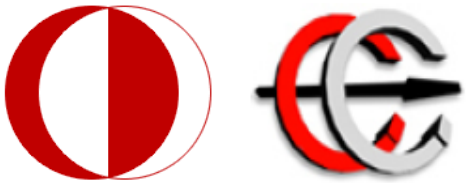


EE 441 Data Structures

Lecture 5:
Classes and Dynamic Memory

Memory Allocation

- A source code statement declares a variable or object
- The compiler creates information that specifies the amount of memory the variable or object will occupy.
- Example: An object from person class occupies 8 Bytes (4 Bytes for gender, 4 Bytes for Age)
- The compiler creates an executable file → memory requirements for all objects are defined

Static Memory Allocation

- When a program block is activated, i.e., procedure called, object created etc., memory is allocated for its data items
  - Local variables, input parameters (or their addresses) are copied into the local data areas
- Global data areas are allocated upon start of execution before they are used in the program.
  - The size to be allocated is determined during compile time (e.g., 1 word for short int, 2 words for long int, as much as required for array size, etc.)
- Upon exit, the local data areas are “freed”, i.e., returned to system memory manager

Static Memory Allocation

- When you compile your code, the compiler can examine primitive data types and calculate how much memory they will need ahead of time.
- The required amount is then allocated to the program in the memory stack space.
- For example, consider the following declarations:

```
int x[10];
double m;
```

- The compiler can immediately see that the code requires

```
10*size(int) + size(double) Bytes
```

- The compiler will insert code that will interact with the operating system to request the necessary number of bytes on the memory stack for your variables to be stored.
- Allocation is from the stack area of the memory at the beginning of the execution

Function Calls: Call Stack

- Special Memory Segment that holds memory addresses (Operating System)
- Stores Stack Frames with function data
- When the application starts, the main() function is pushed on the call stack by the operating system. Then the program begins executing.

http://www.learncpp.com/cpp-tutorial/79-the-stack-and-the-heap/

Function Calls: Call Stack

- Stack frame:
  - Return address: where CPU will return after function call is finished
  - All function arguments
  - Memory for local variables
  - Saved copies of registers to restore after function returns
- When function is called:
  - Stack frame is constructed and pushed on call stack
  - Static Memory for the local variables is allocated
  - The first function call is for main**()**
- When function returns:
  - Registers are restored
  - Stack frame is popped ➔ Frees the static memory for all local variables
  - Return value is handled

http://www.learncpp.com/cpp-tutorial/79-the-stack-and-the-heap/

Function Calls: Stack Memory

```
void test(int j)
{
        int k=j;
}


int main()
{
        int b[2]={10,20};

        test(10);
        int c=b[0];
        return 0;
}
```

`c, b[2]` are pushed onto the stack

`j` (copied, pass by value), `k` are pushed onto the stack

`j` (copied, pass by value), `k` are popped on return

`c, b[2]` are popped

Dynamic Memory Allocation

- During run-time a program might need additional memory
- Required Memory size can be determined by the input
- System memory manager makes additional memory available to the program:
- A program can allocate (borrow) memory from the memory heap and deallocate (return) memory to the memory heap when no longer needed in runtime
- Heap is another data structure similar to trees

http://www.learncpp.com/cpp-tutorial/79-the-stack-and-the-heap/

Memory Stack vs Memory Heap

- Different memory areas that can be allocated to the program
- Stack
  - Very fast access
  - don't have to explicitly de-allocate variables
  - space is managed efficiently by CPU, memory will not become fragmented
  - local variables only
  - limit on stack size (OS-dependent)
  - variables cannot be resized
- Heap
  - variables can be accessed globally
  - no defined limit on memory size (depends on the current state of the memory)
  - (relatively) slower access
  - no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
  - you must manage memory (you're in charge of allocating and freeing variables)
  - variables can be resized using realloc()

Memory allocation operator: new

- Declare a static allocated pointer `p` to an object of class `T`:

```
T  *p;
```

- Memory allocation for the pointer is static
- Allocate uninitialized memory for variable of type `T`:

```
p=new T;
```

- Memory is allocated for type `T` object and the address of the allocated memory is returned in `p`
- This statement is executed run time
- Object can have an initial value on the heap

```
p=new T(initvalue);
```

Memory allocation operator:

new

- Dynamically allocated memory can only be accessed through pointers
- Examples:

```
int  *p1 //p1 points to 1-word (4B) integer
long *p2 //p2 points to 2-word (8B) integer
```

  - Allocate space for 1-word integer and put its address in p1

```
p1 = new int;
p2 = new long;
```

Memory allocation operator: new

- If memory is not available the system returns value 0 → NULL pointer

```
double* dblPtr=new double(5.3);
if(dblPtr==NULL)
{       cerr << "Memory allocation failure!\n";
        exit(1);        }
```

- Dynamic memory allocation for objects

```
MyClass*ptr;
ptr=new MyClass(arguments);//pass the arguments to constructor
```

Memory allocation operator: new

- If memory is not available the system returns value 0 → NULL pointer

```
double* dblPtr=new double(5.3);
if(dblPtr==NULL)
{       cerr << "Memory allocation failure!\n";
        exit(1);        }
```

- Dynamic memory allocation for objects

```
MyClass*ptr;
ptr=new MyClass(arguments);//pass the arguments to constructor
```

- Note: Default constructor:
  - The constructor which takes no arguments is called the default constructor.
  - The compiler will generate its own default constructor for a class provided that no other constructors have been defined by the programmer.
  - The compiler generated default constructor initializes the member variables of the class using their respective default constructors

Dynamic Array Allocation

- Problem:
  - We might not know the size of the array until runtime:

```
int size;
cin>>size;
int IntArr[size];
```

  - Will not work! We need to declare the size constant:

```
const int size=40;
int IntArr[size];
```

- Solution: Dynamic memory allocation for arrays:

```
T* ArrPtr;
ArrPtr = new T[n];
```

  - Requests memory from the heap for n objects of type T
  - The elements in the array are uninitialized
  - Default constructor is called for an array of objects
  - If the required memory is not available:

```
if(ArrPtr==NULL)
{cerr << "Memory allocation failure!\n";
exit(1);
}
```

Memory deallocation operator: `delete`

- Heap is a finite resource→ program should return the memory when it is not needed

```
T*p, *q;
p=new T;
q=new T[n];
```

- deallocate the object currently pointed by p:

```
delete p;
```

- deallocate the entire array currently pointed by q:

```
delete[] q;
```

Memory deallocation operator: `delete`

- An example for bad memory management:

```
T s;
T*p, *q;
p=new T;
q=new T[n];
q=&s;
```

- Allocated memory for `T[n]` is completely lost, it is useless and it still consumes heap resource
- A memory leak:
  - Allocated memory is not freed although it is never used again.
  - Usually occurs because objects become unreachable without being freed.
  - Repeated memory leaks cause the memory usage of a process to grow without bound and crash of the code

Example

```
void mdemo(int initval1, float
initval2)
{
    int *p1=new int;
    float *p2=new float[3];
    *p1=initval1;
    int i;
    for(i=0;i<3;i++)
        p2[i]=initval2*2;
    cout<<*p1<<" ";
    for(i=0;i<3;i++)
        cout<<p2[i]<<" ";
    cout<<endl;
    delete p1;
    delete []p2;
```

```
    p1=new int[3];
    p2=new float;
    *p2=initval2;
    for(i=0;i<3;i++)
        *(p1+i)=initval1;
    cout<<*p2<<" ";
    for(i=0;i<3;i++)
        cout<<*(p1+i)<<" ";
    cout<<endl;
    delete []p1;
    delete p2;
}

mdemo(4, 2.5);
```

```
4 5 5 5
2.5 4 4 4
```

Example modified and revisited

```
void mdemo2(int initval1, float
initval2)
{
    int size=3;
    cout<<size<<endl;
    int *p1=new int;
    float *p2= new float[size];
    *p1=initval1;
    int i;
    for(i=0;i<3;i++)
        p2[i]=initval2*2;
    cout<<*p1<<" ";
    for(i=0;i<size;i++)
        cout<<p2[i]<<" ";
    cout<<endl;
    delete p1;
    delete []p2;
}
```

```
int main ()
{
    int init1=4;
    float init2=2.5;
    int *z1;
    float *z2;
    mdemo2(init1, init2);
    z1=new int (init1*2);
    z2=new float(init2+1.5);
    mdemo2(*z1,*z2);
    delete z1;
    delete z2;

    return 0;
}
```

```
3
4 5 5 5
3
8 8 8 8
```

Pointers and References

- Consider the function with pass by address

```
void test (int * p)
{
        //LINES OF CODE
}
```

- Problem: Somebody calls `test` in `main()` as follows:

```
test(new int);
```

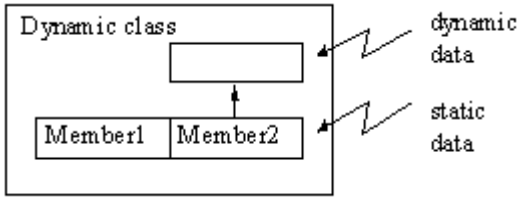
Where to call **`delete`**? What is the address of the allocated memory?

- Solution: Pass by reference is safer!

```
void test (int & p)
{
        //LINES OF CODE
}
```

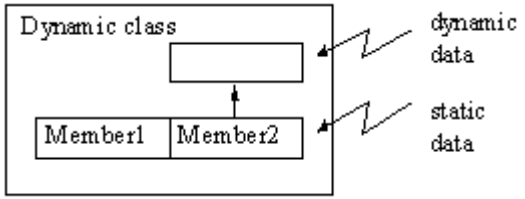
https://stackoverflow.com/questions/11854529/new-operator-in-function-parameter

Objects: Dynamic Memory Allocation




```
template <class T>
class DynamicClass
{
        private:
        // variable of type T and a pointer to data of type T
                T member1;
                T *member2;

        public:
        //constructor with parameters to initialize member data
                DynamicClass(const T &m1, const T &m2);
        //other methods to come…
};
```

Objects: Dynamic Memory Allocation




```
template <class T>
DynamicClass<T>::DynamicClass (const T &m1, const T &m2)
{
        // parameter m1 initializes static member
        member1=m1;
        //allocate dynamic memory and initialize it with value m2
        member2=new T(m2);
        cout<<"Constructor:"<<member1<<'/'<<*member2<<endl;
}
```

Example Run

- The following statements define a static variable `staticObj` and a pointer variable `dynamicObj`.
- The `staticObj` has parameters 1 and 100 that initialize the data members:

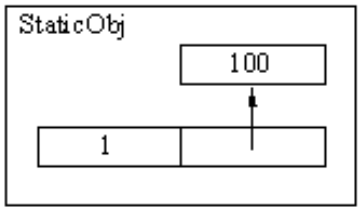
```
//DynamicClass object
DynamicClass<int> StaticObj(1,100);
```

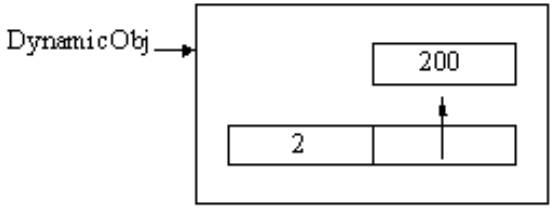
- In the following, the object `DynamicObj` points to an object created by the new operator. Parameters 2 and 200 are supplied as parameters to the constructor:

```
//pointer variable
DynamicClass<int>*DynamicObj;
//allocate an object
DynamicObj=new DynamicClass<int>(2,200);
```

Example Run

```
//Dynamic Class object
DynamicClass<int> StaticObj(1,100);
//pointer variable
DynamicClass<int>*DynamicObj;
//allocate an object
DynamicObj=new DynamicClass<int>(2,200);
```





Running the program results in

```
Constructor:1/100
Constructor:2/200
```

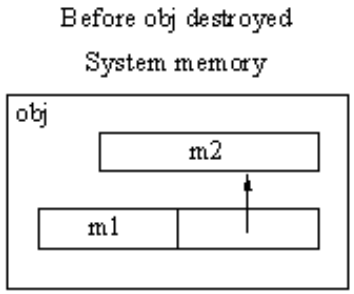
Objects: Deallocation of the Dynamic Memory

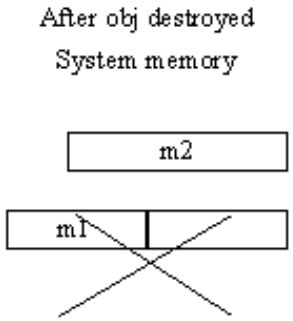
- Example: Consider a function that creates a `DynamicClass` object having integer data

```
void DestroyDemo(int m1,int m2)
{
    DynamicClass<int> obj(m1,m2);
}
```

- Dynamic data still remains in the system memory.
- For effective memory management, we need to deallocate the dynamic data within the object at the same time the object being destroyed.
- We need to reverse the action of the constructor, which originally allocated the dynamic data.

Upon return from `DestroyDemo` `obj` is destroyed; however the process does not deallocate the dynamic memory associated with the object





Objects: Deallocation of the Dynamic Memory

- The C++ language provides a member function, called the destructor, which is called by the compiler when an object is destroyed.
- Purpose: `delete` calls corresponding to the `new` calls in the Constructor
- For DynamicClass, the destructor has the declaration:

```
~ DynamicClass(void);
```

Objects: Deallocation of the Dynamic Memory

- `~ DynamicClass(void);`
- The character "`~` " represents "complement", so `~DynamicClass` is the complement of a constructor.
- A destructor never has a parameter or a return type.
- For our sample class, the destructor is responsible for deallocating the dynamic data for `member2`.

Objects: Deallocation of the Dynamic Memory

- Constructor:

```
template <class T>
DynamicClass<T>::DynamicClass (const T &m1, const T &m2)
{
        member1=m1;
        member2=new T(m2);
        cout<<"Constructor:"<<member1<<'/'<<*member2<<endl;
}
```

- Destructor:

```
template <class T>
DynamicClass<T>::~DynamicClass(void)
{
        cout<<"Destructor:"<<member1<<'/'<<*member2<<endl;
        delete member2;
}
```

When is Destructor called

- Whenever an object is deleted.
  - It is normally not necessary to call a destructor explicitly.
  - The destructor for an object is automatically invoked at the end of the lifetime of that object.
  - End of lifetime: When the object goes out of scope→ function returns, main ends, `delete` is called for dynamically allocated objects
- Default Destructor:
  - If the programmer does not define a destructor, the C++ compiler generates a default destructor.
  - The default destructor removes the memory space allocated to static data members.

When is Destructor called

- When a program terminates, all global objects or objects declared in the main program are destroyed.
  - For local objects created within a block, the destructor is called when the program exits the block.
  - The destructors for global variables run after the main routine exits

Example

```
void DestroyDemo(int m1,int m2)
{
        DynamicClass<int> Obj(m1,m2);
}

int main ()
{
        DynamicClass<int> StaticObj(1,100);
        DynamicClass<int>*DynamicObj;
        DynamicObj=new DynamicClass<int>(2,200);
        DestroyDemo(3,300);
        delete DynamicObj;
        return 0;
}
```

Example

Run
Constructor: 1/100
Constructor: 2/200
Constructor: 3/300
Destructor: 3/300
Destructor: 2/200
Destructor: 1/100

```
void DestroyDemo(int m1,int m2)
{
        DynamicClass<int> Obj(m1,m2);
}
```

← Constructor for `Obj(3,300)`

← Destructor for `Obj`

```
int main ()
{
        DynamicClass<int> StaticObj(1,100);
        DynamicClass<int>*DynamicObj;
        DynamicObj=new DynamicClass<int>(2,200);
        DestroyDemo(3,300);
        delete DynamicObj;
        return 0;
}
```

← Constructor for `StaticObj(1,100)`

← Constructor for `*DynamicObj(2,200)`

← Destructor for `DynamicObj`

← Destructor for `StaticObj`

```
Constructor:1/100
Constructor:2/200
Constructor:3/300
Destructor:3/300
Destructor:2/200
Destructor:1/100
```

Memory allocation/deallocation at C style

- C style:

```
int* ip;
ip =(int*)malloc(sizeof(int) * 100);
free((void*)ip);
```

- With new/delete in C++, we would have:

```
int* ip;
ip = new int[100];
delete ip;
```

`this`

- Each C++ object has a pointer named `this`
- Automatically defined when the object is created
- Returns a reference to the current object in a class member function
- Each object is created with an extra memory area that holds the address of the object

this

```
template <class T>
DynamicClass<T>* DynamicClass<T>::Address(void)
{
      return this;
}

      DynamicClass<int> Obj5(1,100);
      DynamicClass<int> *ptr=Obj5.Address();
      //same as:
      //DynamicClass<int> *ptr=&Obj5;
```

Accessing Objects with Pointers

```
template <class T>
void DynamicClass<T>::GetMembers (void)
{
    cout<<"Members:"<<member1<<'/'<<*member2<<"\n";
}
```

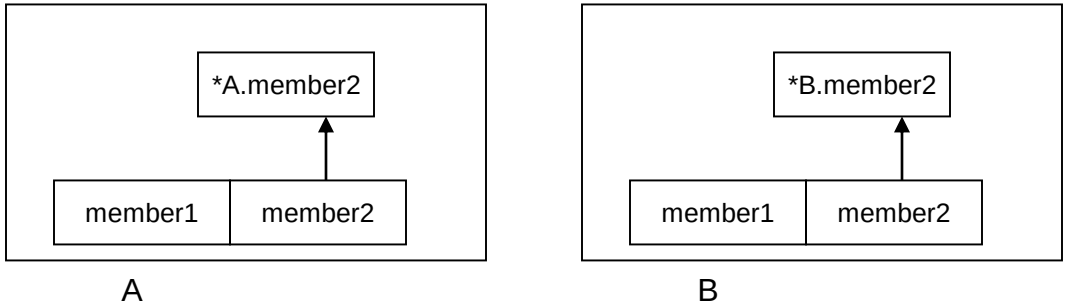
```
Constructor:5/500
Members:5/500
Constructor:6/600
Members:6/600
Destructor:5/500
```

```
DynamicClass<int> Obj5(5,500);
DynamicClass<int> *ptr=&Obj5;
ptr->GetMembers();
DynamicClass<int>* ptr6=new DynamicClass<int> (6,600);
ptr6->GetMembers();
```

Assignment Operation

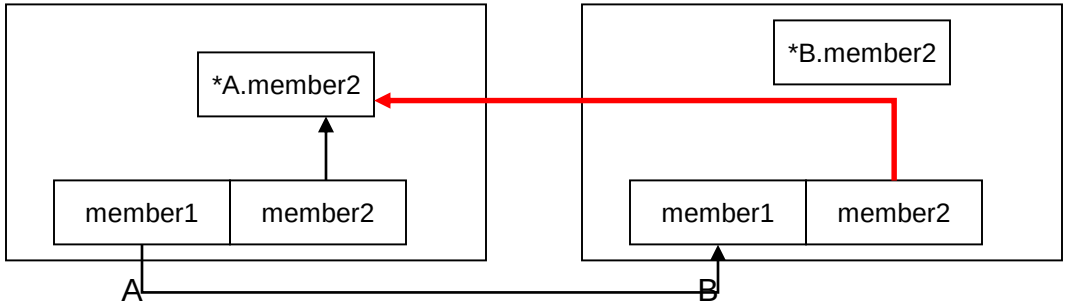
- The standard assignment Y=X
  - Object on left hand side already exists
  - a bitwise copy of the data from object X to the data in object Y
- Special consideration must be used with dynamic memory so that unintended errors are not created.
- We must create new methods that handle object assignment and initialization.

Problem in Assignment




```
// initialization
DynamicClass A(20,50), B(40,30);
//creates DynamicClass objects A and B
```

Creation and
Initialization of A and B



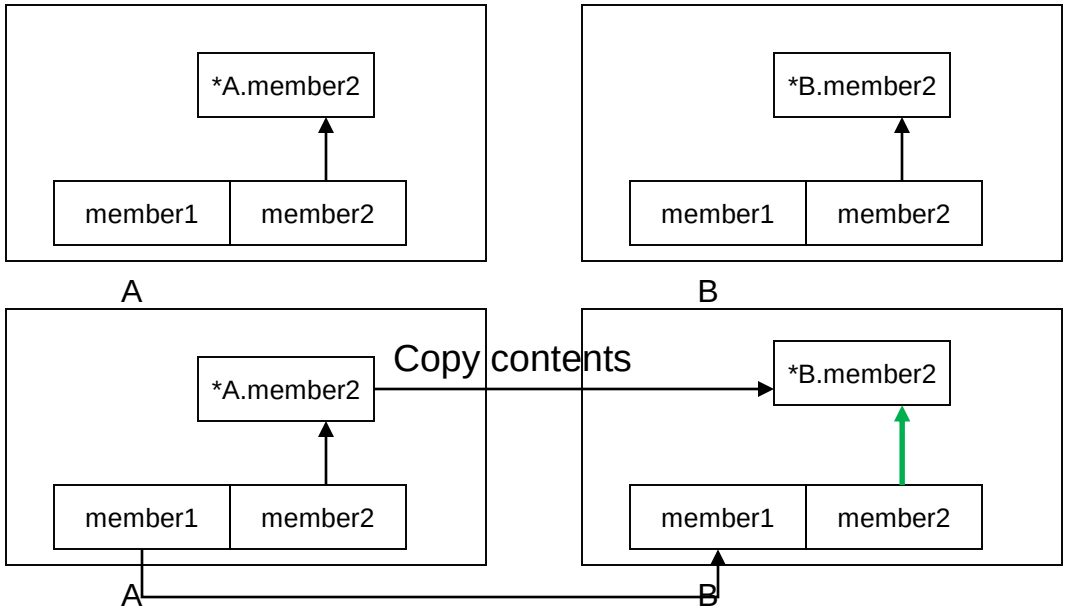

```
// assignment
B=A;
//data in A is overwritten by data in B
```

B=A
(Wrong)

What happened

- The assignment statement of B=A causes the data in A to be copied to B:
  - member1 of B=member1 of A: copies static data from A to B
  - member2 of B=member2 of A: copies pointer from A to B
- Problem after assignment:
  - Pointers of objects A and B reference the same location
  - The dynamic memory originally assigned to B is unreferenced
  - Destructors are called for each object
  - Try to deallocate the same memory twice
- Solution:
  - Instead of copying the pointers copy the contents

What do we want



Creation and Initialization of A and B

B=A

(Correct)

Overloading the assignment Operator

- Overload the assignment operator “=“ as a member function
- Explicitly assigns all data members (not just bitwise copy)

```
DynamicClass<T>& operator= (const DynamicClass <T>& rhs)
```

- const reference: (right-hand side) rhs object is not altered
- `B=A;//means B.=(A);`
- B is the current object
- A is the rhs object

Overloading the assignment Operator

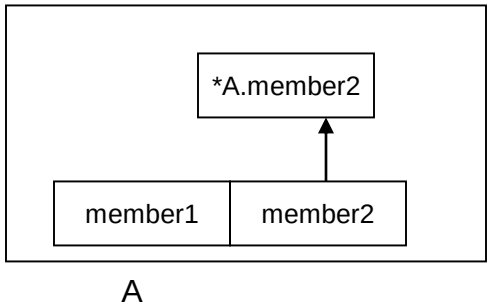
```
// Overloaded assignment operator = returns a reference to the
//current object
template <class T>
DynamicClass<T>& operator= (const DynamicClass <T>& rhs)
{
    //copy static data member from rhs to the current object
    member1=rhs.member1;
    // content of the dynamic memory must be same as that rhs
    *member2=*rhs.member2;
    cout <<"Assignment Operator: "<<member1<<'/'<<*member2<<endl;
    return *this;
}
```

`this` in assignment operator

- `B=A;//means B.=(A);`
- B is the current object
- A is the rhs object
- = returns a reference to the current object
- After calling assignment, B can be used as a rhs object for another object C
- Enables: C=B=A

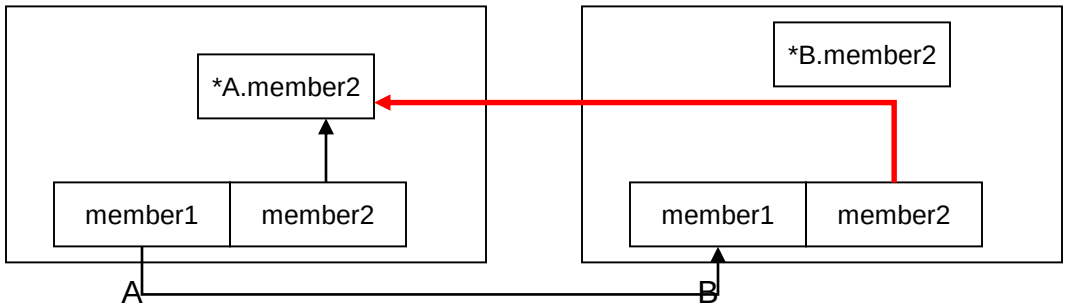
Problem in Initialization

- Initialization creates a new object that is a copy of another object.



Creation and Initialization of A

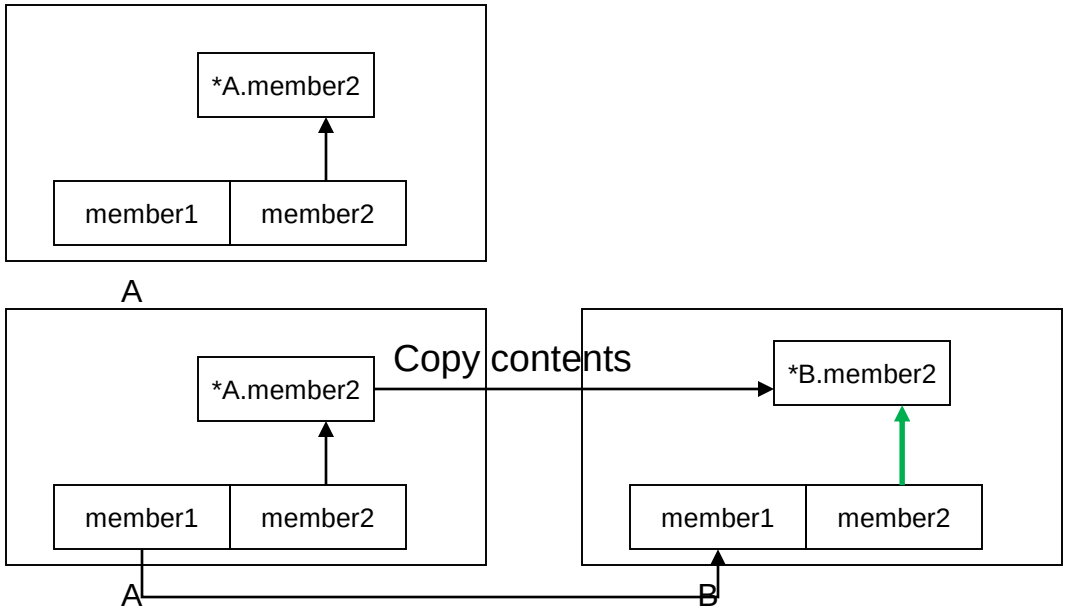
```
// initialization

DynamicClass A(20,50), B=A;

//creates DynamicClass
//objects A and B

// data in B is initialized
//by data in A
```



Creation of B and Initialization of B with A

B=A (Wrong)

What do we want



Creation of A

Creation and Initialization of B(Correct)

Copy Constructor

- The copy constructor is a member function
- It is a constructor:
  - Declared with the class name
  - It does not have a return value.
- It is used when:
  - An object is used to initialize another object in declaration
  - An object is passed as parameter to a function by value
  - A temp object is created for use as return value by a function
- Usage:

```
DynamicClass<int> A(3,5), B=A;
```

Constructor for A Copy Constructor for B

Copy Constructor

```
//copy constructor: initialize new object to have the same
//data as obj.
template <class T>
DynamicClass<T>::DynamicClass(const DynamicClass<T>& obj)
{
    // copy static data member from obj to current object
    member1=obj.member1;
    //allocate dynamic memory and initialize it with value
    //*obj.member2
    member2=new T(*obj.member2);
    cout<<"Copy Constructor:"<<member1<<'/'<<*member2<<endl;
}
```

Example

```
#include <iostream>
using namespace std;
#include "DynamicClass.hpp"

template <class T>
DynamicClass<int> Demo(DynamicClass<T> one, DynamicClass<T>& two, T m)
{
    DynamicClass<T> obj(m,m);
    return obj;
}

void main()
{
    DynamicClass<int> A(3,5), B=A, C(0,0);
    C=Demo(A,B,5);
}
```

Example

```
void main()
     { DynamicClass<int> A(3,5), B=A, C(0,0);
        C=Demo(A,B,5);
     }
```

Constructor: 3/5 // construct A
Copy Constructor: 3/5 // construct B
Constructor: 0/0 // construct C

Example

Demo is called:

```
C=Demo(A,B,5);
```

```
DynamicClass<int> Demo(DynamicClass<T> one, DynamicClass<T>& two, T m)
    { DynamicClass<T> obj(m,m);
       return obj;
    }
```

Copy Constructor: 3/5 // construct one (as a copy of A)

Constructor: 5/5 // construct obj

Copy Constructor: 5/5 // construct return object for Demo

Example

```
C=Demo(A,B,5);

DynamicClass<int> Demo(DynamicClass<T> one, DynamicClass<T>&
two, T m)
    { DynamicClass<T> obj(m,m);
       return obj;
    }
```

Demo is returning:

Assignment Operator: 5/5 // assign return object of Demo to C

Destructor: 5/5 // destruct obj upon return of Demo

Destructor: 3/5 // destruct one (copy of A) upon return of Demo

Destructor: 5/5 // destruct return object of demo

While returning from a function, destructor is the last method to be executed.

Example

`main` is returning

```
void main()
    { DynamicClass<int> A(3,5), B=A, C(0,0);
      C=Demo(A,B,5);
    }
```

Destructor: 5/5 // destruct C

Destructor: 3/5 // destruct B

Destructor: 3/5 // destruct A

Example

```
#include <iostream>
using namespace std;
#include "DynamicClass.hpp"

template <class T>
DynamicClass<int> Demo(DynamicClass<T> one,
DynamicClass<T>& two, T m)
{
    DynamicClass<T> obj(m,m);
    return obj;
}

void main()
{
    DynamicClass<int> A(3,5), B=A, C(0,0);
    C=Demo(A,B,5);
}
```

Why didn't our copy constructor get called for the return object?

Note that initializing an anonymous object and then using that object to direct initialize our defined object takes two steps (one to create the anonymous object, one to call the copy constructor). However, the end result is essentially identical to just doing a direct initialization, which only takes one step.

For this reason, in such cases, the compiler is allowed to opt out of calling the copy constructor and just do a direct initialization instead. This process is called **eliding**.

Running the program results in;

```
Constructor: 3/5 // construct A
Copy Constructor: 3/5     // construct B
Constructor: 0/0          // construct C
Copy Constructor: 3/5     // construct one
Constructor: 5/5          // construct obj
Copy Constructor: 5/5     // construct return object for Demo
Assignment Operator: 5/5  // assign return object of Demo to C
Destructor: 5/5           // destruct obj upon return
Destructor: 3/5           // destruct A upon return
Destructor: 5/5 // destruct return object of demo
Destructor: 5/5           // destruct C
Destructor: 3/5           // destruct B
Destructor: 3/5           // destruct A
```

GCC provides the `-fno-elide-constructors` option to disable copy-elision.
It is generally not recommended to disable this important optimization.

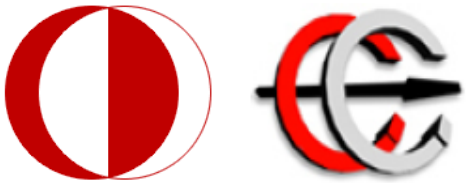
https://en.wikipedia.org/wiki/Copy_elision

http://www.learncpp.com/cpp-tutorial/911-the-copy-constructor/

Copy Constructor fatality

- The parameter in a copy constructor must be passed by reference.
- `DynamicClass(DynamicClass<T>& X)`

- What happens if we pass by value?
  `DynamicClass(DynamicClass<T> X)`
  - Call: `DynamicClass(DynamicClass <T>X)`
  - Since we pass A to X by value, the copy constructor must be called to handle the copying of A to X. This call in turn needs the copy constructor, and we have an infinite chain of copy constructor calls. In addition, the reference parameter X should be declared constant, since we certainly do not want to modify the object we are copying.



EE 441 Data Structures

Lecture 5:
Classes and Dynamic Memory