Introduction to C++ Programming (PLC144)

UNIT – 3

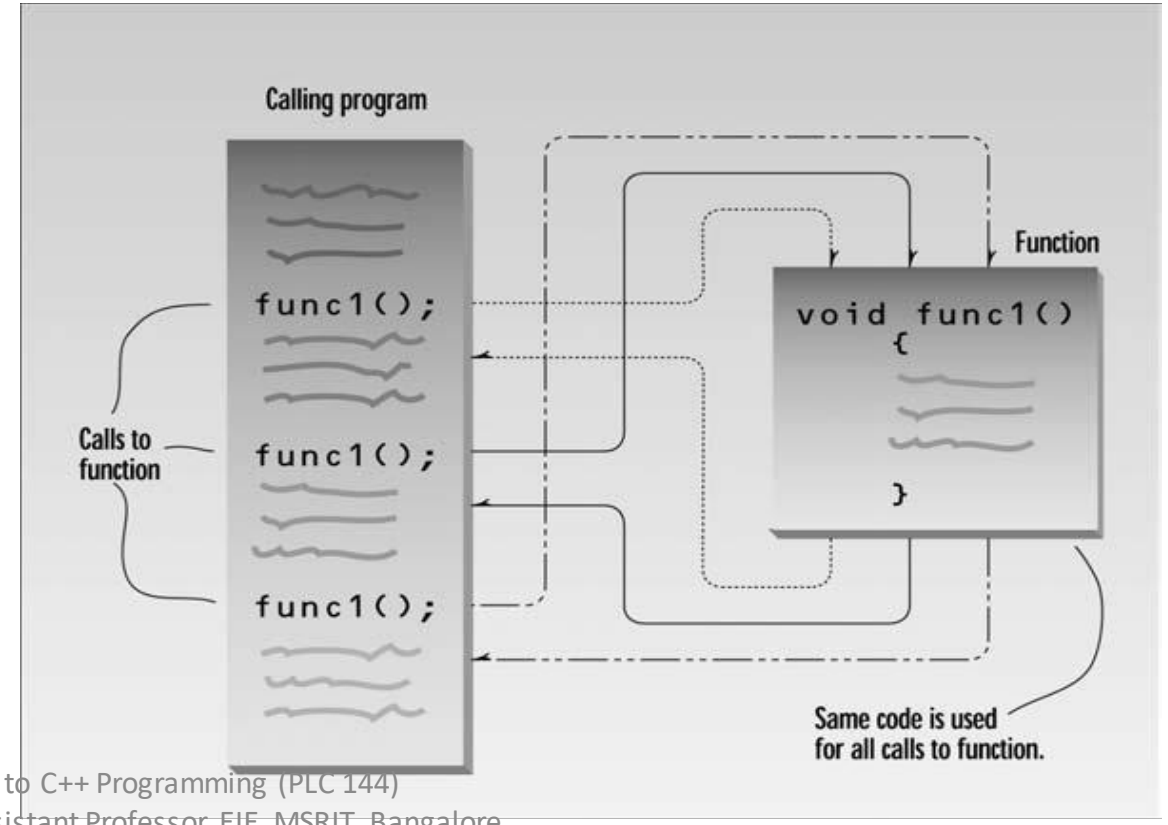
Text Book: Object-Oriented Programming with C++ , E-Balaguruswamy.

Object-Oriented Programming with C++, Robert Lafore.

Programming with ANSI C++, Trivedi Bhushan

Functions:

- A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked from other parts of the program.
- **The Function Declaration**
- **Calling the Function**
- **The Function Definition**



Simple Functions: print a line of 45 asterisks

```
*********************************************
Data type   Range
*********************************************
char         -128 to 127
short        -32,768 to 32,767
int          System dependent
long      -2,147,483,648 to 2,147,483,647
*********************************************
```

The Function Declaration

- you can't use a variable without first telling the compiler what it is, you also can't use a function without telling the compiler about it.

```
void starline();
```

- The declaration tells the compiler that at some later point we plan to present a function called ***starline*.**
- The keyword void specifies that the function has no return value, and the empty parentheses indicate that it takes no arguments.
- Function declarations are also called ***prototypes***, since they provide a model or blueprint for the function.
- They tell the compiler, "a function that looks like this is coming up later in the program.

Calling the Function

- The function is *called* (or *invoked*, or *executed*) three times from main().
- Each of the three calls looks like this:

```
starline();
```

The Function Definition

- The definition contains the actual code for the function
- The definition consists of a line called the *declarator*, followed by the function *body*.
- The function body is composed of the statements that make up the function, delimited by braces

```
void starline() //declarator
{
for(int j=0; j<45; j++) //function body
cout << '*';
cout << endl;
}
```

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()`

```
#include <iostream>
using namespace std;
void starline();
int main()
{
starline();
cout << “Data type Range” << endl;
starline();
cout << “char -128 to 127” << endl
<< “short -32,768 to 32,767” << endl
<< “int System dependent” << endl
<< “long -2,147,483,648 to
2,147,483,647” << endl;
starline();
return 0;
}
```

```
*********************************************
Data type   Range
*********************************************
char         -128 to 127
short        -32,768 to 32,767
int          System dependent
long      -2,147,483,648 to 2,147,483,647
*********************************************
```

```
void starline()
{
for(int j=0; j<45; j++)
        cout << ‘*’;
cout << endl;
}
```

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

void starline()
{
for(int j=0; j<45; j++)
cout << '*';
cout << endl;
}

int main()
{
starline();
cout << "Data type Range" << endl;
starline();
cout << "char -128 to 127" << endl
<< "short -32,768 to 32,767" << endl
<< "int System dependent" << endl
<< "long -2,147,483,648 to 2,147,483,647" << endl;
starline();
return 0;
}
```

Passing Arguments to Functions

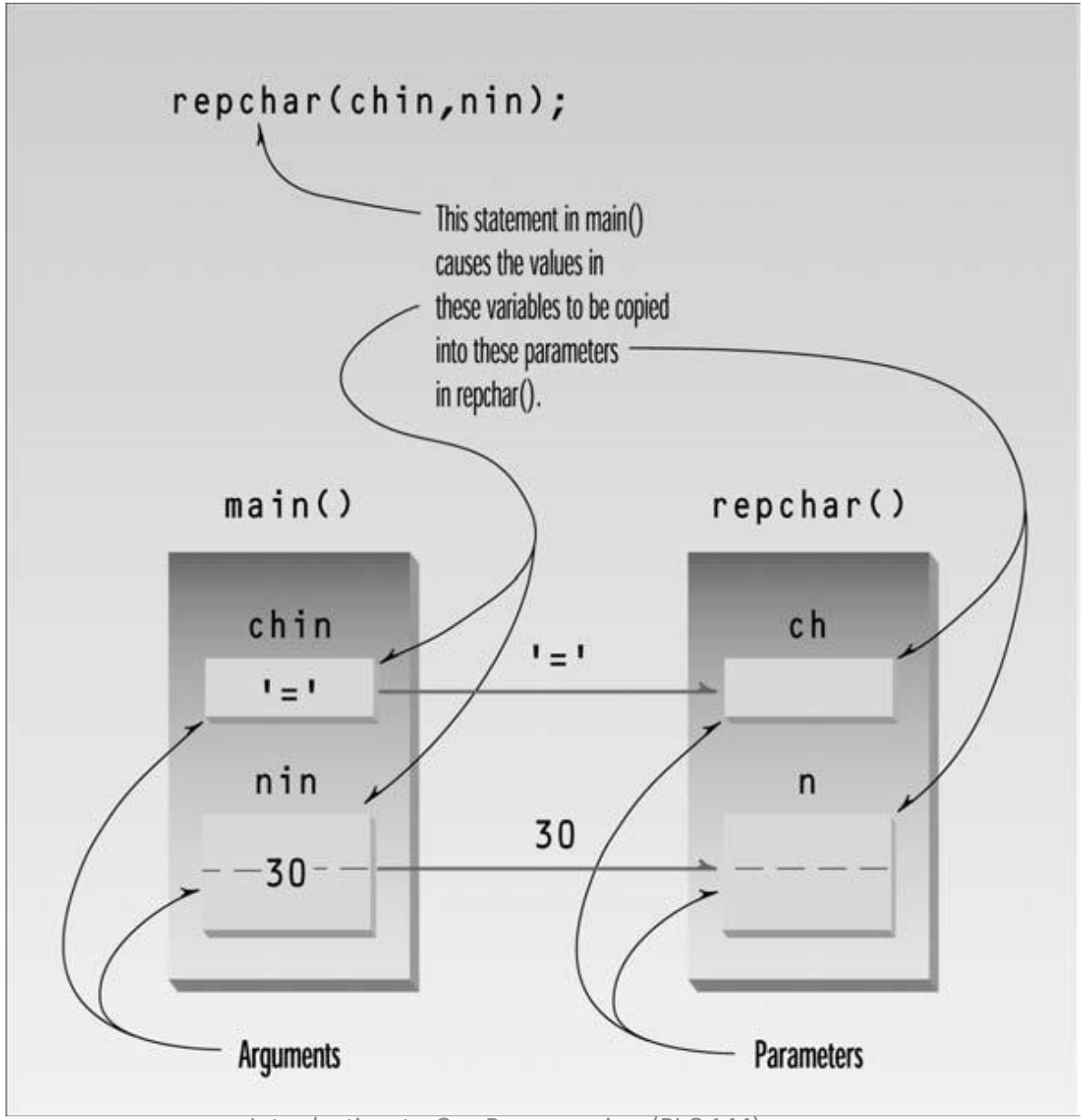
- An *argument* is a piece of data (an int value, for example) passed from a program to the function
- **Passing Constants**

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

void repchar(char, int);

int main()
{
repchar('-', 43);
cout << "Data type Range" << endl;
repchar('=', 23);
cout << "char -128 to 127" << endl
<< "short -32,768 to 32,767" << endl
<< "int System dependent" << endl
<< "double -2,147,483,648 to 2,147,483,647" <<
endl;
repchar('-', 43);
return 0;
}
void repchar(char ch, int n)
{
        for(int j=0; j<n; j++)
                cout << ch;
        cout << endl;
}
```

```
-------------------------------------------
Data type   Range
===========================
char        -128 to 127
short       -32,768 to 32,767
int         System dependent
long         -2,147,483,648 to 2,147,483,647
-------------------------------------------
```

Passing Variables

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

void repchar(char, int);

int main()
{
char chin;
int nin;
cout << "Enter a character: ";
cin >> chin;
cout << "Enter number of times to repeat it: ";
cin >> nin;
repchar(chin, nin);
return 0;
}

void repchar(char ch, int n)
{
      for(int j=0; j<n; j++)
         cout << ch;
      cout << endl;
}
```


repchar(chin,nin);
This statement in main()
causes the values in
these variables to be copied
into these parameters
in repchar().
main()
repchar()
chin
'='
nin
30
'='
30
ch
n
Arguments
Parameters

Passing by Value

- the function creates copies of the arguments passed to it, is called *passing by value*.
- The function gives these new variables the names and data types of the parameters specified in the declarator**: ch of type char and n of type int**. It initializes these parameters to the values passed

```
#include <iostream>

using namespace std;
void Swap(int, int);
int main()
{
   cout << "Happy Programming! Demonstrating Pass By Value" << endl;
   int Var1,Var2;
   cout<<"Enter two numbers to be swapped:"<<endl;
   cin>>Var1>>Var2;
   cout<<"Before swapping:"<<endl;
   cout<<" Var1 is: \t"<<Var1<<endl;
   cout<<" Var2 is: \t"<<Var2<<endl;
   Swap(Var1,Var2);
 cout<<"After swapping:"<<endl;
   cout<<" Var1 is: \t"<<Var1<<endl;
   cout<<" Var2 is: \t"<<Var2<<endl;

   return 0;
}
void Swap(int x, int y)
{
   int temp=x;
   x=y;
   y=temp;
}
```

Reference Arguments

- A *reference* provides an *alias*—a different name—for a variable
- uses for references is in passing arguments to functions.
- *In Pass by Value,* the function cannot access the original variable in
- the calling program, only the copy it created.
- Passing arguments by value is useful when the function does not need to modify the original variable in the calling program.
- Instead of a value being passed to the function, a *reference to* the original variable, in the calling program

```
int var = 10;
int & Refvar = var;
```

8000 (var)	10

var → Original variable

Refvar → Reference Variable

Changes in Reference Variable modifies the Original variable

```
Refvar = 100;
cout<< Refvar<<var<<endl;
```

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

void order(int&, int&);
int main()
{
int n1=99, n2=11;
int n3=22, n4=88;
order(n1, n2);
order(n3, n4);
cout << “n1=” << n1 << endl;
cout << “n2=” << n2 << endl;
cout << “n3=” << n3 << endl;
cout << “n4=” << n4 << endl;
return 0;
}
void order(int& numb1, int& numb2)
{
if(numb1 > numb2)
{
int temp = numb1;
numb1 = numb2;
numb2 = temp;
}
}
```

```
#include <iostream>
using namespace std;
void intfrac(float, float&, float&); //declaration
int main()
{
float number, intpart, fracpart;
do {
cout << "\nEnter a real number: ";
cin >> number;
intfrac(number, intpart, fracpart);
cout << "Integer part is " << intpart
<< ", fraction part is " << fracpart << endl;
} while( number != 0.0 );
return 0;
}

void intfrac(float n, float& intp, float& fracp)
{
long temp = long(n);
intp = float(temp);
fracp = n - intp;
}
```

Overloaded Functions

- Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.
- Function overloading can be considered as an example of polymorphism feature in C++.

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

void repchar();
void repchar(char);
void repchar(char, int);

int main()
{
repchar();
repchar('=');
repchar('+', 30);
return 0;
}
```

```
void repchar()
{
for(int j=0; j<45; j++)
cout << '*';
cout << endl;
}
void repchar(char ch)
{
for(int j=0; j<45; j++)
cout << ch;
cout << endl;
}
void repchar(char ch, int n)
{
for(int j=0; j<n; j++) // loops n times
cout << ch; // prints specified character
cout << endl;
}
```

Function Overloading

Resolving function overloads

can have two or more functions with the same name so long as they differ in their parameter lists.

```
void f(double d1, int i1)
{
...
}
void f(double d1, double d2)
{
...
}
int main( )
{
cout << f(1.0, 2);
}
```

How does the compiler know which version of f() to call? 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 that don't have the right number of parameters to match the arguments in the call. (It has to be careful about parameters with default values; void f(int x, int y = 0) is a candidate for the call f(25);)
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.

Best matching

- In deciding on the best match, the compiler works on a rating system for the way the types passed in the call and the competing parameter lists match up.
- An exact match, e.g. argument is a double and parameter is a double
- A promotion
- A standard type conversion
- A constructor or user-defined type conversion

- **Exact matches**
- An exact match is where the parameter and argument datatypes match exactly.

```
void f(double d1, int i1)
{
...
}
void f(double d1, double d2)
{
...
}
int main( )
{
cout << f(1.0, 2);
}
```

```
void func(bool x)
{
   cout<<"Integer Conversions \t"<<x<<endl;
   cout<<x<<endl;
}

void func(int x)
{
   cout<<“Exact Match  "<<x<<endl;
   cout<<x<<endl;
}
int main()
{
   cout << "Hello world!" << endl;
   int Var=10;
   func(Var);


   return 0;
}
```

```
Hello world!
Exact Match  10
10
```

```
void func(long int x)
{
   cout<<x<<endl;
}
void func(bool x)
{
   cout<<"Integer Conversions \t"<<x<<endl;
   cout<<x<<endl;
}
void func(int x)
{
   cout<<“Exact Match  "<<x<<endl;
   cout<<x<<endl;
}
int main()
{
   cout << "Hello world!" << endl;
   int Var=10;
   func(Var);
   return 0;
}
```

```
Hello world!
Exact Match  10
10
```

Type promotion

The following are described as "promotions":

A `char`, `unsigned char` or `short` can be promoted to an `int`. For example `void f(int);` can be a match for `f('a');`

A `float` can be promoted to a `double`

A `bool` can be promoted to an `int` (FALSE counts as 0, TRUE as 1).

```
void func(int x)
{
    cout<<"promotion"<<x<<endl;
    cout<<x<<endl;
}
int main()
{
    func('a');

    return 0;
}
```

```
void func(int x)
{
   cout<<"Type promotion  "<<x<<endl;
   cout<<x<<endl;
}
void func(long int x)
{
 cout<<“Standard Conversion \t"<<x<<endl;
 cout<<x<<endl;
}
void func(bool x)
{
   cout<<“Standard Conversion \t"<<x<<endl;
   cout<<x<<endl;
}
int main()
{
   cout << "Hello world!" << endl;
   func('a');
   return 0;
}
```

```
Hello world!
Type promotion   97
97
```

```
void func(double x)
{
   cout<<“Type Promotion  "<<x<<endl;
   cout<<x<<endl;
}

void func(long int x)
{
   cout<<“Standard Conversion \t"<<x<<endl;
   cout<<x<<endl;
}
int main()
{
   cout << "Hello world!" << endl;
   float Var=10;
   func(Var);
return 0;
}
```

```
Hello world!
Type Promotion  10
10
```

- **Standard conversions**

All the following are described as "standard conversions":

•conversions between integral types, apart from the ones counted as promotions. 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.

•conversions between floating and integral types

•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.

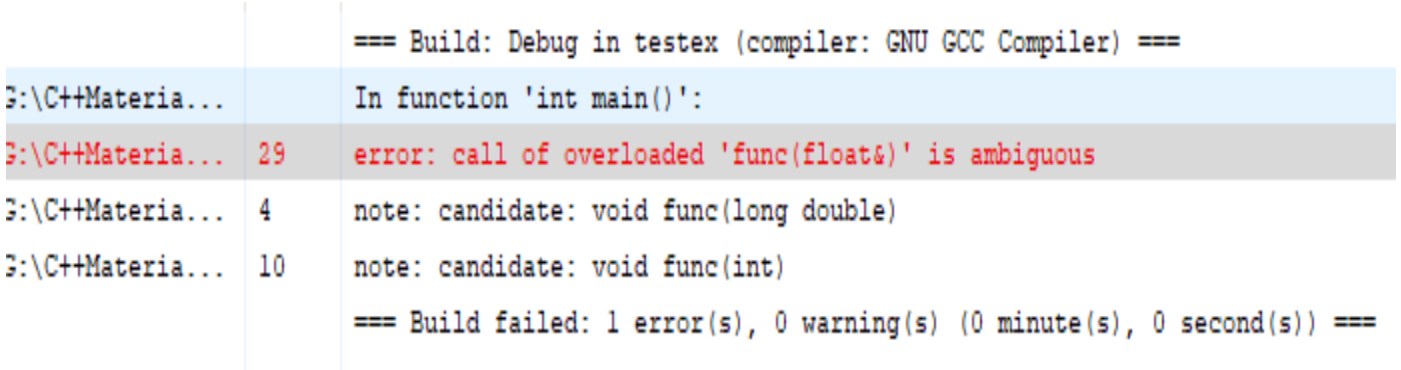
Conversion between int types: Integer to bool Conversions

```
void func(bool x)
{
   cout<<"Integer Conversions \t"<<x<<endl;
   cout<<x<<endl;
}
void func(int x[])
{
   cout<<"Std"<<x<<endl;
   cout<<x<<endl;
}
int main()
{
   cout << "Hello world!" << endl;
   int Var=10;
   func(Var);
   return 0;
}
```

Output:

```
Hello world!
Integer Conversions    1
1
```

```
void func(long double x)
{
   cout<<"Standard Conversion"<<x<<endl;
   cout<<x<<endl;
}

void func( int x)
{
   cout<<"Standard Conversion"<<x<<endl;
   cout<<x<<endl;
}
int main()
{
   cout << "Hello world!" << endl;
   float Var=10;
   func(Var);
 return 0;
}
```



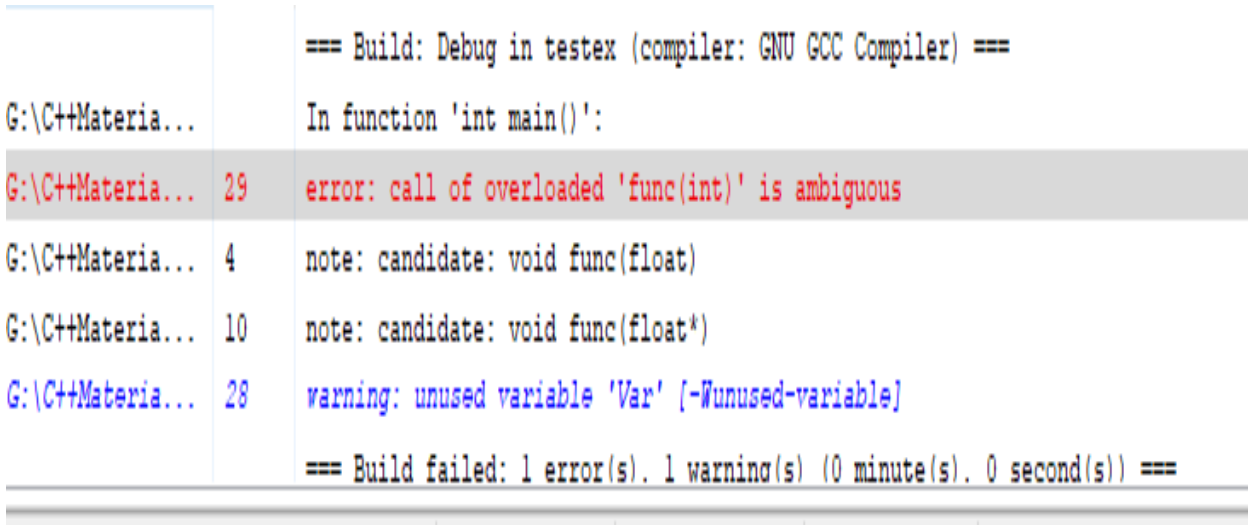
Note: Both are Standard Conversions 1) float to long double 2) float to int

So both has equal weightage and the functions are ambiguous.

```
using namespace std;
void func( float  x)
{
   cout<<"Standard Conversion"<<x<<endl;
   cout<<x<<endl;
}

void func( float * x)
{
   cout<<x<<endl;
}
int main()
{
   cout << "Hello world!" << endl;
   float Var=10;
   func(0);
return 0;
}
```

Note: Both are Standard Conversions 1) int to float 2) 0 to float *

So both has equal weightage and the functions are ambiguous.



```
void func(int x)
{
   cout<<"Type promotion"<<x<<endl;
   cout<<x<<endl;
}

void func(long int x)
{
   cout<<x<<endl;
}
int main()
{
   cout << "Hello world!" << endl;
   float Var=10;
   func(Var);
return 0;
}
```

		=== Build: Debug in testex (compiler: GNU GCC Compiler) ===
G:\C++Materia...		In function 'int main()':
G:\C++Materia...	29	error: call of overloaded 'func(float&)' is ambiguous
G:\C++Materia...	4	note: candidate: void func(int)
G:\C++Materia...	10	note: candidate: void func(long int)
		=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

```
void func( float  x, int y)
{
   cout<<"1st arg Exact Match , 2 nd arg Std Conversion"<<x<<endl;
   cout<<x<<endl;
}

void func( bool x, int y)
{
   cout<<"1st arg std Conv , 2 nd arg Std Conversion"<<x<<endl;
   cout<<x<<endl;
}
int main()
{
   cout << "Hello world!" << endl;
   float Var=10;
   func(Var,12.5);
   return 0;
}
```

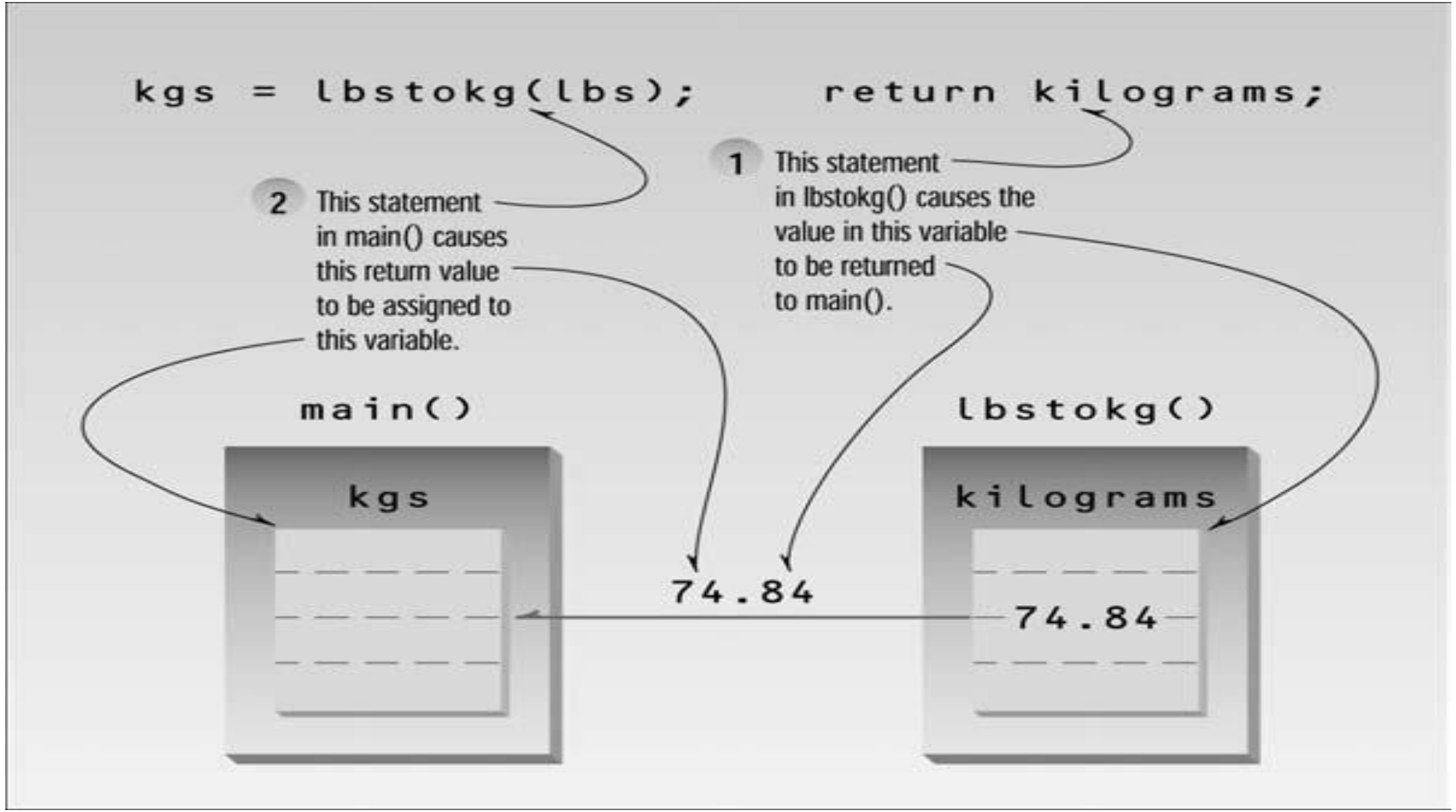
```
Hello world!
1st arg Exact Match , 2 nd arg Std Conversion10
10
```

```
void func( int  x,int y)
{
   cout<<"1st arg Std Conv , 2 nd arg Std Conversion"<<x<<endl;
   cout<<x<<endl;
}

void func( bool x, int y)
{
   cout<<"1st arg std Conv , 2 nd arg Std Conversion"<<x<<endl;
   cout<<x<<endl;
}
int main()
{
   cout << "Hello world!" << endl;
   float Var=10;
   func(Var,12.5f);
 return 0;
}
```

```
                    === Build: Debug in testex (compiler: GNU GCC Compiler) ===
G:\C++Materia...    In function 'int main()':
G:\C++Materia... 30 error: call of overloaded 'func(float&, float)' is ambiguous
G:\C++Materia... 4  note: candidate: void func(int, int)
G:\C++Materia... 10 note: candidate: void func(bool, int)
                    === Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Returning Values from Functions

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

float lbstokg(float pounds)
{
float kilograms = 0.453592 * pounds;
return kilograms;
}
```


kgs = lbstokg(lbs);
return kilograms;
1 This statement in lbstokg() causes the value in this variable to be returned to main().
2 This statement in main() causes this return value to be assigned to this variable.
main()
lbstokg()
kgs
kilograms
74.84
74.84

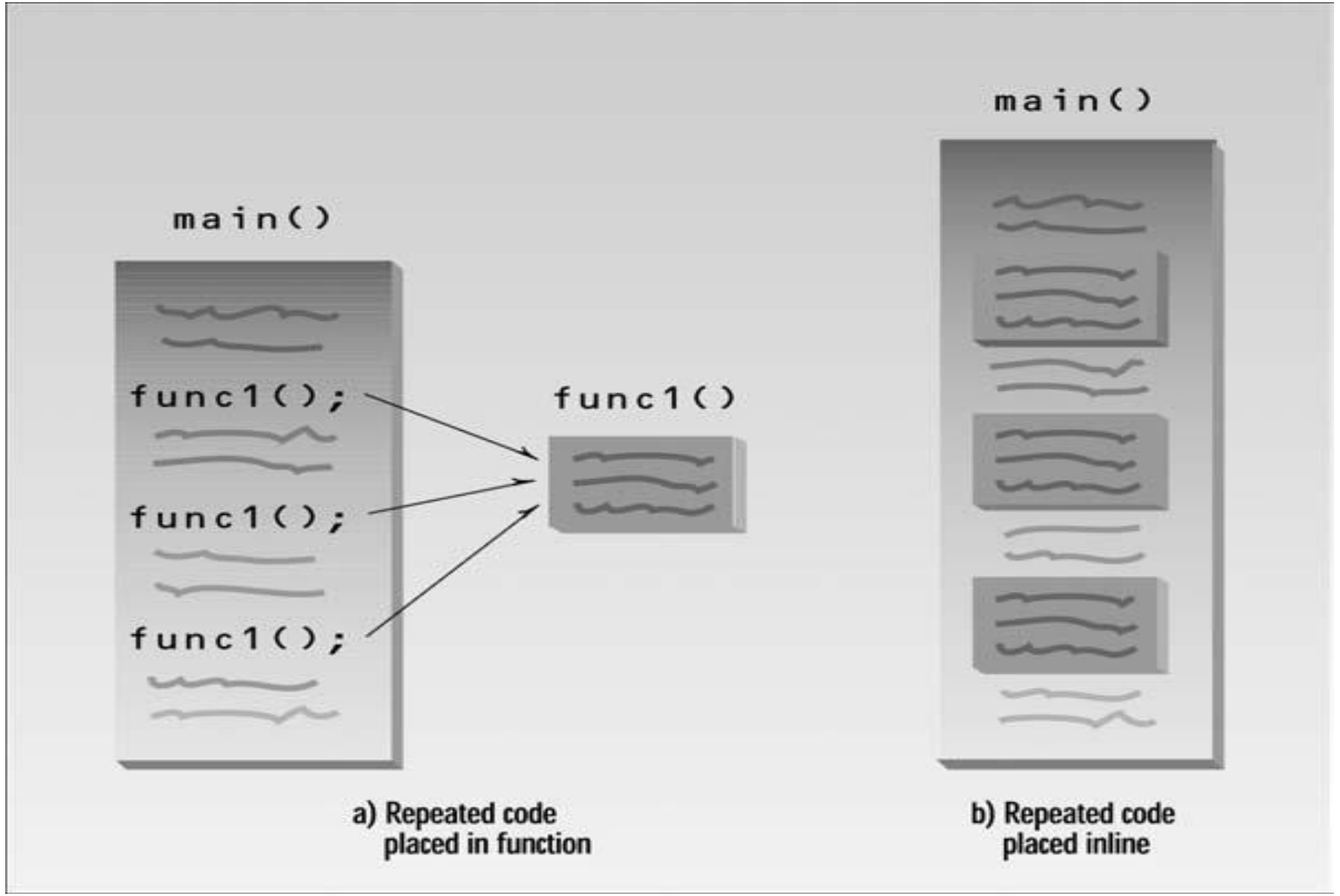
```
#include <iostream>
using namespace std;
float lbstokg(float); //declaration
int main()
{
float lbs;
cout << "\nEnter your weight in pounds: ";
cin >> lbs;
cout << "Your weight in kilograms is " << lbstokg(lbs)
<< endl;
return 0;
}

float lbstokg(float pounds)
{
return 0.453592 * pounds;
}
```

Inline Functions

- For a Function call be an instruction for the jump to the function (actually the assembly-language instruction CALL or something similar), instructions for saving registers, instructions for pushing arguments onto the stack in the calling program and removing them from the stack in the function (if there are arguments), instructions for restoring registers, and an instruction to return to the calling program. The return value (if any) must also be dealt with.
- All these instructions slow down the program.
- 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

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining.

Compiler may not perform inlining in such circumstances like:
1) If a function contains a loop. (for, while, do-while)
2) If a function contains static variables.
3) If a function is recursive.
4) If a function return type is other than void, and the return statement doesn't exist in function body.
5) If a function contains switch or goto statement.


main()
func1();
func1();
func1();
func1()
a) Repeated code placed in function
main()
b) Repeated code placed inline

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

inline float lbstokg(float pounds)
{
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;
}
```

Default Arguments

The function declaration may provide default values for those arguments that are not specified.

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

void repchar(char='*', int=45);

int main()
{
repchar();
repchar('=');
repchar('+', 30);
return 0;
}

void repchar(char ch, int n)
{
for(int j=0; j<n; j++)
cout << ch;
cout << endl;
}
```

Scope and Storage Class

- two features of C++ that are related to the interaction of variables and functions: *scope* and *storage class.*
- The scope of a variable determines which parts of the program can access it, and its storage class determines how long it stays in existence
- Variables with *local* scope are visible only within a block.
- • Variables with *file* scope are visible throughout a file.
- There are two storage classes: automatic and static.
- Variables with storage class *automatic* exist during the lifetime of the function in which
- they’re defined.
- Variables with storage class *static* exist for the lifetime of the program.

Local Variables:

```
void somefunc()
{
int somevar; //local variables
float othervar;
somevar = 10; //OK
othervar = 11; //OK
nextvar = 12; //illegal: not visible in somefunc()
}
```

```
void otherfunc()
{
int nextvar; //local variable
somevar = 20; //illegal: not visible in otherfunc()
othervar = 21; //illegal: not visible in otherfunc()
nextvar = 22; //OK
}
```

Global Variables

```
#include <iostream>
using namespace std;
char ch = 'a'; //global variable ch
void getachar();
void putachar();
int main()
{
while( ch != '\r' ) //main() accesses ch
{
getachar();
putachar();
}
cout << endl;
return 0;
}
void getachar() //getachar() accesses ch
{
ch = getch();
}
void putachar() //putachar() accesses ch
{
cout << ch;
}
```

Static Local Variables

```
#include <iostream>
using namespace std;
float getavg(float);

int main()
{
float data=1, avg;
while( data != 0 )
{
cout << “Enter a number: “;
cin >> data;
avg = getavg(data);
cout << “New average is “ << avg <<
endl;
}
return 0;
}

float getavg(float newdata)
{
static float total = 0; //static variables are initialized
static int count = 0; // only once per program
count++; //increment count
total += newdata; //add new data to total
return total / count; //return the new average
}
```

TABLE 5.2 Storage Types

	Local	*Static Local*	*Global*
Visibility	function	function	file
Lifetime	function	program	program
Initialized value	not initialized	0	0
Storage	stack	heap	heap
Purpose	Variables used by a single function	Same as local, but retains value when function terminates	Variables used by several functions

Returning by Reference

- Besides passing values by reference, you can also return a value by reference
- Why you would Want -One reason is to avoid copying a large object
- to allow you to use a function call on the left side of the equal sign

```
#include <iostream>
using namespace std;
int x;
int& setx();

int main()
{
setx() = 92;
cout << “x=” << x << endl;
return 0;
}
```

```
int& setx()
{
return x;
}
```

1) A function's single most important role is to

a. give a name to a block of code.

b. reduce program size.

c. accept arguments and provide a return value.

d. help organize a program into conceptual units.

2) A function itself is called the function d_____.

3) A one-statement description of a function is referred to as a function d_____ or a p_____.

4) The statements that carry out the work of the function constitute the function _____.

5) Which of the following can legitimately be passed to a function?

a. A constant

b. A variable

c. A structure

d. A header file

6) Here's a function:

```
int times2(int a)
{
return (a*2);
}
```

Write a main() program that includes everything necessary to call this function.

Refer to the CIRCAREA program in Chapter 2, “C++ Programming Basics.” Write a function called circarea() that finds the area of a circle in a similar way. It should take an argument of type float and return an argument of the same type. Write a main() function that gets a radius value from the user, calls circarea(), and displays the result.

- Raising a number n to a power p is the same as multiplying n by itself p times. Write a function called power() that takes a double value for n and an int value for p, and returns the result as a double value. Use a default argument of 2 for p, so that if this argument is omitted, the number n will be squared. Write a main() function that gets values from the user to test this function

```
#include <iostream>
 using namespace std;
double power(double, int = 2);
int main()
{
int exp;
double base, result;
cout << "Enter the number: " ;
cin >> base ;
cout << endl;
cout << "Enter the power you want to raise it to: " ;
cin >> exp;
cout << endl;
result = power(base, exp);
cout << base << " to the power " << exp << " = " << result <<
endl;
cout << "When the value of exponent is not passed, " << endl;
result = power(base);
cout << "The number is squared: " << result << endl;
return 0;
}
```

```
double power( double x, int y)
{
  int i;
  double ans = 1; for(i =0; i < y; i++)
  {
    ans = ans * x;
  }
  return ans;
}
```

- Write a function called zeroSmaller() that is passed two int arguments by reference and then sets the smaller of the two numbers to 0. Write a main() program to exercise this function.

3 Components : **Defining the Class, Using the Class, Calling Member Functions**

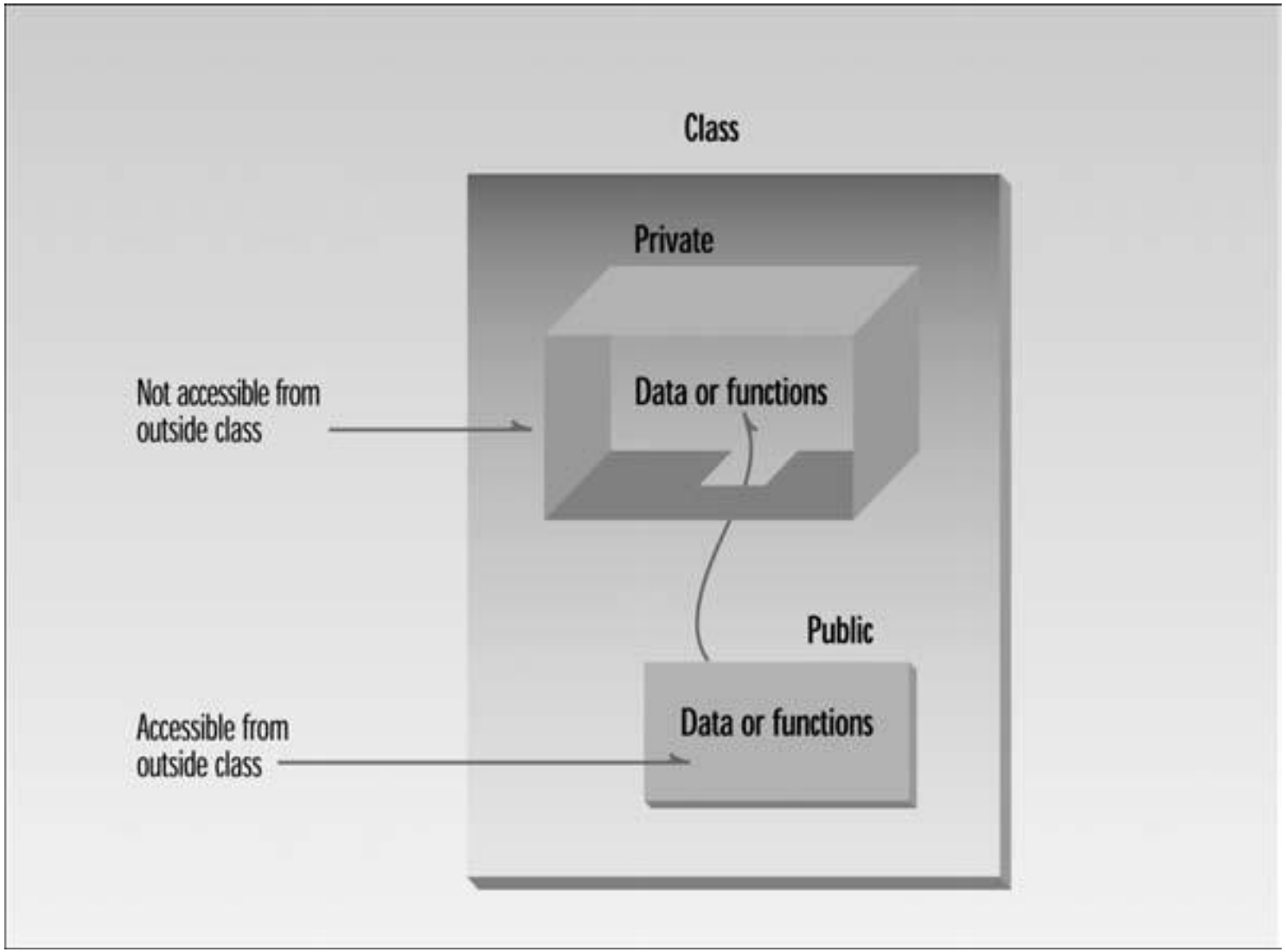
Defining the Class

Access Specifiers: private and public- Concept of *data hiding*

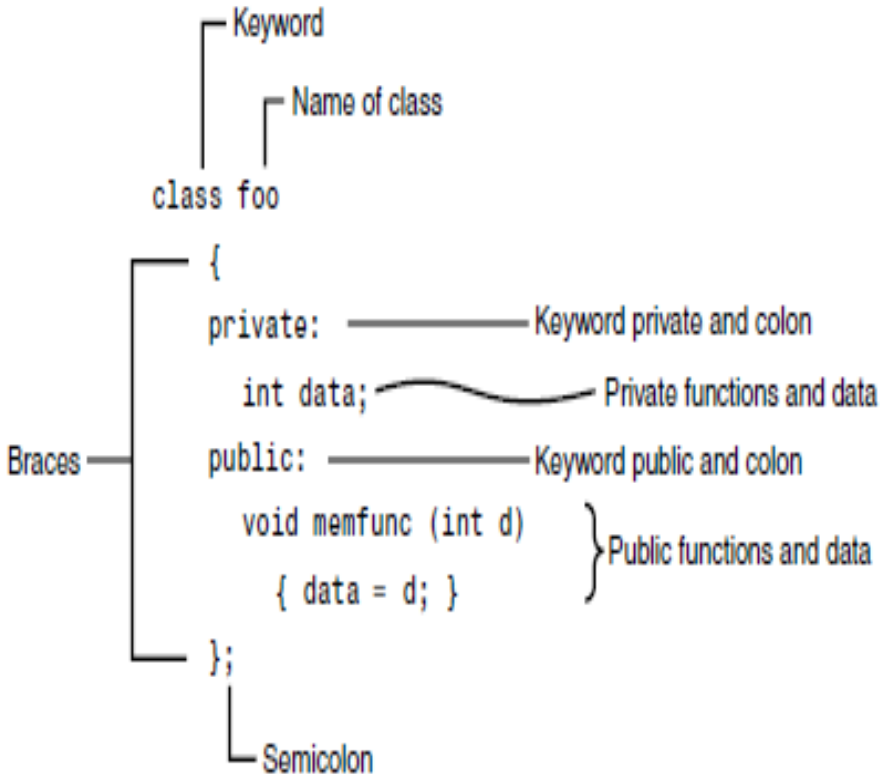
- The primary mechanism for hiding data is to put it in a class and make it private. Private data or functions can only be accessed from within the class.
- Public data or functions, on the other hand, are accessible from outside the class.
- **Class Data:**The data items within a class are called *data members*
- **Member Functions:**
- *Member functions* are functions that are included within a class.

Defining the Class:

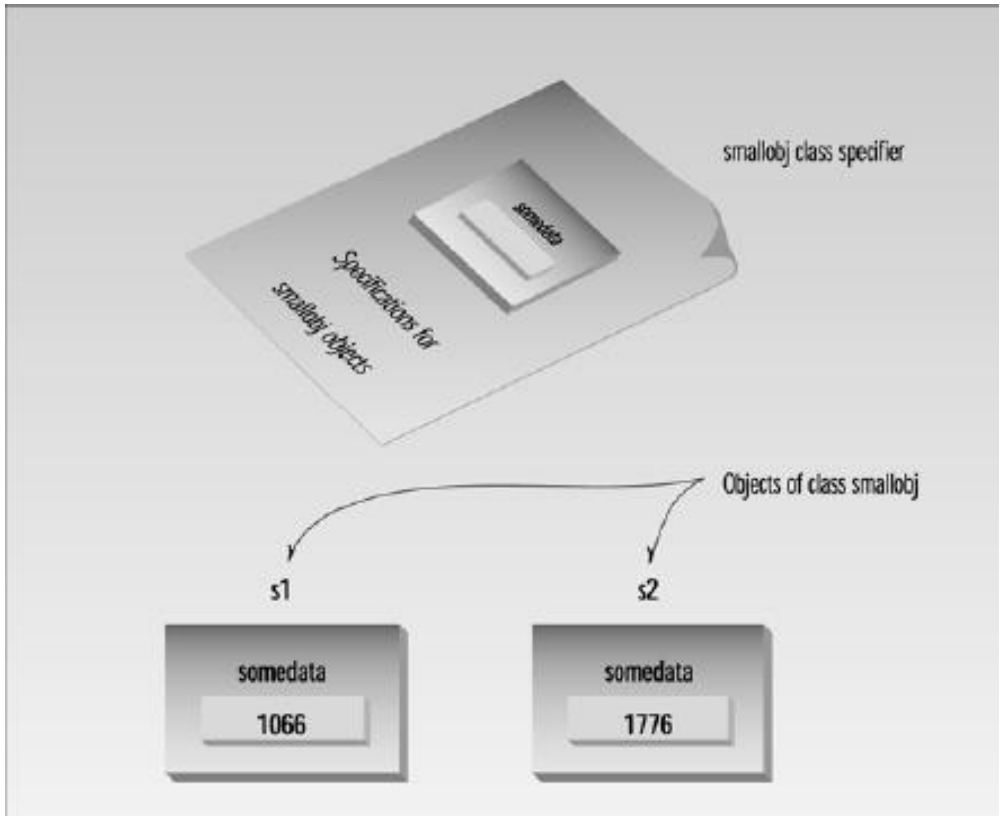
```
class smallobj
{
private:
int somedata;
public:
void setdata(int d)
{
somedata = d;
}
void showdata()
 { cout << "\nData is " << somedata;
}
};
```


Class
Private
Data or functions
Not accessible from
outside class
Public
Data or functions
Accessible from
outside class

- **Functions Are Public, Data Is Private**



- **Using the Class**
  - **Defining Objects**
  - smallobj s1, s2;
- **Calling Member Functions**
  - s1.setdata(1066);
  - s2.setdata(1776);



- **Constructors**
- member functions can be used to give values to the data items in an object.
- Is it possible to an object can initialize itself when it's first created, without requiring a separate call to a member function???
- Automatic initialization is carried out using a special member function called a *constructor*.
- A constructor is a member function that is executed automatically whenever an object is created.
- **Same Name as the Class**

- A constructor is different from normal functions in following ways:
- Constructor has same name as the class itself
- Constructors don’t have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).
- **Constructors** cannot be private.
- A **constructor** cannot be static
- A **constructor** can be overloaded.
- **Constructors** cannot return a value.
- Constructors specified in public access specifier

```
#include <iostream>
using namespace std;
class Counter
{
private:
unsigned int count;
public:
Counter() : count(0) //constructor
{ /*empty body*/ }
void inc_count()
{
count++;
}
int get_count()
{
return count;
}
};
int main()
{
Counter c1, c2;
cout << "\nc1=" << c1.get_count();
cout << "\nc2=" << c2.get_count();
c1.inc_count();
c2.inc_count();
c2.inc_count();
cout << "\nc1=" << c1.get_count();
cout << "\nc2=" << c2.get_count();
cout << endl;
return 0;
}
```

Destructors

function—the constructor—is called automatically when an object is first created.

another function is called automatically when an object is destroyed. Such a function is called a *destructor.*

A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde:

```
class Foo
{
private:
int data;
public:
Foo() : data(0) //constructor (same name as class)
{ }
~Foo() //destructor (same name with tilde)
{ }
};
```

Objects as Function Arguments & Overloaded Constructors

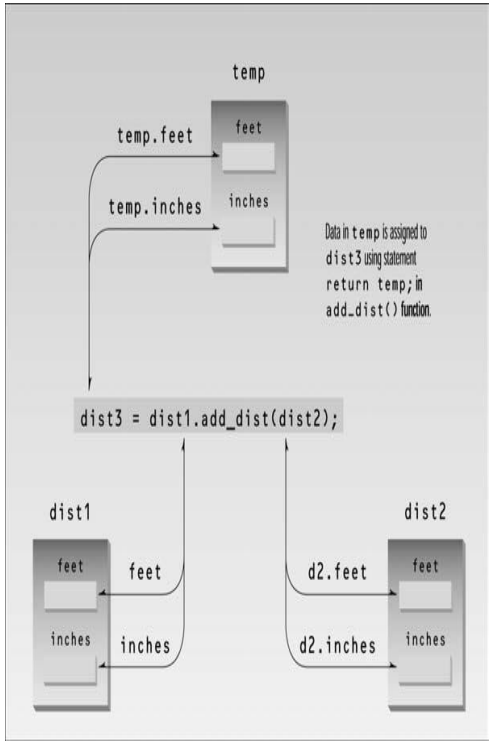
```
class Distance {
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)
{ }
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist()
 {
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist()
{
cout << feet << "\'-" << inches << '\"';
}
void add_dist( Distance, Distance );
};
void Distance::add_dist(Distance d2, Distance d3)
{
inches = d2.inches + d3.inches;
feet = 0;
if(inches >= 12.0)
{
inches -= 12.0;
feet++;
}
feet += d2.feet + d3.feet;
}
int main()
{
Distance dist1, dist3;
Distance dist2(11, 6.25);
dist1.getdist();
dist3.add_dist(dist1, dist2);

cout << "\ndist1 = ";
dist1.showdist();
cout << "\ndist2 = ";
dist2.showdist();
cout << "\ndist3 = ";
dist3.showdist();
cout << endl;
return 0;
}
```

- **The Default Copy Constructor:**
- you can initialize it with *another object of the same type*. Surprisingly, you don't need to create a special constructor for this; one is already built into all classes. It's called the *default copy constructor*. It's a one argument constructor whose argument is an object of the same class as the constructor.

```
int main()
{
Distance dist1(11, 6.25); //two-arg constructor
Distance dist2(dist1); //one-arg constructor
Distance dist3 = dist1; //also one-arg constructor

cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
return 0;
}
```

Returning Objects from Functions

```
#include <iostream>
using namespace std;
class Distance
{
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)
{ }
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist()
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist()
{ cout << feet << "\'-" << inches << '\"'; }

Distance add_dist(Distance);
};
Distance Distance::add_dist(Distance d2)
{
Distance temp;
temp.inches = inches + d2.inches;
if(temp.inches >= 12.0)
{
temp.inches -= 12.0;
temp.feet = 1;
}
temp.feet += feet + d2.feet;
return temp;
}
int main()
{
Distance dist1, dist3;
Distance dist2(11, 6.25);
dist1.getdist();
dist3 = dist1.add_dist(dist2);
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
return 0;
```

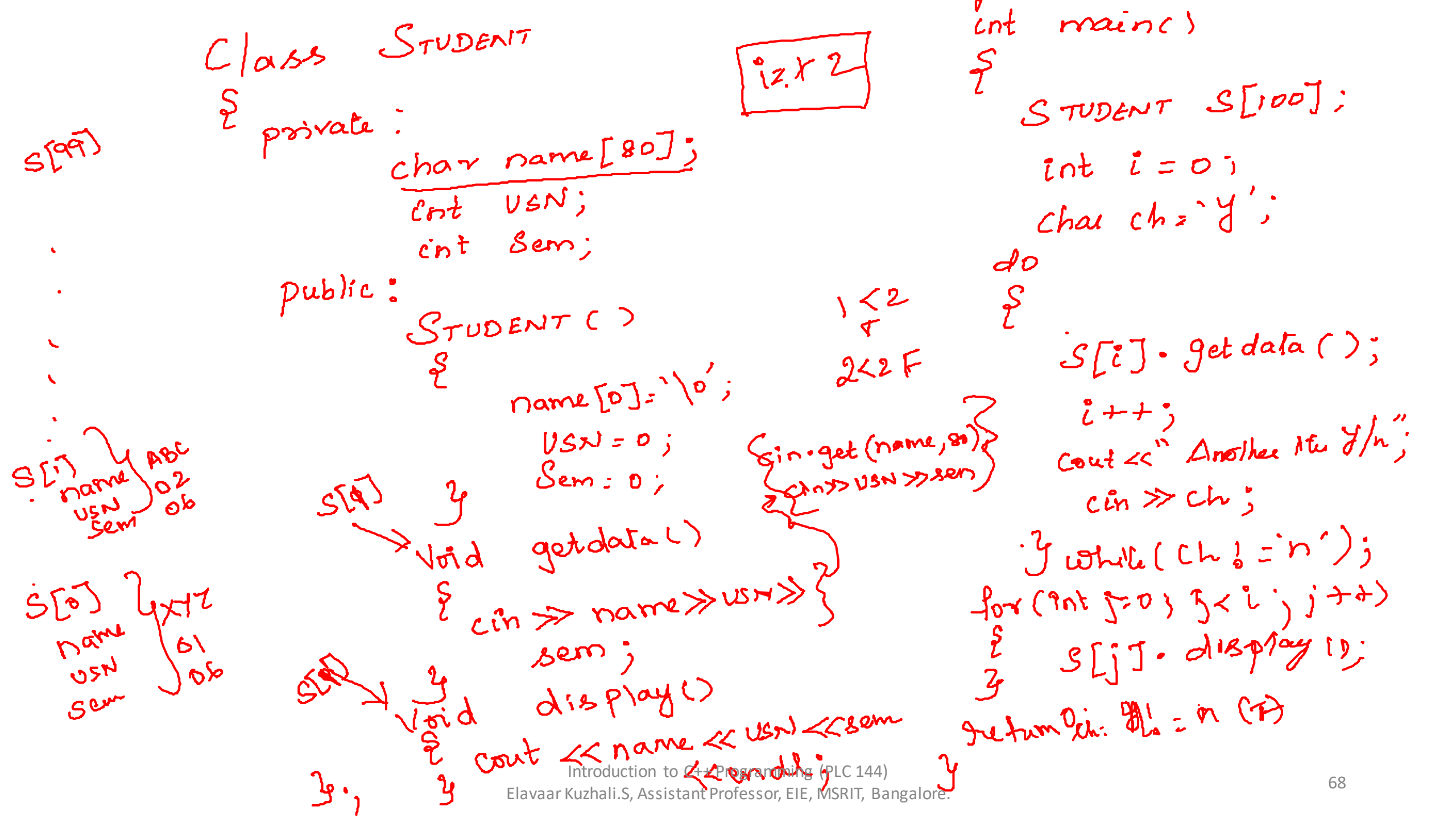


Arrays of Objects

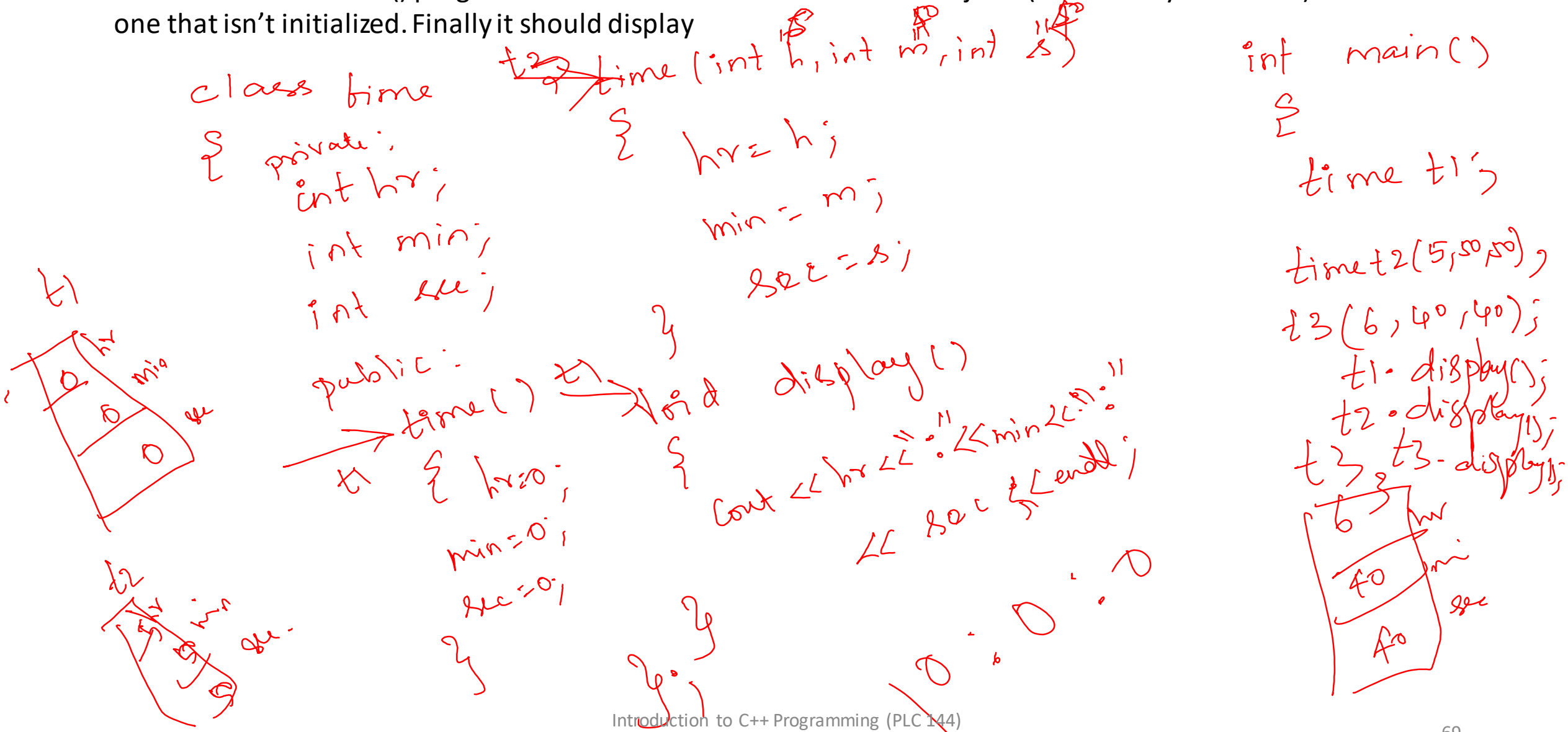
```
class Distance //English Distance class
{
private:
int feet;
float inches;
public:
void getdist()
{
cout << "\n Enter feet: "; cin >> feet;
cout << " Enter inches: "; cin >> inches;
}
void showdist() const
{ cout << feet << "\'-" << inches << '\"'; }
};
int main()
{
Distance dist[100];
int n=0;
char ans;
cout << endl;
do {
cout << "Enter distance number " << n+1;
dist[n++].getdist();
cout << "Enter another (y/n)?: ";
cin >> ans;
} while( ans != 'n' );
for(int j=0; j<n; j++)
{
cout << "\nDistance number " << j+1 << " is ";
dist[j].showdist();
}
cout << endl;
return 0;
}
```

A User-Defined String Type

```
class String
{
private:
enum { SZ = 80; };
char str[SZ];
public:
String()
{ str[0] = '\0'; }
String( char s[] )
{ strcpy(str, s); }
void display()
{ cout << str; }
void concat(String s2)
{
if( strlen(str)+strlen(s2.str) < SZ )
strcat(str, s2.str);
else
cout << "\nString too long";
}
};
int main()
{
String s1("Merry Christmas! ");
String s2 = "Season's Greetings!";
String s3;
cout << "\ns1="; s1.display();
cout << "\ns2="; s2.display();
cout << "\ns3="; s3.display();
s3 = s1;
cout << "\ns3="; s3.display();
s3.concat(s2);
cout << "\ns3="; s3.display();
cout << endl;
return 0;
}
```

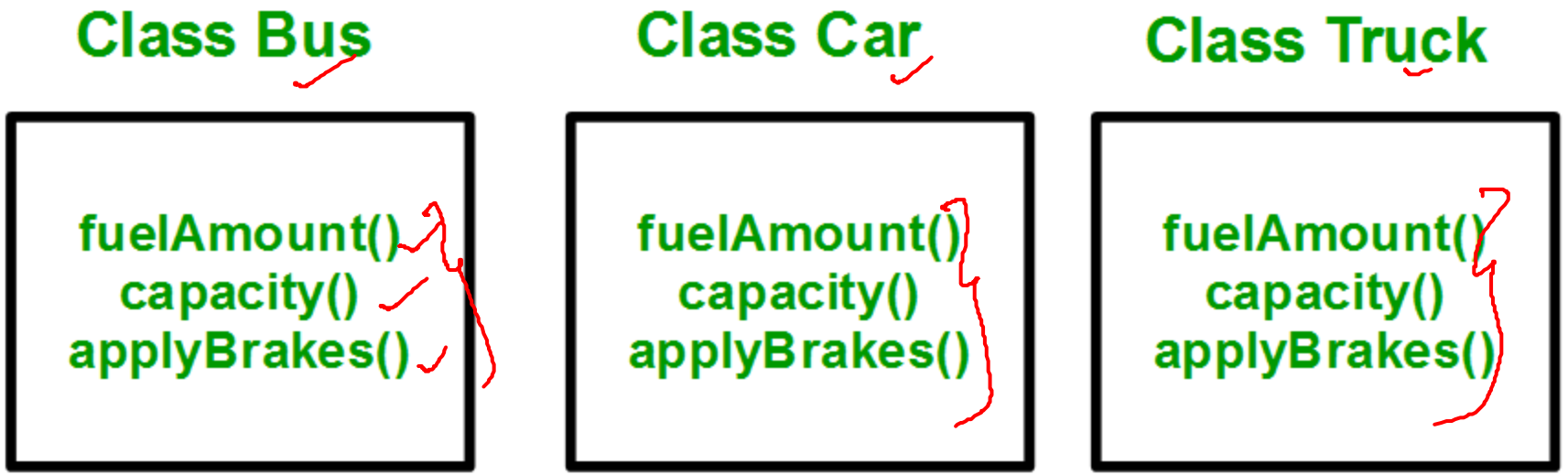

Class STUDENT
{
private:
char name[80];
int USN;
int Sem;
public:
STUDENT()
{
name[0]='\0';
USN=0;
Sem=0;
}
void getdata()
{
cin>>name>>USN>>
sem;
}
void display()
{
cout<<name<<USN<<sem
<<endl;
}
};
s[99]
s[i]
name
USN
Sem
ABC
02
06
s[0]
name
USN
sem
XYZ
01
06
s[0]
s[1]
cin.get(name,80)
cin>>USN>>sem
1<2
T
2<2 F
int main()
{
STUDENT s[100];
int i=0;
char ch='y';
do
{
s[i].getdata();
i++;
cout<<"Another Stu y/n";
cin>>ch;
}while(ch!='n');
for(int j=0;j<i;j++)
{
s[j].display();
}
return 0;

Create a class called time that has separate int member data for hours, minutes, and seconds. One constructor should initialize this data to 0, and another should initialize it to fixed values. A main() program should create two initialized time objects (should they be const?) and one that isn't initialized. Finally it should display

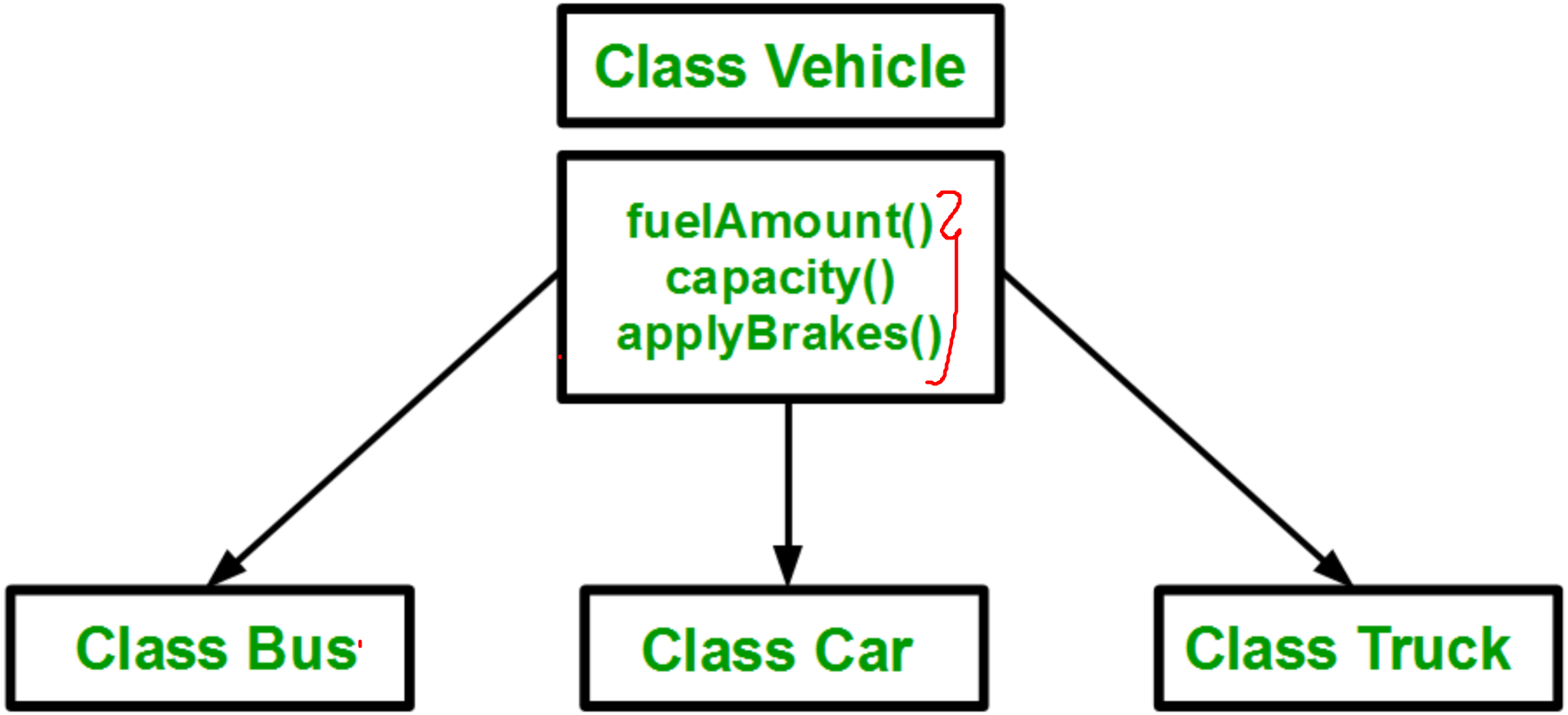


Inheritance – Extending class

- The capability of a class to derive properties and characteristics from another class is called **Inheritance**.
- The class that inherits properties from another class is called Sub class or Derived Class.
- The class whose properties are inherited by sub class is called Base Class or Super class.
- **Why and when to use inheritance?**



- Above process results in duplication of same code 3 times.
- This increases the chances of error and data redundancy.
- To avoid this type of situation, inheritance is used.



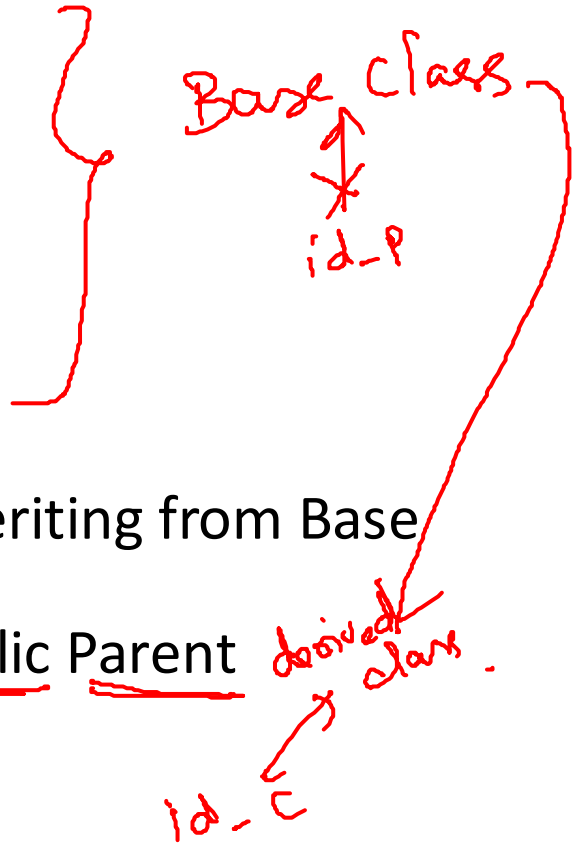
- If we create a class Vehicle and write these three functions in it
- inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability.
- **Implementing inheritance in C++**:

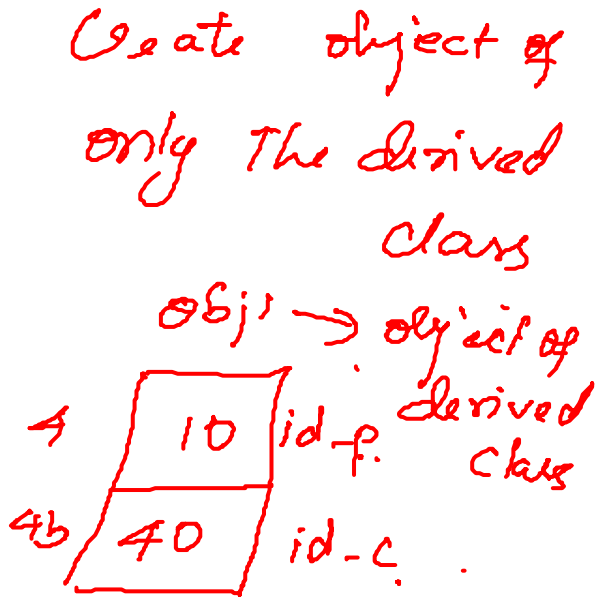
```
class Derivedclass_name : access_mode base_class_name
{
//body of subclass
}
```

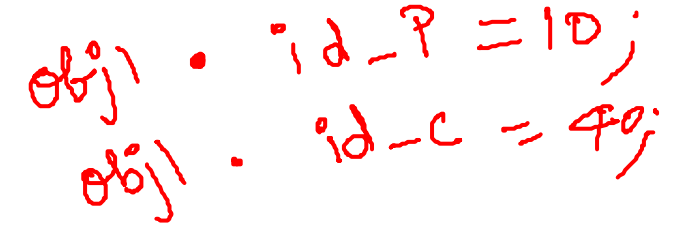
its own properties listed

private
public
Protected

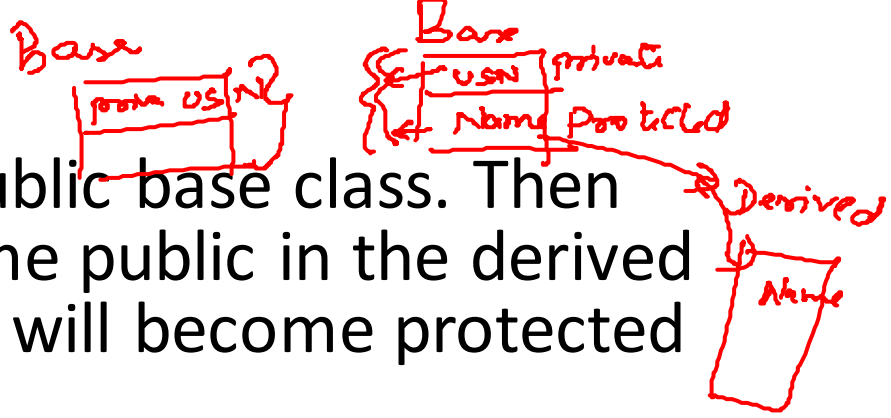
```
using namespace std;

//Base class
class Parent
{
        public:
        int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
        public:
        int id_c;
};

//main function
int main()
{

Child obj1;

obj1.id_c = 7;
obj1.id_p = 91;
cout << "Child id is " << obj1.id_c << endl;
cout << "Parent id is " << obj1.id_p << endl;
return 0;
}
```







- **Modes of Inheritance**
- **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
- **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
- **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.
- ***Protected: Protected access modifier is similar to private access modifier in the sense that it can't be accessed outside of it's class, the difference is that the class members declared as Protected can be accessed by any subclass(derived class) of that class as well.***



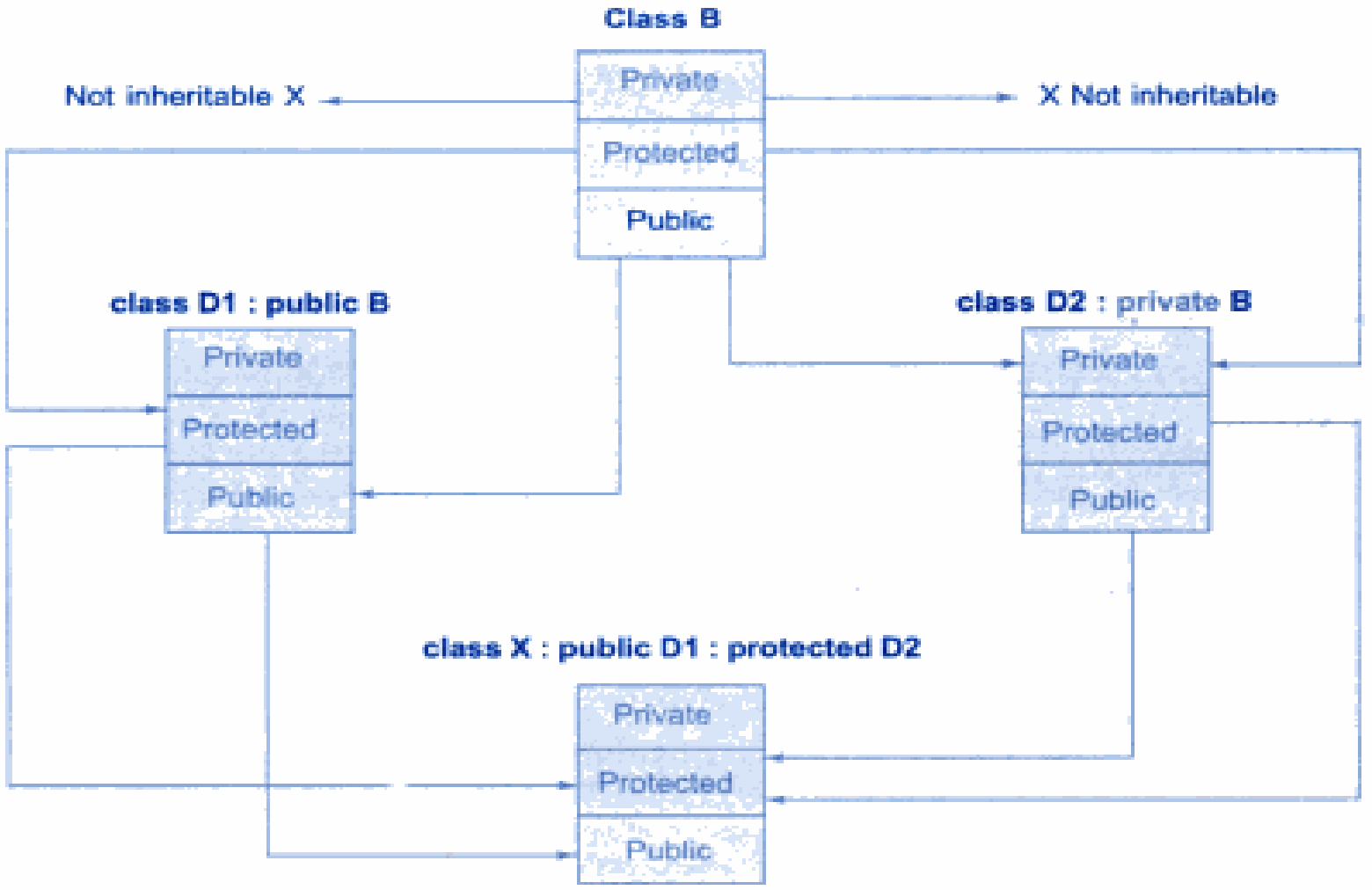
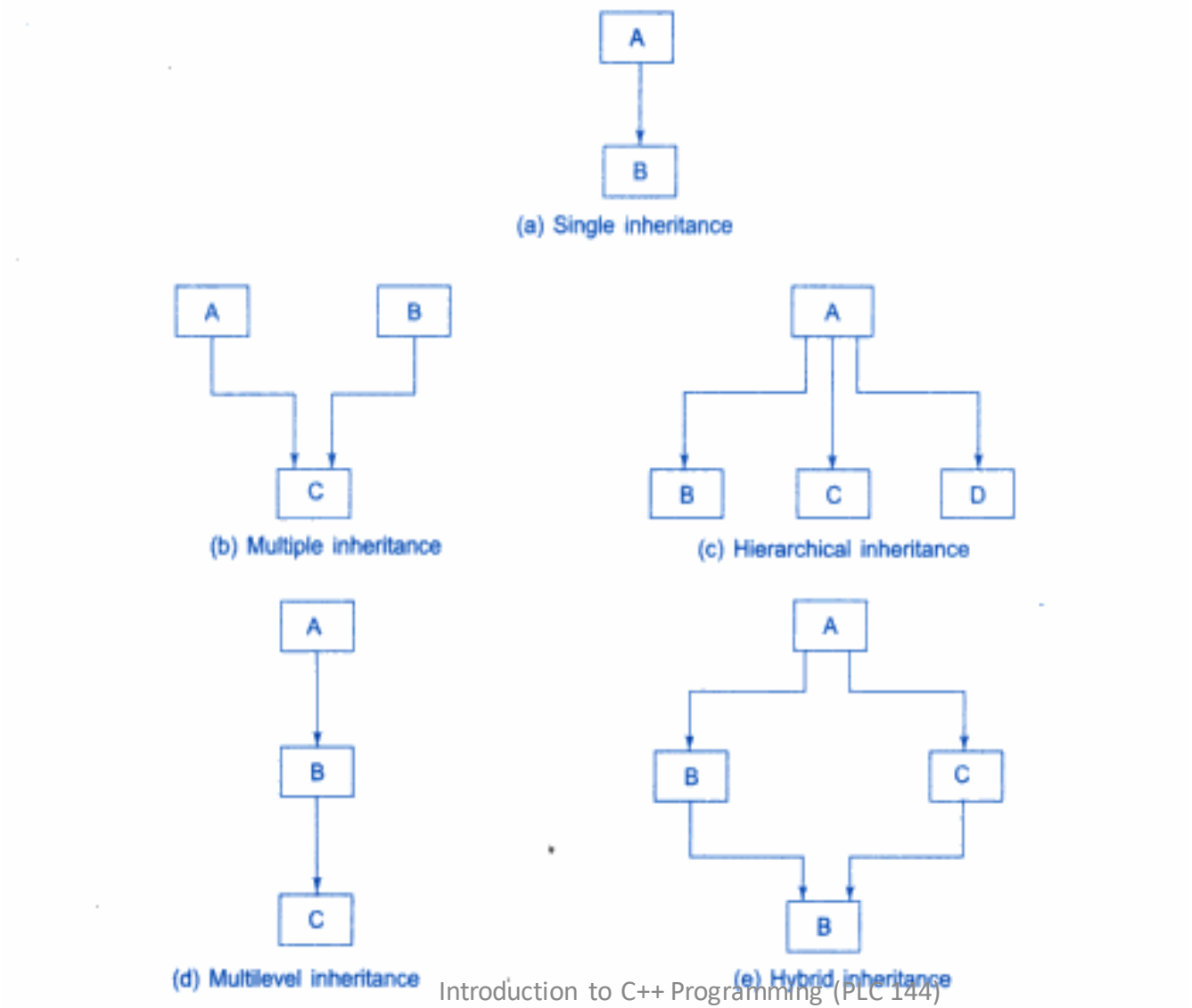
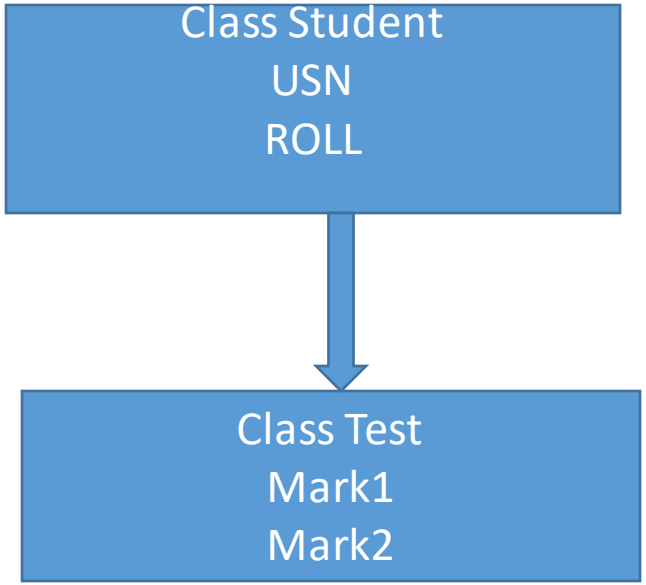


Fig. 8.4 ⇔ *Effect of inheritance on the visibility of members*

- Types of Inheritance



- Single Inheritance:

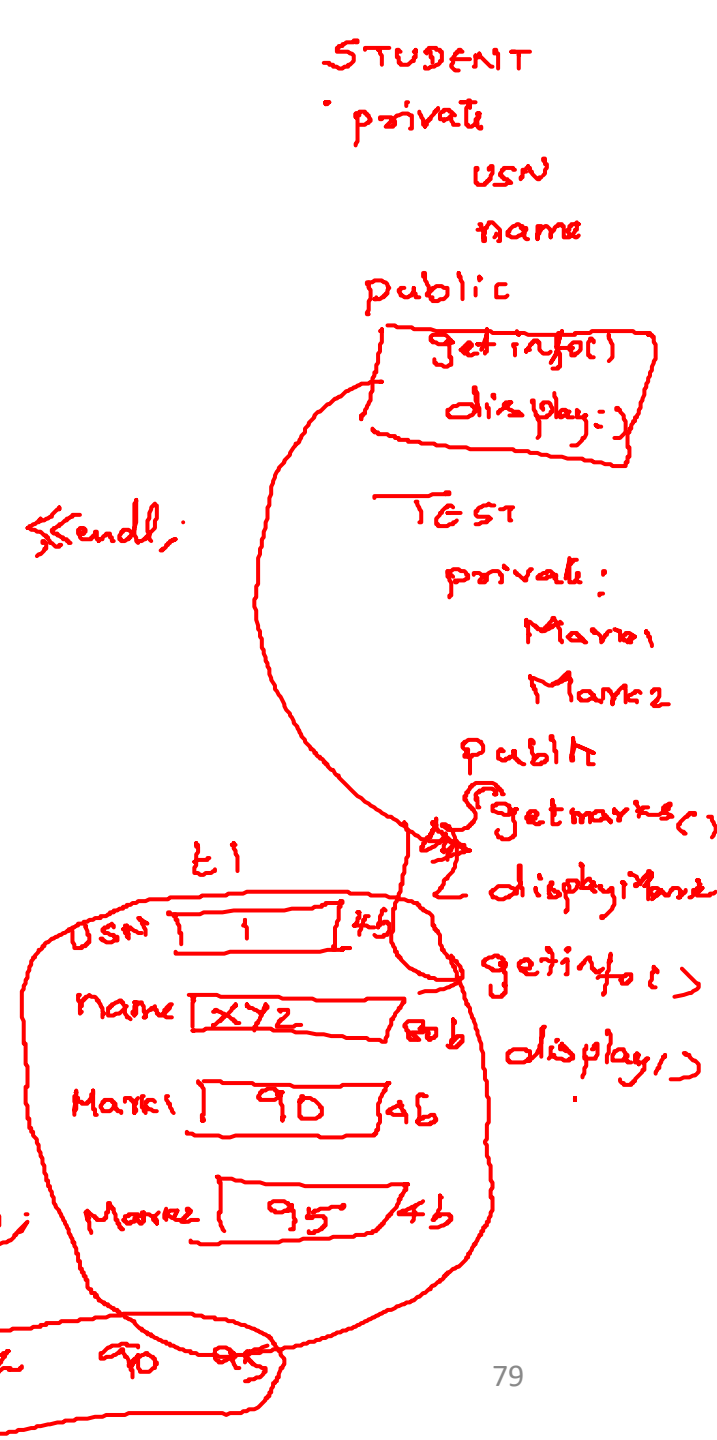


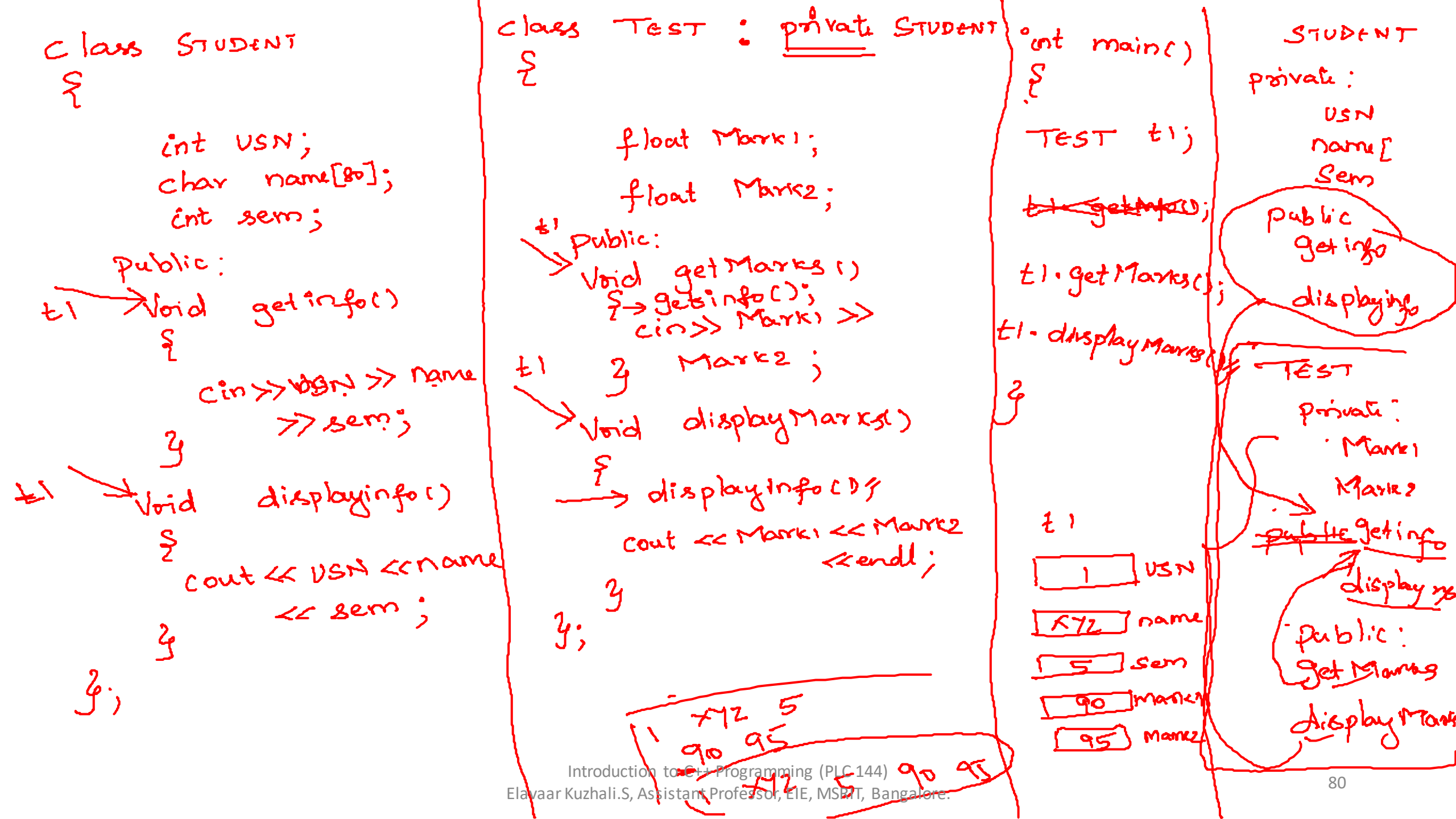
Base class

```
class STUDENT
{
    int USN;
    char name[80];
public:
    void getinfo()
    {
        cin >> USN >> name;
    }
    void display()
    {
        cout << USN << name;
    }
};
```

Derived class

```
class TEST : public STUDENT
{
    float Mark1;
    float Mark2;
public:
    void getmarks()
    {
        cin >> Mark1 >> Mark2;
    }
    void displayMarks()
    {
        cout << Mark1 << Mark2 << endl;
    }
};

int main()
{
    Test t1;
    t1.getinfo();
    t1.getmarks();
    t1.display();
    t1.displayMarks();
}
```

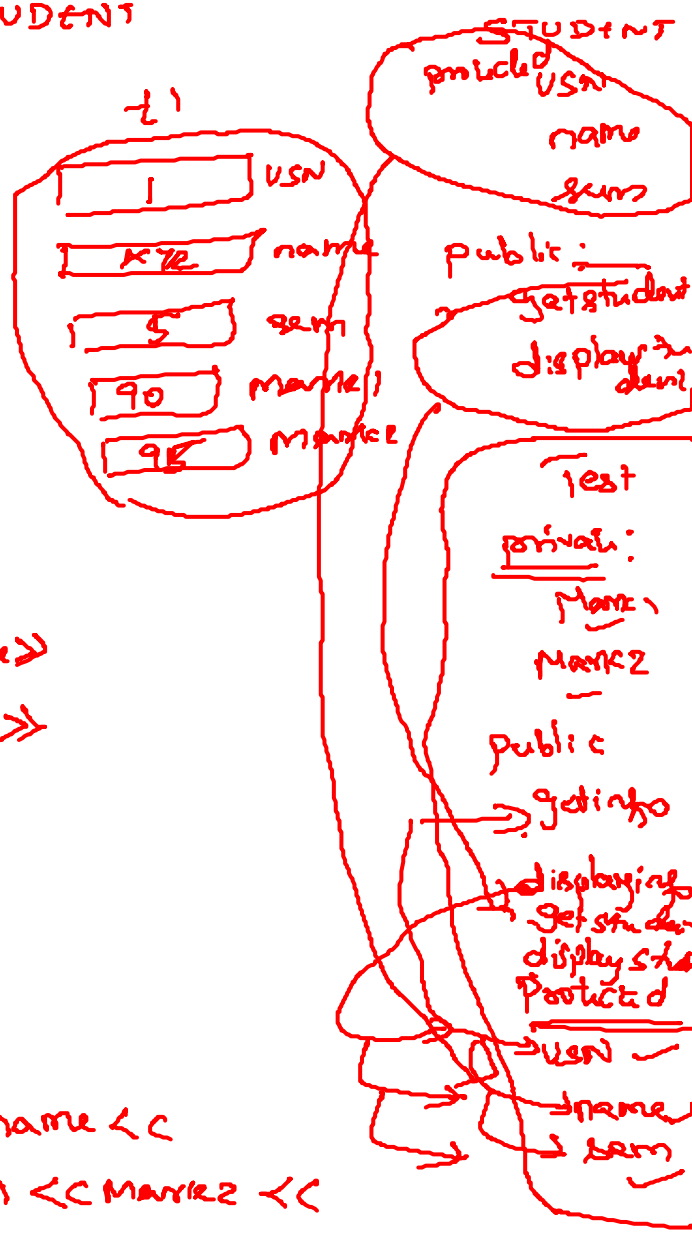



class STUDENT
{
int USN;
char name[80];
int sem;
public:
Void getinfo()
{
cin >> USN >> name
>> sem;
}
Void displayinfo()
{
cout << USN << name
<< sem;
}
};
class TEST : private STUDENT
{
float Mark1;
float Mark2;
public:
Void getMarks()
{ getinfo();
cin >> Mark1 >>
Mark2;
}
Void displayMarks()
{
displayinfo();
cout << Mark1 << Mark2
<< endl;
}
};
int main()
{
TEST t1;
t1.getMarks();
t1.displayMarks();
}
STUDENT
private:
USN
name[
Sem
public
getinfo
displayinfo
TEST
private:
Mark1
Mark2
getinfo
displayMarks
public:
GetMarks
displayMarks
t1
1 USN
XYZ name
5 sem
90 Mark1
95 Mark2
1 XYZ 5
90 95

```
class STUDENT
{
  protected:
     int USN;
     char name[80];
     int sem;
  public:
   void getstudent()
   {
     cin >> USN >> name >>
       sem;
   }
  void displaystudent()
   {
     cout << USN << name <<
        sem ;
   }
};
```

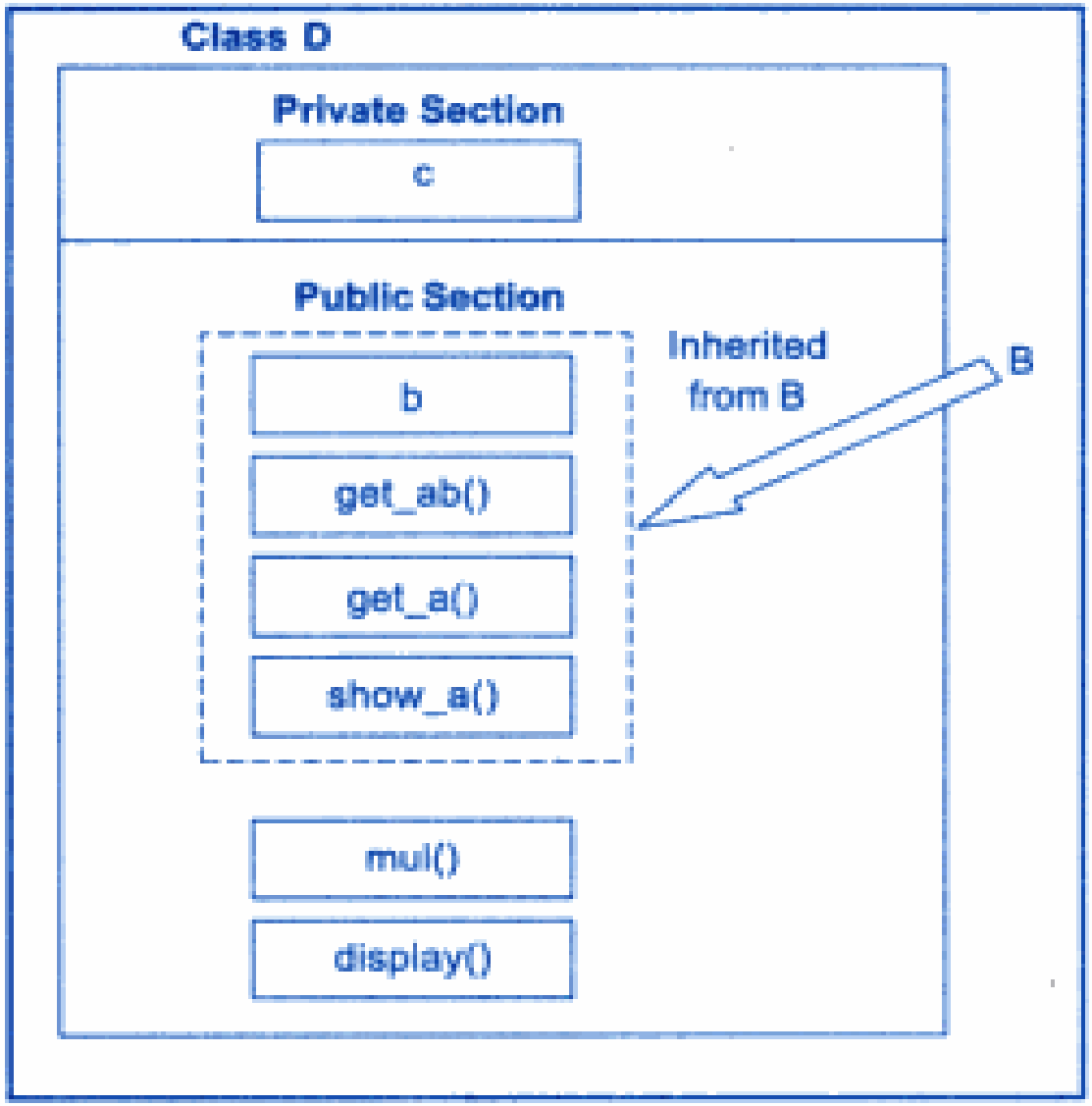
```
class TEST : public STUDENT
{
  private:
     float Mark1;
     float Mark2;
  public:
   void getinfo()
   {
     cin >> USN >> name >>
       sem >> Mark1 >>
       Mark2 ;
   }
  void displayinfo()
   {
     Cout << USN << name <<
       sem << Mark1 << Mark2 <<
                    endl;
   }
};
```

```
int main()
{
  TEST t1;
  t1.getinfo();
  t1.displayinfo();
  return 0;
}
```



SINGLE INHERITANCE : PUBLIC

```
#include <iostream>

using namespace std;

class B
{
    int a;                  // private; not inheritable
  public:
    int b;                  // public; ready for inheritance
    void get_ab();
    int  get_a(void);
    void show_a(void);
};

class D : public B          // public derivation
{
    int c;
  public:
    void mul(void);
    void display(void);
};
//
void B :: get_ab(void)
{
    a = 5; b = 10;
}
int B :: get_a()
{
    return a;
}
void B :: show_a()
{
    cout << "a = " << a << "\n";
}
void D :: mul()
{
    c = b * get_a();
}
void D :: display()
{
    cout << "a = " << get_a() << "\n";
    cout << "b = " << b << "\n";
    cout << "c = " << c << "\n\n";
}
int main()
{
    D d;

    d.get_ab();
    d.mul();
    d.show_a();
    d.display();

    d.b = 20;
    d.mul();
    d.display();

    return 0;
}
```


Class D
Private Section
c
Public Section
b
get_ab()
get_a()
show_a()
Inherited
from B
B
mul()
display()

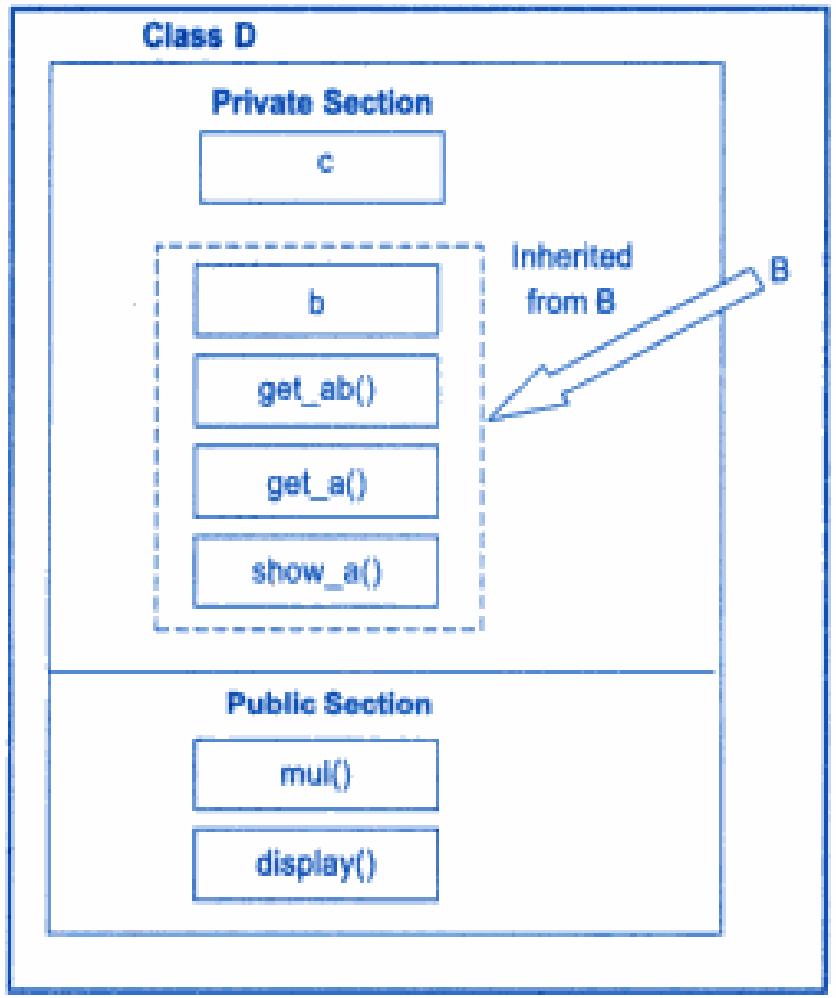


Fig. 8.3 ⇔ *Adding more members to a class (by private derivation)*

```
int main()
{
      D d;

      // d.get_ab();  WON'T WORK
      d.mul();
      // d.show_a();  WON'T WORK
      d.display();

    // d.b = 20;     WON'T WORK; b has become private
    d.mul();
    d.display();

    return 0;
}
```

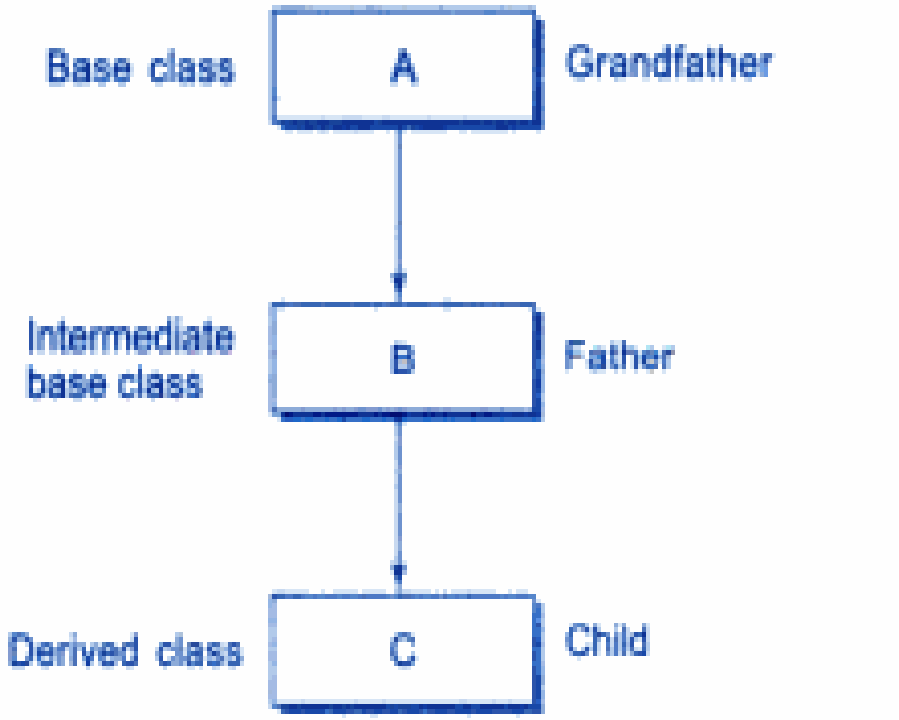
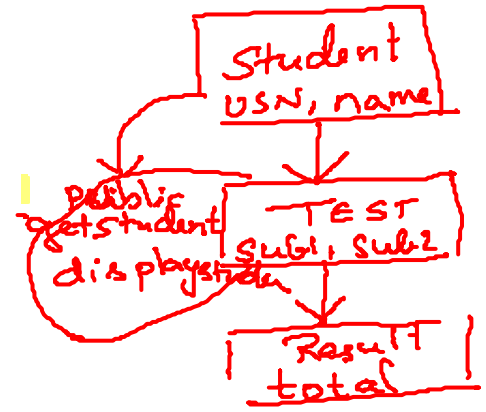


Fig. 8.7 ⇔ *Multilevel inheritance*

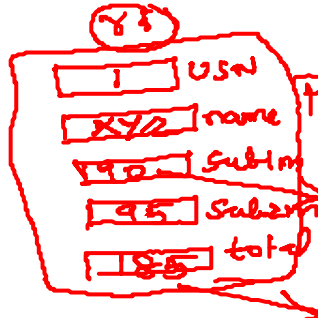
```
int main()
{
  Result r;
  r.getinfo();
  r.display();
}
```

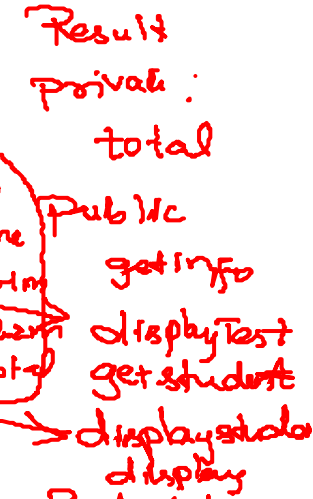
```
Class STUDENT
{
  Private:
    int USN;
    char name[80];
  public:
    Void getstudent()
    {
      cin>> USN>> name;
    }
    Void displaystudent()
    {
      cout << USN << name;
    }
};
```

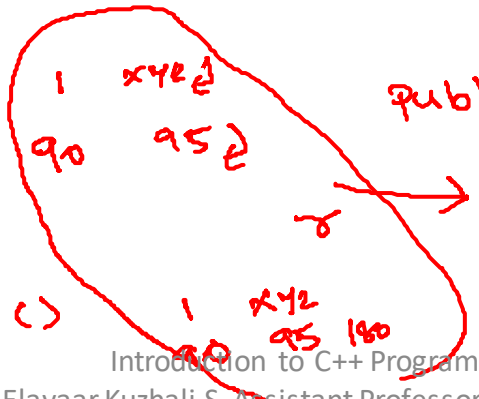
```
Class TEST : public STUDENT
{
  Protected:
    float sub1m;
    float sub2m;
  Public:
    Void getinfo()
    {
      getstudent();
      cin >> sub1m >> sub2m;
    }
    Void displayTest()
    {
      displaystudent();
      cout << sub1m << sub2m;
    }
};
```




```
class Result : public TEST
{
  private:
    float total;
  public:
    Void display()
    {
      total = sub1m + sub2m;
      displayTest();
      cout << total << endl;
    }
};
```







```
class student
{
      protected:
      int roll_number;
   public:
      void get_number(int);
      void put_number(void);
};
void student :: get_number(int a)
{
    roll_number = a;
}
void student :: put_number()
{
       cout << "Roll Number: " << roll_number << "\n";
}

class test : public student           // First level derivation
{
  protected:
     float sub1;
     float sub2;
  public:
     void get_marks(float, float);
     void put_marks(void);
};
```

```
void test :: get_marks(float x, float y)
{
     sub1 = x;
     sub2 = y;
}
void test :: put_marks()
{
     cout << "Marks in SUB1 = " << sub1 << "\n";
     cout << "Marks in SUB2 = " << sub2 << "\n";
}
class result : public test            // Second level derivation
{
     float total;              // private by default
  public:
     void display(void);
};
```

```
void result :: display(void)
{
     total = sub1 + sub2;
     put_number();
     put_marks();
     cout << "Total = " << total << "\n";
}

int main()
{
     result student1;        // student1 created

     student1.get_number(111);
     student1.get_marks(75.0, 59.5);

     student1.display();

   return 0;
}
```

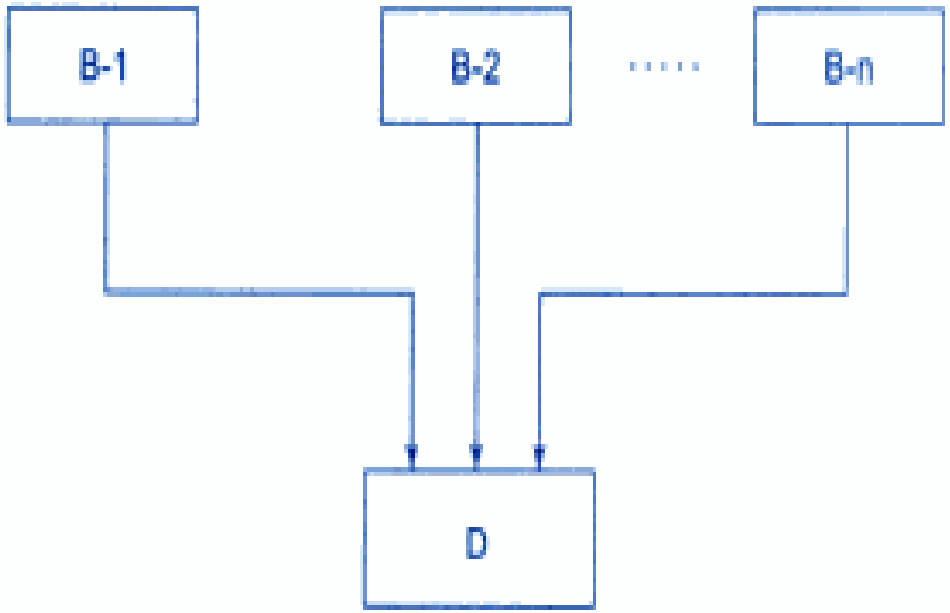
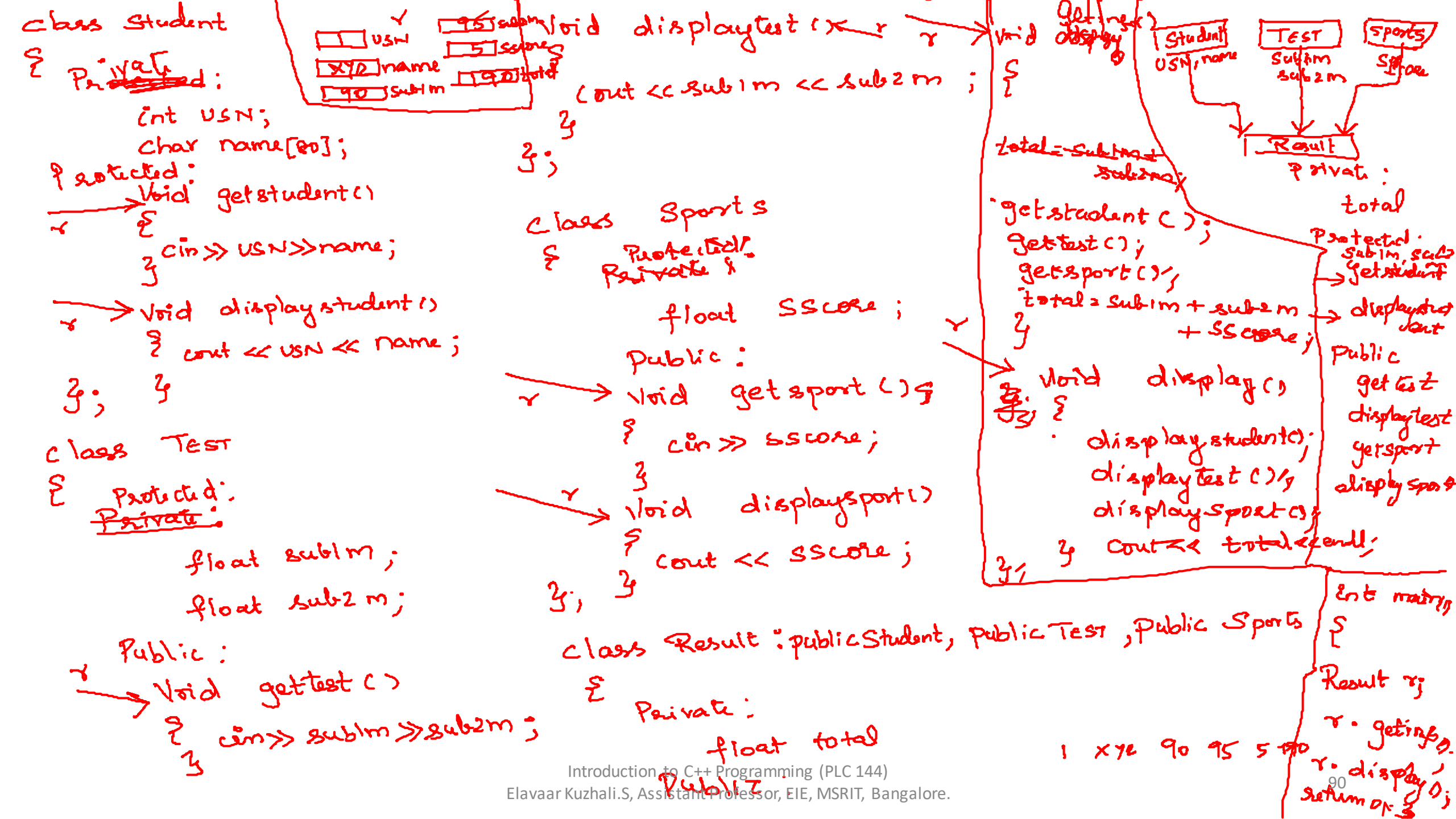


Fig. 8.8 ⇔ *Multiple inheritance*

```
class M
{
  protected:
        int m;
  public:
        void get_m(int);
};

class N
{
  protected:
      int n;
  public:
      void get_n(int);
};

class P : public M, public N
{
  public:
      void display(void);
};

void M :: get_m(int x)
{
     m = x;
}

void N :: get_n(int y)
{
     n = y;
}

void P :: display(void)
{
     cout << "m = " << m << "\n";
     cout << "n = " << n << "\n";
     cout << "m*n = " << m*n << "\n";
}
```

```
int main()
{
     P p;

     p.get_m(10);
     p.get_n(20);
     p.display();

     return 0;
}
```


class student
{
private:
int USN;
char name[20];
protected:
void getstudent()
{
cin>> USN>>name;
}
void displaystudent()
{ cout << USN << name;
};
class Test
{
protected:
private:
float sub1m;
float sub2m;
Public:
void gettest()
{
cin>> sub1m >>sub2m;
}
void displaytest(){ cout<<sub1m<<sub2m;
};
class Sports
{
Protected:
Private:
float SScore;
Public:
void getsport(){
cin>> sscore;
}
void displaysport()
{
cout << sscore;
};
class Result :publicStudent, publicTest ,Public Sports
{
Paivate:
float total
Public:
void display()
{
displaystudent();
displaytest();
displaysport();
cout<< total<<endl;
};
Student
USN, name
Test
Sub1m
sub2m
Sports
SScore
Result
Private:
total
Protected:
sub1m, sub2m
SScore
total=sub1m+sub2m+SScore;
getstudent();
gettest();
getsport();
Public
getstudent
gettest
displaytest
getsport
displaysport
displaystudent
int main()
{
Result r;
r.getinfo();
r.display();
return 0;
}
1 xyz 90 95 5 190

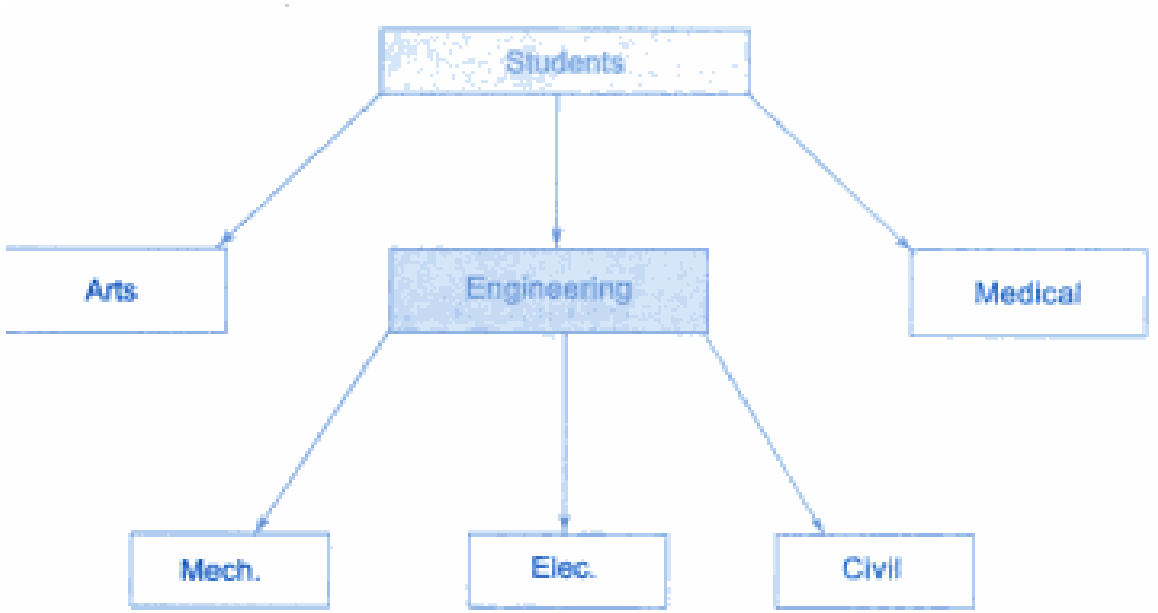
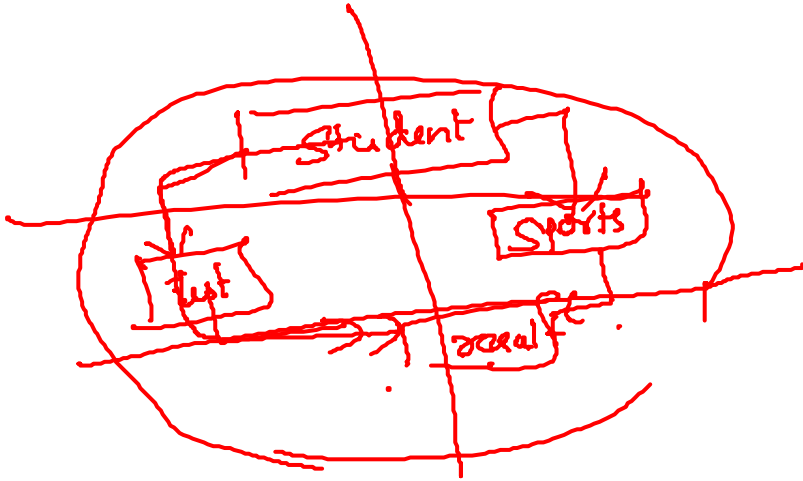
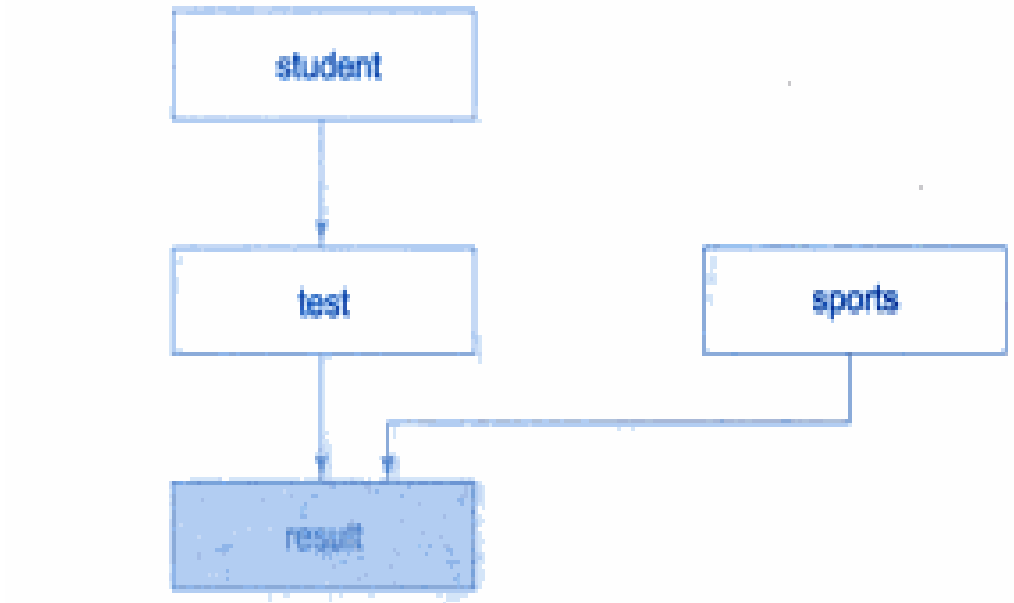


Fig. 8.9 ⇔ *Hierarchical classification of students*

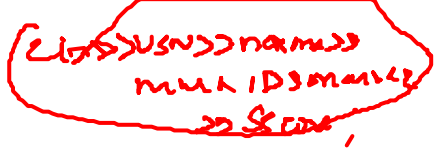
- Hybrid Inheritance

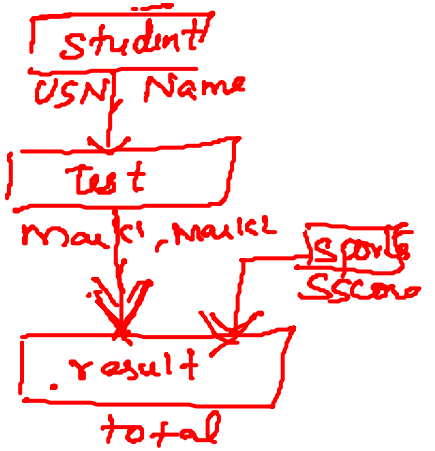


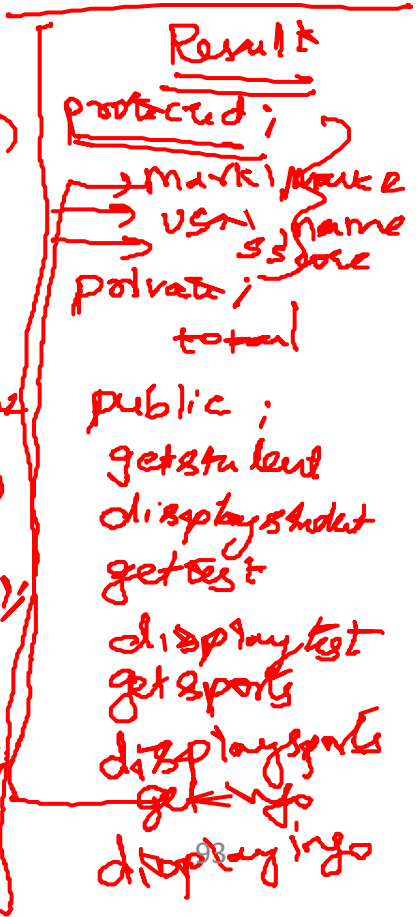


. 8.11 ⇔ *Multilevel, multiple inheritance*

```
class Student
{
protected:
    int USN;
    char name[50];
public:
    void getstudent()
    { cin >> USN >> name;
    }
    void displaystudent()
    {
    cout << USN << name
    }
};

class test : public Student
{
protected:
    float mark1, mark2;
public:
    void gettest()
    { cin >> mark1 >> mark2;
    }
};

class Sports
{
protected:
    float Sscore;
public:
    void getsports()
    { cin >> sscore;
    }
    void displaysports()
    { cout << sscore;
    }
};

int main()
{ result r;
  r.getinfo();
  r.displayinfo();
}

void displaytest()
{
cout << mark1 << mark2;
}
};

class result : public test,
               public Sports
{
private:
    float total;
public:
    void getinfo()
    {
    getstudent();
    gettest();
    getsports();
    total = mark1 + mark2
            + sscore
    }
    void displayinfo()
    {
    displaystudent();
    displaytest();
    displaysports();
    cout << total <<
    endl;
    }
};
```







```
class sports
{
  protected:
     float  score;
  public:
     void get_score(float);
      void put_score(void);
};
```

```
class result : public test, public sports
{
      .....
      .....
};
```

```
#include <iostream>

using namespace std;

class student
{
  protected:
      int  roll_number;
  public:
      void get_number(int a)
      {
          roll_number = a;
      }
      void put_number(void)
      {
            cout << "Roll No: " << roll_number << "\n";
      }
};

class test : public student
{
  protected:
      float part1, part2;
  public:
      void get_marks(float x, float y)
      {
            part1 = x;  part2 = y;
      }
      void put_marks(void)
      {
            cout << "Marks obtained: " << "\n"
                 << "Part1 = " << part1 << "\n"
                 << "Part2 = " << part2 << "\n";
      }
};
```

```
class sports
{
   protected:
       float score;
   public:
       void get_score(float s)
       {
            score = s;
       }
       void put_score(void)
       {
            cout << "Sports wt: " << score << "\n\n";
       }
};

class result : public test, public sports
{
       float total;
  public:
       void display(void);
};

void result :: display(void)
{
      total = part1 + part2 + score;

      put_number();
      put_marks();
      put_score();

      cout << "Total Score: " << total << "\n";
}

int main()
{
      result student_1;
      student_1.get_number(1234);
      student_1.get_marks(27.5, 33.0);
      student_1.get_score(6.0);
      student_1.display();
      return 0;
}
```

Abstract Class

- Is the one that is not used to create objects.
- They are designed to act as Base class.
- It is the design concept in program development and provides a base upon which other classes are built.

- Constructors in derived classes

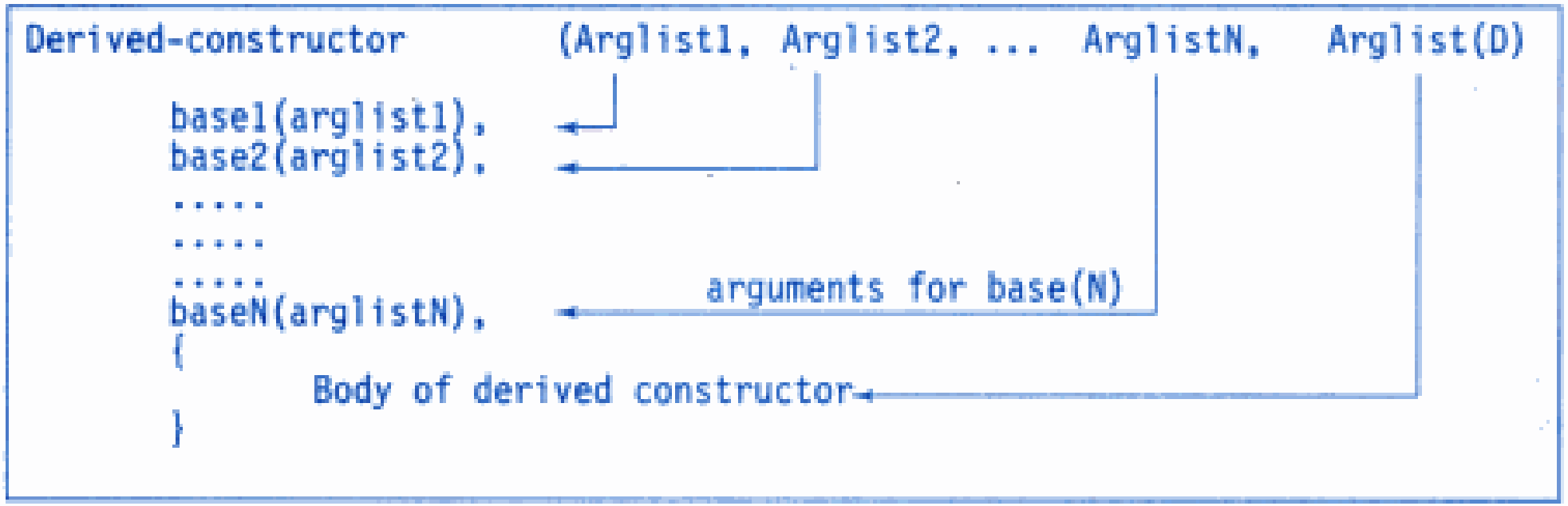
here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is *mandatory* for the derived class to have a constructor and pass the arguments to the base class constructors. Remember, while applying inheritance we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base classes are constructed *in the order in which they appear in the declaration of the derived class*. Similarly, in a multilevel inheritance, the constructors will be executed *in the order of inheritance*.

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. How are they passed to the base class constructors so that they can do their job? C++ supports a special argument passing mechanism for such situations.

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor.

The general form of defining a derived constructor is:



Example:

```
D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2),          /* call to constructor A */
B(b1, b2)          /* call to constructor B */
{
     d = d1;       // executes its own body
}
```

```
.....
D objD(5, 12, 2.5, 7.54, 30);
.....
```

5 ⟶ a1
12 ⟶ a2
2.5 ⟶ b1
7.54 ⟶ b2
30 ⟶ d1

Table 8.2 *Execution of base class constructors*

Method of inheritance	***Order of execution***

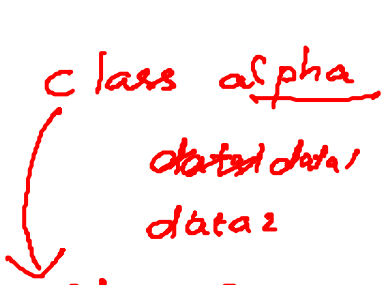
Class B: public A
{
};
	A( ) ; base constructor
	B( ) ; derived constructor

class A : public B, public C
{
};
	B( ) ; base(first)
	C( ) ; base(second)
	A( ) ; derived

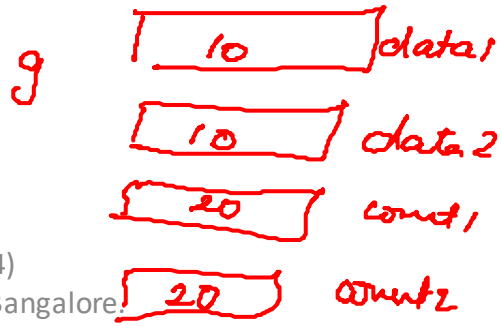
class A : public B, virtual public C
{
};
	C( ) ; virtual base
	B( ) ; ordinary base
	A( ) ; derived

```
class alpha
{
  private:
    int data1;
    int data2;
  public:
    alpha ( )
    {
      data1 = 10;
      data2 = 10;
    }
    void displayalpha ( )
    {
      cout << data1 << data2
           << endl;
    }
};
```

```
class gamma : public alpha
{
  private:
    int count1;
    int count2;
  public:
    gamma ( )
    {
      count1 = 20;
      count2 = 20;
    }
    void displaygamma ( )
    {
      cout << count1 << count2
           << endl;
    }
};
```




```
int main ( )
{
  gamma g;
  g.displayalpha ( );
  g.displaygamma ( );
  return 0;
}
```

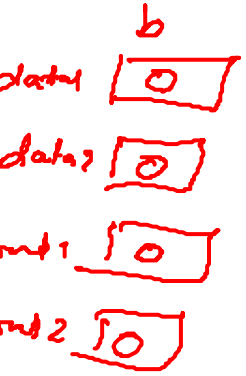


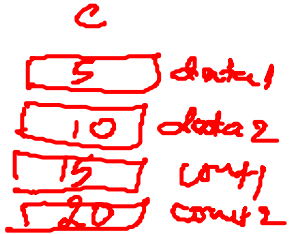
```
class alpha
{ private :
      int data1;
      int data2;
  public :
    alpha ( )
    {
      data1 = 0;
      data2 = 0;
    }
    alpha ( int i, int j )
    {
      data1 = i ;
      data2 = j ;
    }
  void display alpha ( )
  {
    cout << data1 << data2
         << endl;
  }
};
```

```
class beta : public alpha
{ private :
      int count1 ;
      int count2 ;
  public :
    beta ( )
    {
      count1 = 0;
      count2 = 0;
    }
    beta ( int x, int y, int l, int m ) : alpha ( x , y )
    {
      count1 = l ;
      count2 = m;
    }
  void display gamma ( )
  {
    cout << count1 << count2 << endl;
  }
};
```

```
int main ( )
{
  beta b ;
  beta c(5, 10, 15, 20);
  b . display alpha ( );
  b . display gamma ( );
  c . display alpha ( );
  c . display gamma ( );
}
```

alpha: data1, data2 — beta: count1, count2





```
class alpha
{
    int x;
  public:
    alpha(int i)
    {
        x = i;
        cout << "alpha initialized \n";
    }
    void show_x(void)
    { cout << "x = " << x << "\n"; }
};
```

```
class beta
{
    float y;
  public:
    beta(float j)
    {
        y = j;
        cout << "beta initialized \n";
    }
    void show_y(void)
    { cout << "y = " << y << "\n"; }
};
```

```
class gamma: public beta, public alpha
{
    int m, n;
  public:
    gamma(int a, float b, int c, int d):
        alpha(a), beta(b)
    {
        m = c;
        n = d;
        cout << "gamma initialized \n";
    }
    void show_mn(void)
    {
        cout << "m = " << m << "\n"
             << "n = " << n << "\n";
    }
};

int main()
{
    gamma g(5, 10.75, 20, 30);
    cout << "\n";
    g.show_x();
    g.show_y();
    g.show_mn();

    return 0;
}
```

```
beta initialized
alpha initialized
gamma initialized

x = 5
y = 10.75
m = 20
n = 30
```