CS217 OBJECT ORIENTED PROGRAMMING

Spring 2020

COPY CONSTRUCTOR

- A copy constructor is used to initialize an object using another object of the same class

- A copy constructor has the following prototype:

ClassName (const ClassName &ob);

COPY CONSTRUCTOR

- If we don't define our own copy constructor, the compiler creates a default copy constructor for each class
- The default copy constructor performs member-wise copy between objects
- Default copy constructor works fine unless an object has pointers or any runtime allocation

COPY CONSTRUCTOR

In C++, a Copy Constructor may be called when:

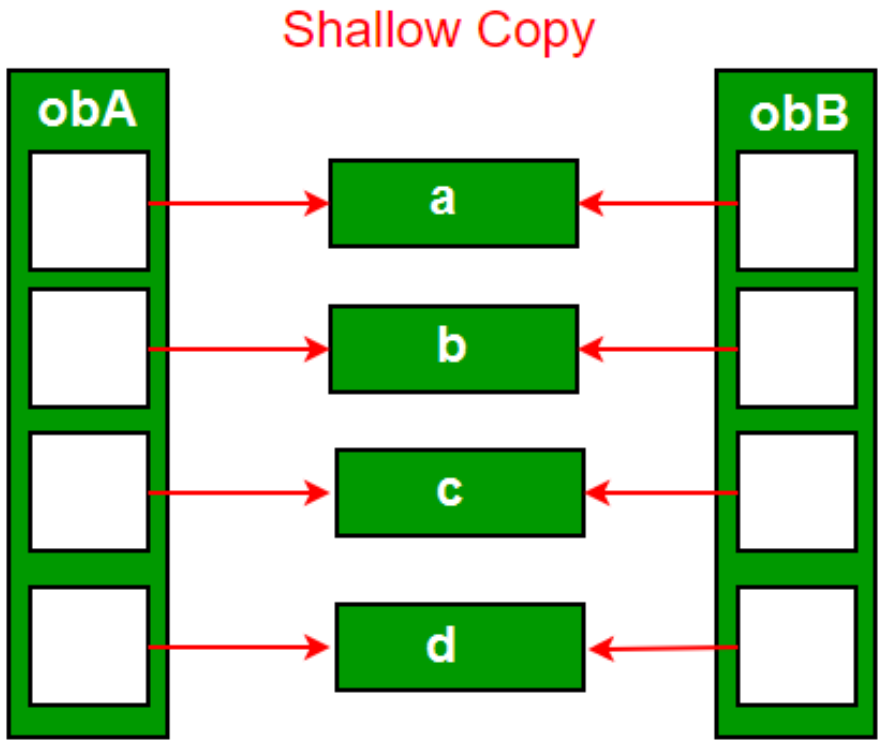
1) An object of the class is returned by value
2) An object of the class is passed (to a function) by value as an argument
3) An object is constructed based on another object of the same class
4) The compiler generates a temporary object

SHALLOW COPY

- Default constructor always perform a *shallow copy*

- Changes made by one object are also made for the other object

SHALLOW COPY

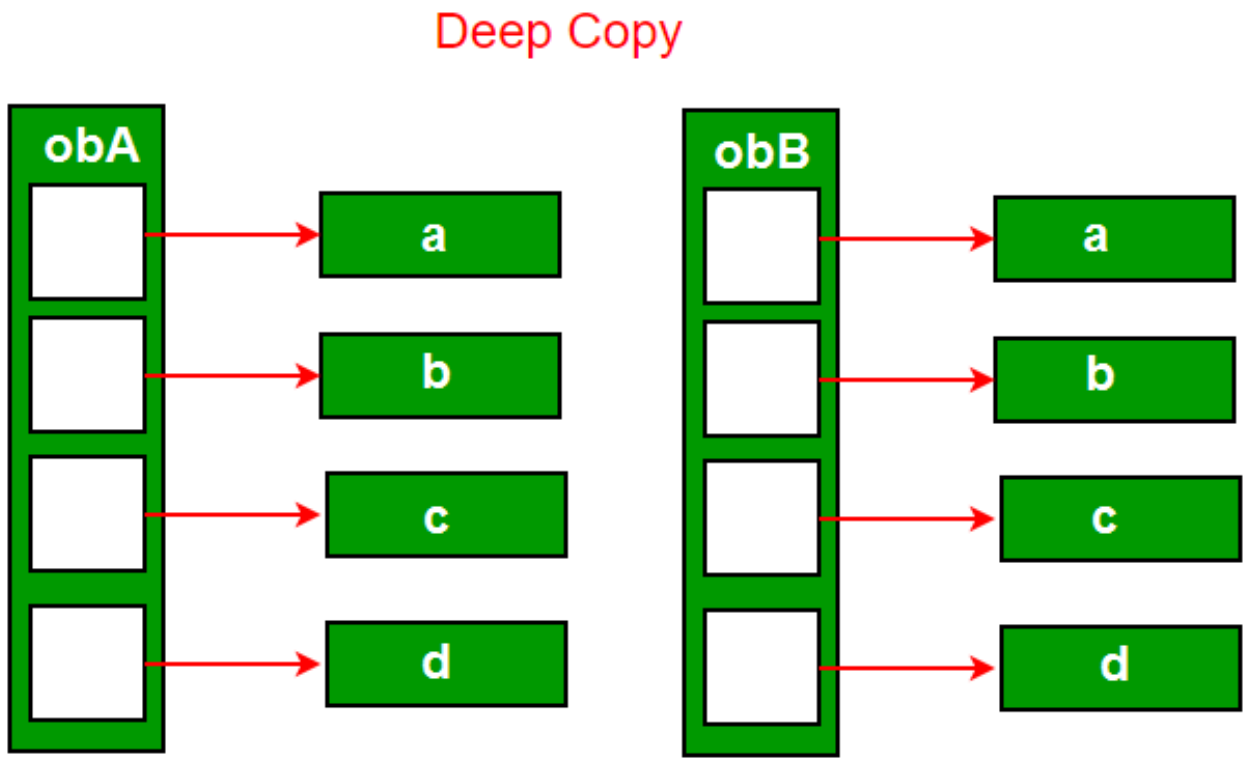


DEEP COPY

- Deep copy is only possible with user-defined copy constructors

- In user-defined copy constructors, we make sure that pointers (or references) of copied object point to new memory locations

DEEP COPY



```
class A
{
  char *s; int size;
  public:

  A(const char *str = NULL)
  {
          size = strlen(str);
          s = new char[size+1];
          strcpy(s, str);
  }

  A(const A& ob)
  {
                  size = ob.size;
                  s = new char[size+1];
          strcpy(s, ob.s);

  ~A( ) { delete [] s; }

  void print() {cout << s <<
  endl;}

  void change(const char
  *str)
  {
        delete [] s;
        size = strlen(str);
        s = new char[size+1];
        strcpy(s, str);
  }

  };
```

```
int main()
{
  A a1("Old string");
  A a2 = a1;

  a1.print();
  a2.print();

  a2.change("New string");

  a1.print();
  a2.print();
}
```

OUTPUT:

Old string *a1.s*
Old string *a2.s*

Old string *a1.s*
New string *a2.s*

WHY COPY CONSTRUCTOR?

- If we remove copy constructor from the above program, we don't get the expected output

- The changes made to a2 reflect in a1 as well which is never expected.

```
int main()
{
// with no copy constructor
  A a1("Old string");
  A a2 = a1;

  a1.print();
  a2.print();

  a2.change("New string");

  a1.print();
  a2.print();
}
```

OUTPUT:

Old string *a1.s*
Old string *a2.s*

New string *a1.s*
New string *a2.s*

DISCUSSION

- Can we make copy constructor private?
- Why argument to a copy constructor must be passed as a reference?
- Why argument to a copy constructor should be const?

CONST ARGUMENT FOR COPY CONSTRUCTOR

```
class Test
{
/* Class data members */
public:
Test(Test &t) { /* Copy data members from t*/}
Test()    { /* Initialize data members */ }
};
```

CONST ARGUMENT FOR COPY CONSTRUCTOR

```
Test func()
{
   cout << "func() Called\n";
   Test t;
   return t;
}

int main()
{
   Test t1;
   Test t2 = func();      \\Error at this line
   return 0;
}
```

SOLUTIONS

- Solution 1: Modify copy constructor:

```
Test(const Test &t) { /* Copy data members*/}
```

- Solution 2: Or do this (overloaded assignment operator):

```
Test t2;
t2 = func();
```

CONSTANT VARIABLES

- The keyword ***const*** be used to declare constant variables
- They must be initialized when they are declared and cannot be modified later
- Using constant variables to specify array size makes program more *scalable*
- Constant variables are also called ***named constants*** or ***read-only variables***

CONSTANT VARIABLES

```
int main()
{
  const int a = 5;
  const int b;       // will cause error
  b = 10;            // will cause error

  const int arr[] = {1, 2, 3, 4, 5};
  arr[0] = 10;       // will cause error
}
```

CONSTANT PARAMETERS

```
int func(const int a, int b)
{
  a += 10;                    //will cause error
  b += 20;
}
```

Principle of Least Privilege???

CONSTANT WITH POINTERS

- There are four ways to use **const** with pointers:
  - **Non-constant pointers to non-constant data**
  - **Non-constant pointers to constant data**
  - **Constant pointers to non-constant data**
  - **Constant pointers to constant data**

NON-CONSTANT POINTERS TO NON-CONSTANT DATA

- The highest access is granted by a **non-constant pointer to non-constant data**

- Data can be modified through pointer, and pointer can be made to point to other data

NON-CONSTANT POINTERS WITH NON-CONSTANT DATA

```
int main()
{
  int a = 10;
  int b = 50;

  int* pA = &a;
  *pA = 20;
  pA = &b;
}
```

NON-CONSTANT POINTERS TO CONSTANT DATA

- Pointer can be modified to point to any other data, but the data to which it points cannot be modified through that pointer

```
const  int * pVal;
```

NON-CONSTANT POINTERS TO CONSTANT DATA

```
int main()
{
  int a = 10;
  int b = 50;

  const int* pA = &a;
  *pA = 20;                    // this line will cause error
  pA = &b;
}
```

CONSTANT POINTERS TO NON-CONSTANT DATA

- Always points to the same memory location, but the data at that location can be modified through the pointer

```
int * const pVal = &val;
```

CONSTANT POINTERS TO NON-CONSTANT DATA

```
int main()
{
  int a = 10;
  int b = 50;

  int* const pA = &a;
  *pA = 20;
  pA = &b;                    // this line will cause error
}
```

CONSTANT POINTERS TO CONSTANT DATA

- Always points to the same memory location, and the data at that location cannot be modified via the pointer

```
const int * const pVal = &val;
```

CONSTANT POINTERS TO CONSTANT DATA

```
int main()
{
  int a = 10;
  int b = 50;

  const int* const pA = &a;
  *pA = 20;              // cannot do this
  pA = &b;               // cannot do this as well
}
```

INLINE FUNCTIONS

- C++ provides an inline functions to reduce the function call overhead.
- Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call.
- This substitution is performed by the C++ compiler at compile time.
- Placing the qualifier inline before a function's return type in definition **"advises"** the compiler to generate a copy of the function's code in place **(when appropriate)** to avoid a function call.

INLINE FUNCTIONS

- 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.

- Compiler usually ignores this request unless the function does not have too much code

INLINE FUNCTIONS

```
inline void square(int a)
{
  cout << "Square of given number is: " << a * a;
}

int main()
{
  int a = 2;
  square(a);
}
```

ADVANTAGES OF INLINE FUNCTIONS

1) Function call overhead doesn't occur
2) It also saves overhead of a return call from a function
3) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return

DISADVANTAGES OF INLINE FUNCTIONS

1) The added variables from the inline function consumes additional registers

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code

3) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled

4) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed

MEMBER INITIALIZATION LIST

- Constant class members can only be initialized through constructor's member initialization list

MEMBER INITIALIZATION LIST

```
class A
{
  const int x;
  const int y;
public:
  A ( int val1 , int val2 ) : x ( val1 ) , y ( val2)
  { }
};
```

```
int main()
{
  A a ( 5 , 10 );
}
```

STATIC CLASS DATA

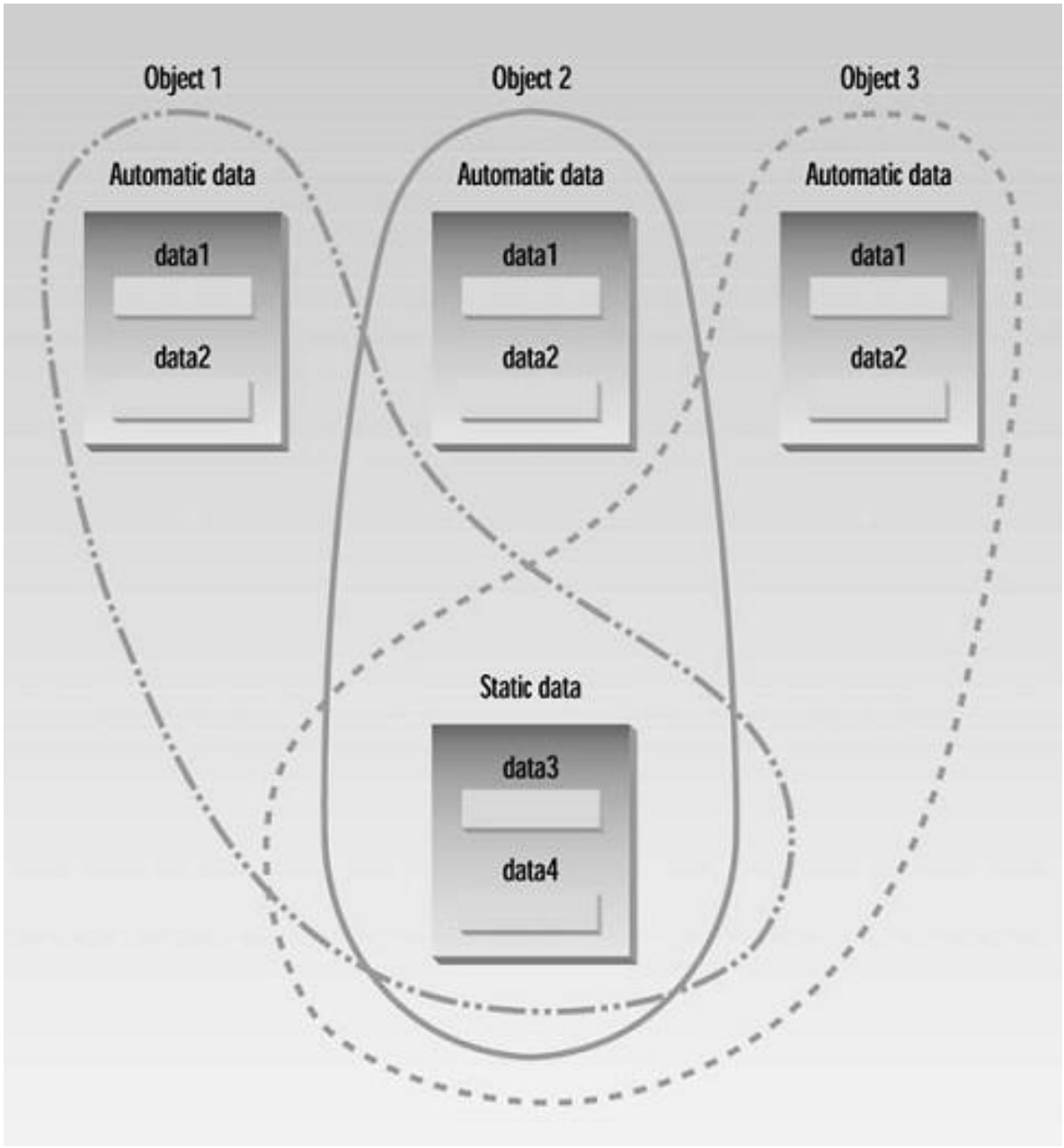
- If a data item in a class is declared as `static`, only one such item is created for the entire class, no matter how many objects there are.
  - useful when all objects of the same class must share a common item of information.
- A member variable defined as `static` is visible only within the class, but its lifetime is the entire program.
  - It continues to exist even if there are no objects of the class.
- `static` class member data is used to share information among the objects of a class.

```
class foo
   {
      static int count;   //only one data item for all objects
                          //note: "declaration" only!
public:
      foo()               //increments count when object created
         { count++; }
      int getcount()      //returns count
         { return count; }
   };
//--------------------------------------------------------------
int foo::count = 0;       //*definition* of count
//////////////////////////////////////////////////////////////
int main()
   {
   foo f1, f2, f3;        //create three objects

   cout << "count is " << f1.getcount() << endl;  //each object
   cout << "count is " << f2.getcount() << endl;  //sees the
   cout << "count is " << f3.getcount() << endl;  //same value
   return 0;
   }
```

- **class** `foo` **has one data item** `count`**, of type** `static int`**.**
- **constructor causes** `count` **to be incremented.**
- `main()` **define three objects of class** `foo`**.**
  - **Since the constructor is called three times,** `count` **is incremented three times.**
- **Another member function** `getcount()` **returns the value in** `count`**.**
- **If we had used an ordinary automatic variable—as opposed to a static variable—for** `count`**, each constructor would have incremented its own private copy of** `count` **once.**
- **The output would have been:**

```
count is 1
count is 1
count is 1
```


Object 1
Automatic data
data1
data2
Object 2
Automatic data
data1
data2
Object 3
Automatic data
data1
data2
Static data
data3
data4

- `static` member data requires two separate statements:
  - The variable's declaration appears in the class definition, but the variable is actually defined outside the class.

- If `static` member data were defined inside the class it would violate the idea that a class definition is only a blueprint and does not set aside any memory.

- Putting the definition of `static` member data outside the class also serves that the memory space for such data is allocated only once.

- One static member variable is accessed by an entire class.
  - Each object does not have its own version of the variable, as it would with ordinary member data

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;
}
```