COMPUTER PROGRAMMING WEEK 5

(SEP 11 – 15, 2017)

Instructor:

Abdul Aziz
Assistant Professor
(Computer Science Department)
National University- FAST (KHI Campus)

ACKNOWLEDGMENT

- Publish material by Virtual University of Pakistan.
- Publish material by Deitel & Deitel.
- Publish material by Robert Lafore.

CASTING OR TYPE CASTING

- Typecasting is making a variable of one type, such as an int , act like another type, such as char.
- Converting one type into another.
- Type of casting:
  - Implicit cast: Perform by the compiler automatically.
  - Explicit cast: Perform by the programmer. Data loss might occur.

IMPLICIT CAST

The compiler converts types using the rule that the “Smaller” type is converted to the “wider” type.

(Short OR Char) > (int) > (unsigned) > (long int) > (unsigned long int) > (float) >

(double) > (long double)

Example:

```
int a=10;

float f;

f=a; // implicit casting
```

EXPLICIT CAST

Syntax:

(type_name) expression/value

Or

Type_name (expression/value)

Example:

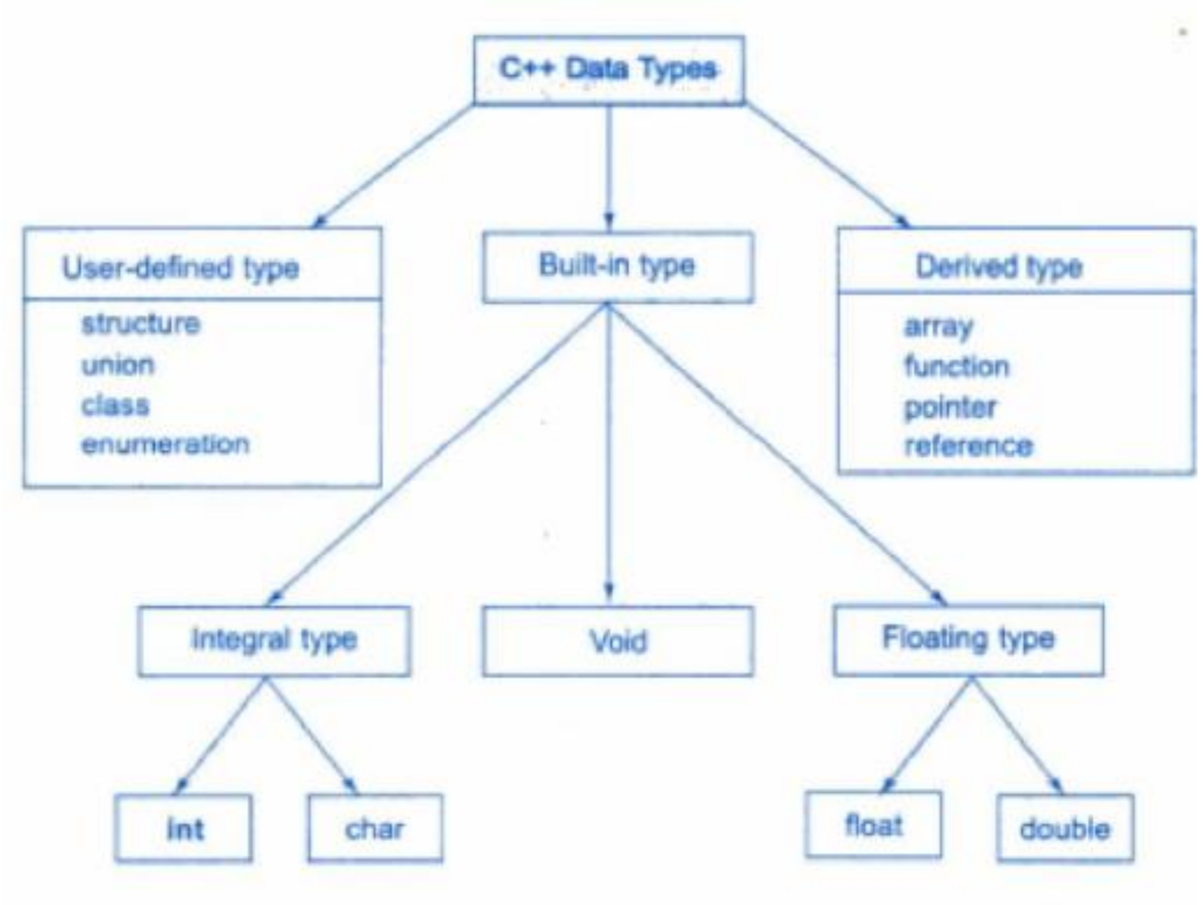
Float f=15.65;

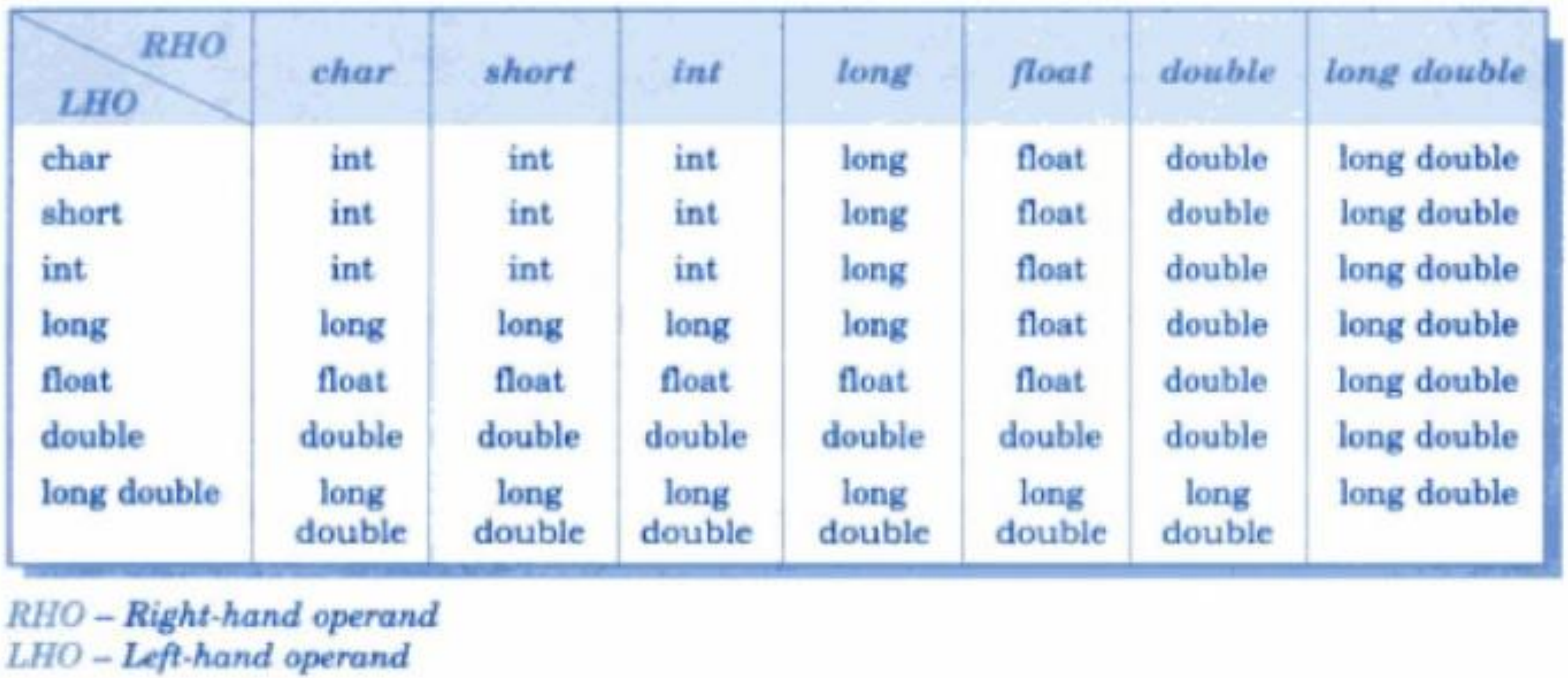
int a;

a= (int) f;

Or

a= int(f);


C++ Data Types
User-defined type
structure
union
class
enumeration
Built-in type
Derived type
array
function
pointer
reference
Integral type
Void
Floating type
int
char
float
double



RHO LHO	char	short	int	long	float	double	long double
char	int	int	int	long	float	double	long double
short	int	int	int	long	float	double	long double
int	int	int	int	long	float	double	long double
long	long	long	long	long	float	double	long double
float	float	float	float	float	float	double	long double
double	double	double	double	double	double	double	long double
long double	long double	long double	long double	long double	long double	long double	long double

RHO – *Right-hand operand*
LHO – *Left-hand operand*

RUN-TIME TYPE IDENTIFICATION (**RTTI**)

Type casting operators

i) Typeid: used to obtain an object type. We must include <typinfo> in order to used typeid.

ii) dynamic_cast

iii) const_cast

iv) static_cast

v) reinterpret_cast

EXPLICIT CONSTRUCTOR

We can make a constructor explicit by using explicit keyword.

```
class student{
private:
        int rollno;
 public:
        student(){
        }
        explicit student(int r){
                rollno=r;
        }
}
```

SEPARATION OF INTERFACE AND IMPLEMENTATION

- Public member function exposed by a class is called interface.
- Separation of implementation from the interface is good software engineering.
- User is only concerned about ways of accessing data (interface).
- User has no concern about the internal representation and implementation of the class.
- Usually functions are defined in implementation files (.cpp) while the class definition is given in header file (.h).
- Some authors also consider this as separation of interface and implementation.

`CONST` MEMBER FUNCTIONS

There are functions that are meant to be read only

There must exist a mechanism to detect error if such functions accidentally change the data member

`CONST` MEMBER FUNCTIONS

Keyword `const` is placed at the end of the parameter list

CONST MEMBER FUNCTIONS

Declaration:

```
class ClassName{
 ReturnVal Function() const;
};
```

Definition:

```
ReturnVal ClassName::Function() const{
 …
}
```

EXAMPLE

```
class Student{
public:
 int getRollNo() const{
      return rollNo;
 }
};
```

CONST FUNCTIONS

Constant member functions cannot modify the state of any object

They are just ***"read-only"***

Errors due to typing are also caught at compile time

EXAMPLE

```
bool Student::isRollNo(int aNo){
 if(rollNo == aNo){
      return true;
 }
 return false;
}
```

EXAMPLE

```
bool Student::isRollNo(int aNo){
 /*undetected typing mistake*/
 if(rollNo = aNo){
     return true;
 }
 return false;
}
```

EXAMPLE

```
bool Student::isRollNo(int aNo)const{
 /*compiler error*/
 if(rollNo = aNo){
      return true;
 }
 return false;
}
```

`CONST` FUNCTIONS

Constructors and Destructors cannot be **`const`**

Constructor and destructor are used to modify the object to a well defined state

EXAMPLE

```
class Time{
public:
 Time() const {}   //error…
 ~Time() const {}  //error…
 };
```

`CONST` FUNCTION

Constant member function cannot change data member

Constant member function cannot access non-constant member functions

EXAMPLE

```
class Student{
 int  rollno;
public:
 int getRollno();
 void setRollno(int aRoll);
 int ConstFunc() const{
     rollno = getRollno();  //error
     setRollno(123);//error
 }
};
```

MEMBER INITIALIZER LIST

A member initializer list is a mechanism to initialize data members

It is given after closing parenthesis of parameter list of constructor

In case of more then one member use comma separated list

EXAMPLE

```
class Student{
 const int rollNo;
 char *name;
 float GPA;
public:
 Student(int aRollNo)
 : rollNo(aRollNo), name(Null), GPA(0.0){
        …
 }
…
};
```

ORDER OF INITIALIZATION

Data member are initialized in order they are declared

Order in member initializer list is not significant at all

EXAMPLE

```
class ABC{
int x;
int y;
int z;
public:
ABC();
};
```

EXAMPLE

```
ABC::ABC():y(10),x(y),z(y)
{
…
}
/*  x = Junk value
    y = 10
    z = 10 */
```

CONST OBJECTS

Objects can be declared constant with the use of const keyword

Constant objects cannot change their state

EXAMPLE

```
int main()
{
 const Student aStudent;
 return 0;
}
```

EXAMPLE

```
class Student{
…
 int rollNo;
public:
…
 int getRollNo(){
     return rollNo;
 }
};
```

EXAMPLE

```
int main(){
 const Student aStudent;
 int a = aStudent.getRollNo();
 //error
}
```

`CONST` OBJECTS

`const` objects cannot access "*non const*" member function

Chances of unintentional modification are eliminated

EXAMPLE

```
class Student{
…
 int rollNo;
public:
…
 int getRollNo()const{
     return rollNo;
 }
};
```

EXAMPLE

```
int main(){
 const Student aStudent;
 int a = aStudent.getRollNo();
}
```

CONSTANT DATA MEMBERS

Make all functions that don’t change the state of the object constant

This will enable constant objects to access more member functions

STATIC VARIABLES

Lifetime of static variable is throughout the program life

If static variables are not explicitly initialized then they are initialized to 0 of appropriate type

EXAMPLE

```
 :
static int staticInt ; // in class definition.
 :
void func1(int i){
       staticInt = i
       cout << staticInt << endl;
}
int main(){
 func1(10);
 func1(20);
}
```

Output:

```
10
20
```

STATIC DATA MEMBER

Definition

“A variable that is part of a class, yet is not part of an object of that class, is called static data member”

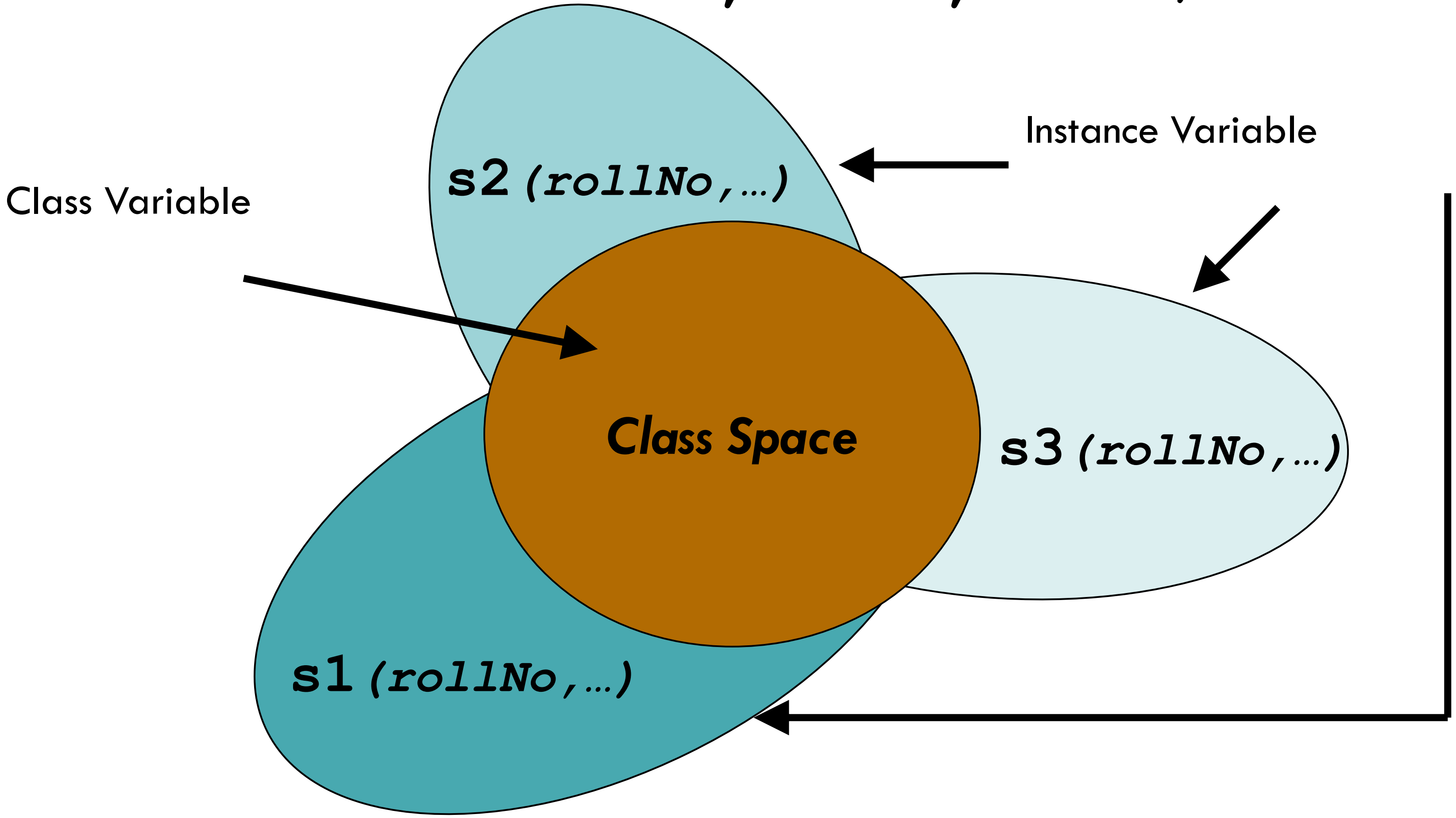
STATIC DATA MEMBER

They are shared by all instances of the class

They do not belong to any particular instance of a class

CLASS VS. INSTANCE VARIABLE

```
Student s1, s2, s3;
```



STATIC DATA MEMBER (SYNTAX)

Keyword static is used to make a data member static

```
class ClassName{
…
static DataType VariableName;
};
```

DEFINING STATIC DATA MEMBER

Static data member is declared inside the class

But they are defined outside the class

DEFINING STATIC DATA MEMBER

```
class ClassName{
…
static DataType VariableName;
};

DataType ClassName::VariableName;
```

INITIALIZING STATIC DATA MEMBER

Static data members should be initialized once at file scope

They are initialized at the time of definition

EXAMPLE

```
class Student{
private:
    static int noOfStudents;
public:
   …
};
int Student::noOfStudents = 0;
/*private static member cannot be accessed
outside the class except for initialization*/
```

INITIALIZING STATIC DATA MEMBER

If static data members are not explicitly initialized at the time of definition then they are initialized to 0

EXAMPLE

```
int Student::noOfStudents;
```

is equivalent to

```
int Student::noOfStudents=0;
```

ACCESSING STATIC DATA MEMBER

To access a static data member there are two ways

- Access like a normal data member
- Access using a scope resolution operator ‘::’

EXAMPLE

```
class Student{
public:
 static int noOfStudents;
};
int Student::noOfStudents;
int main(){
 Student aStudent;
 aStudent.noOfStudents = 1;
 Student::noOfStudents = 1;
}
```

LIFE OF STATIC DATA MEMBER

They are created even when there is no object of a class

They remain in memory even when all objects of a class are destroyed

EXAMPLE

```
class Student{
public:
 static int noOfStudents;
};
int Student::noOfStudents;
int main(){
 Student::noOfStudents = 1;
}
```

EXAMPLE

```
class Student{
public:
 static int noOfStudents;
};
int Student::noOfStudents;
int main(){
 {
  Student aStudent;
  aStudent.noOfStudents = 1;
 }
 Student::noOfStudents = 1;
}
```

USES

They can be used to store information that is required by all objects, like global variables

PROBLEM

noOfStudents is accessible outside the class

Bad design as the local data member is kept public

STATIC MEMBER FUNCTION

Definition:

“The function that needs access to the members of a class, yet does not need to be invoked by a particular object, is called static member function”

STATIC MEMBER FUNCTION

They are used to access static data members

Access mechanism for static member functions is same as that of static data members

They cannot access any non-static members

EXAMPLE

```
class Student{
 static int noOfStudents;
 int rollNo;
public:
 static int getTotalStudent(){
       return noOfStudents;
 }
};
int main(){
 int i = Student::getTotalStudents();
}
```

ACCESSING NON STATIC DATA MEMBERS

```
int Student::getTotalStudents(){
 return rollNo;
}
int main(){
 int i = Student::getTotalStudents();
 /*Error: There is no instance of Student, rollNo
 cannot be accessed*/
}
```

GLOBAL VARIABLE VS. STATIC MEMBERS

Alternative to static member is to use global variable

Global variables are accessible to all entities of the program

- Against information hiding