CS217 OBJECT ORIENTED PROGRAMMING

Spring 2020

STATIC MEMBER FUNCTIONS

- **A static member function is a special member function, which is used to access only static data members**, any other normal data member cannot be accessed through static member function.

- Just like static data member, static member function is also a class function; it is not associated with any class object.

- We can access a static member function with class name, by using following syntax:

```
class_name:: function_name(perameter);
```

```
class Demo
{
	private:
		//static data members
		static int X;
		static int Y;

	public:
	//static member function
	static void  Print()
	{
		cout <<"Value of X: " << X << endl;
		cout <<"Value of Y: " << Y << endl;
	}
};

//static data members initializations
int Demo :: X =10;
int Demo ::Y =20;
```

```
int main()
{
	Demo OB;
	//accessing class name with object name
	cout<<"Printing through object name:"<<endl;
	OB.Print();

	//accessing class name with class name
	cout<<"Printing through class name:"<<endl;
	Demo::Print();

	return 0;
}
```

CONSTANT OBJECTS

- instantiated class objects can also be made constant by using the **const** keyword. Initialization is done via class constructors:

```
1  const Date date1;                    // initialize using default constructor
2  const Date date2(2020, 10, 16);      // initialize using parameterized constructor
3
```

- Once a const class object has been initialized via constructor, any attempt to modify the member variables of the object is disallowed, as it would violate the const-ness of the object.
  - This includes both changing member variables directly (if they are public), or calling member functions that set the value of member variables.

```
class Something
{
public:
    int m_value;

    Something(): m_value(0) { }

    void setValue(int value) { m_value = value; }
    int getValue() { return m_value ; }
};

int main()
{
    const Something something;        // calls default constructor

    something.m_value = 5;            // compiler error: violates const
    something.setValue(5);            // compiler error: violates const

    return 0;
}
```

CONSTANT MEMBER FUNCTIONS

- The const member functions are the functions which are declared as constant in the program.
- The object called by these functions cannot be modified.
- It is recommended to use **const** keyword so that accidental changes to object are avoided.
- A const member function can be called by any type of object. **Non-const functions can be called by non-const objects only**.

```
class Demo {
      int val;
   public:
     Demo(int x) { val = x; }

      int getValue() const { return val; }
};

int main() {
      const Demo d(28);
      Demo d1(8);
      cout << "The value using object d : "          << d.getValue();
      cout << "\nThe value using object d1 : "      << d1.getValue();

      return 0;
}
```

A SIMPLE FUNCTION CALL

```
void staticDemo()
{
   int val = 0;
   ++val;
   cout << “val = "  << val << endl;
}

int main()
{
   staticDemo();     // prints val = 1
   staticDemo();     // prints  val = 1
   staticDemo();     // prints val  = 1
}
```

STATIC LOCAL VARIABLES

```
void staticDemo()
{
   static int val = 0;
   ++val;
   cout << “val = "  << val << endl;
}

int main()
{
   staticDemo();     // prints val = 1
   staticDemo();     // prints  val = 2
   staticDemo();     // prints val  = 3
}
```

TYPE CONVERSION

- The process of converting a value from one data type to another is called a **type conversion.** Type conversions can happen in many different cases:

```
double d{ 3 };// initialize double variable with integer value 3
      d = 6;// assign double variable the integer value 6
```

OR

```
void doSomething(long l){ }
doSomething(3); // pass integer value 3 to a function expecting a lo
ng parameter
```

OR

```
double division{4.0/3}; // division with a double and an integer
```

IMPLICIT TYPE CONVERSION (COERCION)

- **Implicit type conversion** (also called **automatic type conversion** or **coercion**) is performed whenever one data type is expected, but a different data type is supplied. If the compiler can figure out how to do the conversion between the two types, it will.
- All of the above examples are cases where implicit type conversion will be used.
- Whenever a value from one fundamental data type is converted into a value of a larger fundamental data type from the same family, this is called a **numeric promotion** (or **widening**, though this term is usually reserved for integers).

```
long l{64 }; // widen the integer 64 into a long
double d{0.12f }; // promote the float 0.12 into a double
```

- When we convert a value from a larger type to a similar smaller type, or between different types, this is called a **numeric conversion.** Unlike promotions, which are always safe, conversions may or may not result in a loss of data

```
double d{ 3 }; // convert integer 3 to a double (between different types)
short s{ 2 }; // convert integer 2 to a short (from larger to smaller type within same ty
pe family)
```

EXPLICIT TYPE CONVERSION (CASTING)

- When you want to promote a value from one data type to a larger similar data type, using implicit type conversion is fine.
- In C++, there are 5 different types of casts:

1. **C-style casts**
2. **static casts**
3. **const casts**
4. **dynamic casts**
5. and **reinterpret casts**.

- The latter four are sometimes referred to as **named casts**.

C-STYLE CASTS

```
int i1 = 10;
int i2 = 4;
float f = (float) i1 / i2; //Casting
```

- C++ will also let you use a C-style cast with a more function-call like syntax:

```
int i1 = 10;
int i2 = 4;
float f = float(i1) / i2;
```

STATIC_CAST

- C++ introduces a casting operator called **static_cast**, which can be used to convert a value of one type to a value of another type.

```
char c { 'a' };

std::cout << c << ' ' << static_cast<int>(c) << '\n'; // prints a 97
```

- The **static_cast** operator takes a single value as input, and outputs the same value converted to the type specified inside the angled brackets.
- Static_cast is best used to convert one fundamental type into another.

```
int i1 { 10 };
int i2 { 4 };

// convert an int to a float so we get floating point division rather than integer division
float f { static_cast<float>(i1) / i2 };
```

- Compilers will often complain when an *unsafe implicit type conversion* is performed. For example, consider the following program:

```
int i { 48 };
char ch = i; // implicit conversion
```

- Casting an int (4 bytes) to a char (1 byte) is potentially unsafe (as the compiler can't tell whether the integer will overflow the range of the char or not), and so the compiler will typically complain.
- To get around this, we can use a static cast to explicitly convert our integer to a char:

```
int i { 48 };
// explicit conversion from int to char, so that a char is assigned to variable
char ch = static_cast<char>(i);
```