters

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.

Add(1,2) // Here, 2 will be assigned to b or c not clear

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.

Void add(int a, int b=3, int c=4, int d=5) is correct.

Rules for default parameters

3. Default parameters can not be re-defined.

void gun(int);

void gun(int); // this will not give an error.

void fun(int, double=0.5, char='a');

void fun(int, double=1.5, char='a'); // case of default parameter redefining. So, it will give error.

4. void fun(int, double=0.5, char='a');

Main()

{Int i=5; double d=1.2; char c='b';

fun(i);

}

Void fun(int a, double b=0.5, char d='a') //it will give error. No need to define the

{ cout<<a<<b<<d<<endl;}   // default parameters in the function body if   // once defined in the prototype.

Rules for default parameters

```
5. void fun(int, double=0.5, char='a');
Main()
{Int i=5; double d=1.2; char c=’b’;
fun(i);
fun(i,d);
fun(i, d, c);
Fun(); // It will give error as first parameter is not defaulted.
}
Void fun( int a, double b, char d)
{ cout<<a<<b<<d<<endl;}
```

Function Overloading in C is not there

```
#include <iostream>
using namespace std;
void print_int(int i)
{
 cout << " Here is int " << i << endl;
}
void print_pointer(int *d)
{
 cout << " Here is pointer " << *d << endl;
}
```

```
int main()
{
Int a=10;
 print_int(a);
 print_pointer(&a);
   return 0;
}
```

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:

```
int Divide (int n, int m) ;
double Divide (int p, int q) ;
```

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;
}
```

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

Function Overloading

```
#include <iostream>
using namespace std;
void print(int i, int j)
{
  cout << i << j << endl;
}
void print(double c, double d)
{
  cout << c << d << endl;
}
```

No. of parameters are same but types are different.

Function Overloading

```
#include <iostream>
using namespace std;
void print(int i, int j)
{
  cout << i << j << endl;
}
void print(int c)
{
  cout << c <<endl;
}
```

- No. of parameters are different but types are same.
- Function overloading should be done among same function functionality. However, it is not mandatory.
- Return type must be same to overload a function.

Function Overloading Example...

```
#include <iostream>
using namespace std;
void print(int i, int j)
{
  cout << i << j << endl;
}
void print(double  d)
{
  cout <<  d << endl;
}
```

```
int main()
{
  print(10, 20);
  print(5.6, 3.5);
    return 0;
}
```

First function will be called both time

Function Overloading Example…

```
#include <iostream>
using namespace std;
void print(int i, int j=10)
{
 cout << i << j << endl;
}
void print(int  d)
{
 cout <<  d << endl;
}
```

```
int main()
{
 print(10, 20);
 print(5);
   return 0;
}
```

- Print(5) will give error due to confusion as first print function has default parameter.

Function Overloading Example…

```
#include <iostream>
using namespace std;
void print(int i, float j)
{
  cout << i << j << endl;
}
void print(float c, int d)
{
  cout <<  c<<d << endl;
}
```

```
int main()
{
  print(10, 5.5);
  print(5.5, 10);
Print (10, 10);
Print(5.5, 5.5);
    return 0;
}
```

- Print (10, 10) and Print(5.5, 5.5) will give error.
- In case of single print function, no error.

Name Mangling

- In C++, in case of function overloading, function are resolved by their name and the parameters.
- Int soe_jnu(int x, float y, double z) will be converted to

__Z7soe_jnuifd.

- This conversion can be seen in .s file which is created by the assembler.

Rules for resolving function 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 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);
- A double can be promoted to long double.

Rules for resolving function overloading

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

- conversions between integral types, apart from the ones counted as promotions. e.g., int to unsigned int. Remember that bool and char are integral types as well as int, short and long.
- conversions between floating types: double, float and long double, except for float to double which counts as a promotion. Double to float
- conversions between floating and integral types. e.g. int to double
- conversions of integral, floating and pointer types to bool (zero or NULL is FALSE, anything else is TRUE)
- conversion of an integer zero to the NULL pointer.
- All of the standard conversions are treated as equivalent for scoring purposes. That means seemingly minor standard conversion, such as int to long, does not count as any "better" than a more drastic one such as double to bool.

Rules for resolving function 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).
- Note that all standard conversions are considered equal. No standard conversion is considered better than any of the others.

4. A constructor or user-defined type conversion

Rules for resolving function overloading

- **Matching for functions with multiple arguments**
- If there are multiple arguments, C++ applies the matching rules to each argument in turn.
- The function chosen is the one for which each argument matches at least as well as all the other functions, with 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, and no worse for all of the other parameters.
- 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).

Rules for resolving function 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 ‘a’) is unambiguously the best match.

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');
//print(0);
//print(3.14159);
return 0;
}
```

Output?

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

Output?

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

Output?

```
Int fun (double); //F1
void fun1 (); //F2
void fun1 (int); //F3
Double fun2(void); //F4
Int fun(char, int); //F5
Void fun1(double, double=3.4); //F6
Void fun2(int, double); //F7
Void fun1(char, char*);//F8
int main()
{
     fun1(5.6);
    return 0;
}
```

Function Overloading and Default Parameters

```
Int f(int a=1, int b=2);
Main()
{
Int x=5, y=6;
f();
f(x);
f(x,y);
}
```

- Default parameters are special case of function overloading.
- Equivalent to creating different signatures.

```
Int f();
Int f(int);
Int f(int,int);
Main()
{
Int x=5, y=6;
f();
f(x);
f(x,y);
}
```

Overloading and Return Type

```
float jnu_soe(float i)
{
cout<<"in float";
return i;
}
double jnu_soe(double j)
{
cout<<"in double";
return j;
}
int main()
 {
float f=1.1, g;
double i=3.3, j;
g=jnu_soe(i);
j=jnu_soe(i);
g=jnu_soe(f);
j=jnu_soe(f);
}
```

- Return types are not considered in overload resolution.
- Function call should be context-independent.
- If the return type were taken into account, it would no longer be possible to look at a call of jnu_soe() in isolation and determine which function was called.

Output?

```
int fun(int &x, int c)
{
c=c-1;
if(c==0)
Return 1;
x=x+1;
Return (fun(x,c)*x);
}
 int main()
{
    int y=5, z;
    z=fun(y,y);
    cout<<z;
}
```

Output?

```
Int power(int x, int y)
{
cout<<x<<y;
}
Double power(double x, double y)
{
cout<<x<<y;
}
int main()
{
     power(4, 5.6);
    return 0;
}
```

Output?

```
int power(int x, int y)
{
cout<<"int";
cout<<x;
}
double power(int x, float y)
{
cout<<x;
}

int main()
{
power(4, 'a');
}
```

Output?

```
int power(int x, int y)
{
cout<<"int";
cout<<x;
}
double power(int x, float y)
{
cout<<x;
}

int main()
{
power(4, 4.5);
}
```

Output?

```
int power(int x, int y)
{
cout<<"int";
cout<<x;
}
double power(double x, float y)
{
cout<<x;
}

int main()
{
power(4, 4.5);
}
```

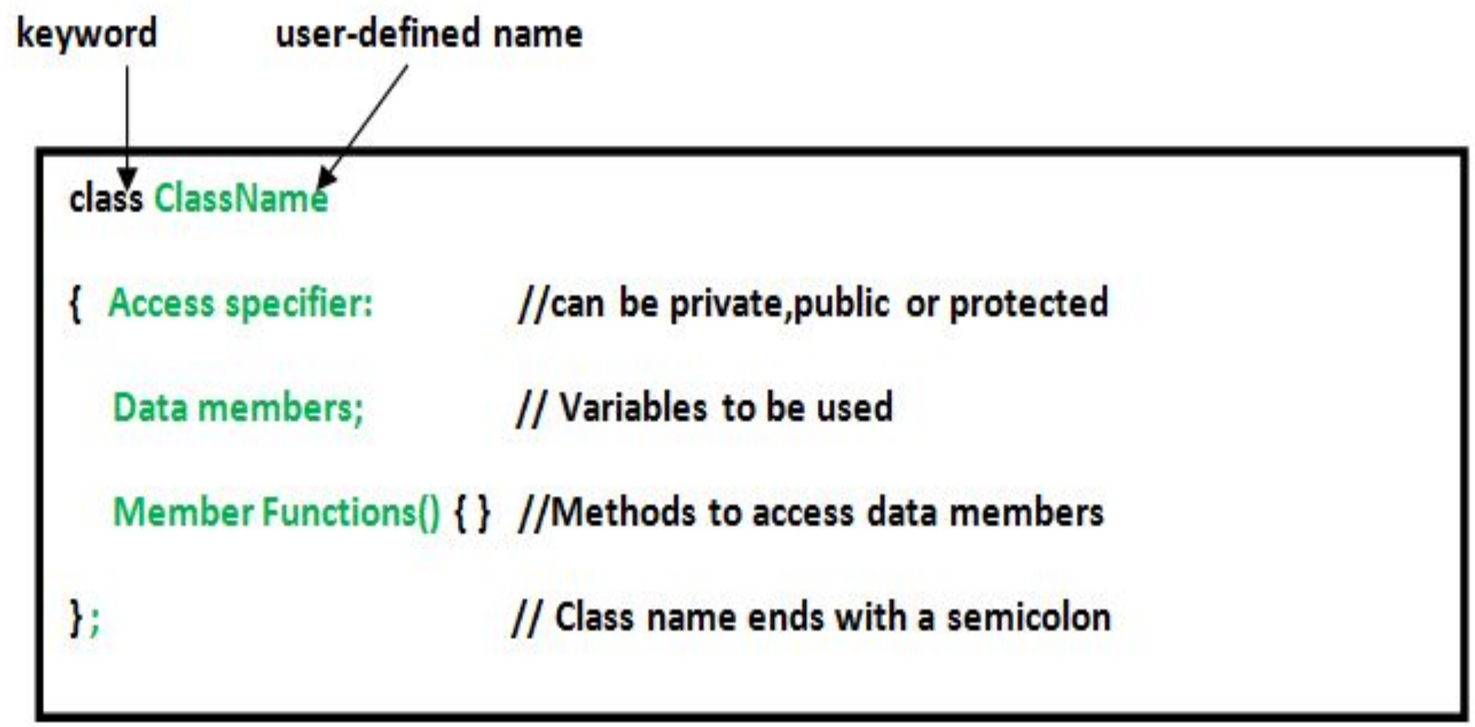
Classes and Objects C++

CLASS & OBJECT

- **Class:** It is the building block that makes C++ as Object Oriented programming.
- **It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.**
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In a class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

Note: The object occupies space in memory during runtime but the class does not.

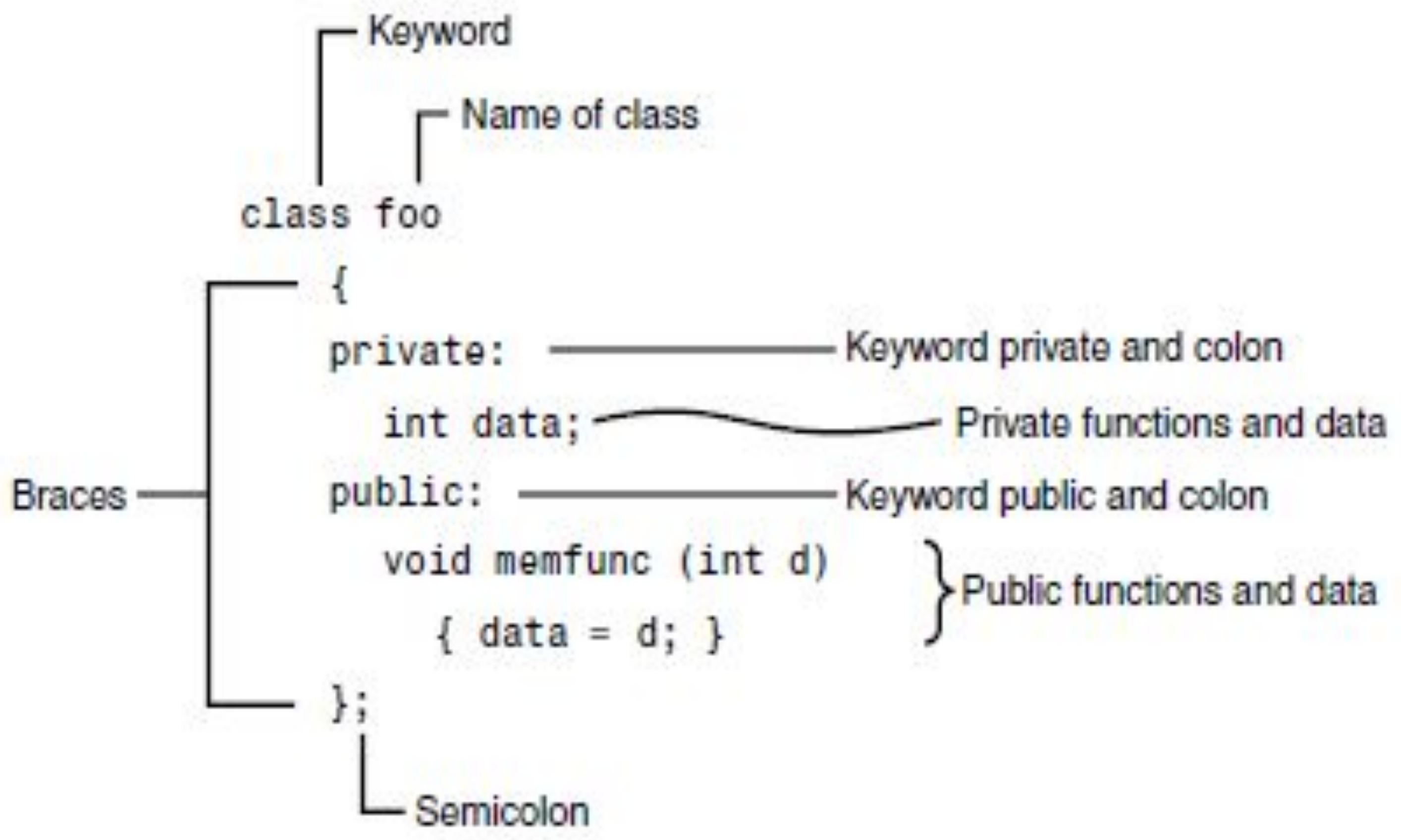
Defining Class and Declaring Objects



```
class ClassName

{  Access specifier:          //can be private,public or protected

   Data members;              // Variables to be used

   Member Functions() { }     //Methods to access data members

};                            // Class name ends with a semicolon
```

- **Declaring Objects:** During class defintion, only the specification for the object is defined; no memory is allocated. To use the data and access functions defined in the class, we need to create objects.
- **Syntax:** **ClassName ObjectName;**

Example of Class


```
class foo
{
private:
   int data;
public:
   void memfunc (int d)
     { data = d; }
};
```

Keyword
Name of class
Braces
Keyword private and colon
Private functions and data
Keyword public and colon
Public functions and data
Semicolon

Fig:Syntax of a class definition.

Defining the Class (cont..)

Declaring the class **smallobj** which contains one data item and two member functions:

```
class smallobj                     //define a class
{
private:
    int somedata;                  //class data or data member
public:
    void setdata(int d) //member function to set data
    {
        somedata = d;
    }
    void showdata()            //member function to display data
    {
        cout << “\nData is “ << somedata;
    }
};
```

private and public (cont…)

- The data member ***somedata*** follows the keyword private, so it can be accessed from within the class, but not from outside.
- setdata() and showdata() follow the keyword public, they can be accessed from outside the class.
- **Generally Functions are public and Data is private:**
  - The data is hidden so it will be safe from **accidental manipulation**, while the functions that operate on the data are public so they can be accessed from outside the class.
  - However, there is no rule that says data must be private and functions public; in some circumstances you may find you'll need to use private functions and public data.

Look back to ***data hiding***

- *data hiding* means that data is concealed within a class so that it cannot be accessed mistakenly by functions outside the class.
- **The primary mechanism for hiding data is to put it in a class and make it private.**
- Private data or functions can only be accessed from within the class. Public data or functions, on the other hand, are accessible from outside the class.