Important Concepts Learned in ENSF 337

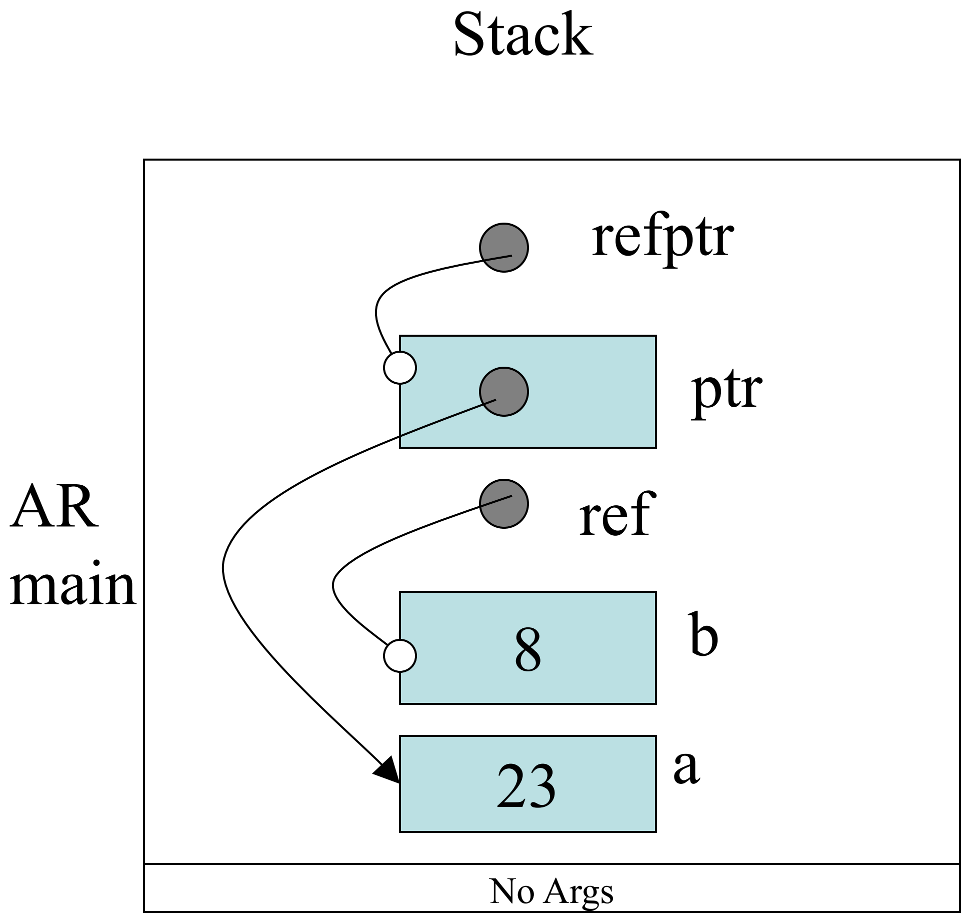
This set of slides are aimed to refresh your memory about some of the important concepts that we discussed in previous course

C++ Reference and Pointer types

Drawing AR diagrams for pointers and references in C++:

- Reference is simply an alias (alternative) for a variable name

```
 int main()
{
        int a , b;
        int& ref = b;
        int * ptr = &a;
         int* & refptr = ptr;
        *ptr = 4;
          ref = 8;
          *refptr = 23;
        …
}
```



Arrays, Pointers, and Pointer Arithmetic

Arrays and Pointers

- The name of an array is treated as a constant pointer that points to the first element of the array. Therefore, the array name and pointers of the same type have some similarities:

```
int a [6] = {4,2,3,1,8, 11};
cout << *a;                    // Using Pointer Notation
cout << a[0];                  // Using Array Notation
```

  - Both statements above print 4;

Pointer Arithmetic

- Legal pointer arithmetic in C++

  Pointer + Integer

  Pointer – Integer

  Pointer – Pointer

  Pointer++, or ++Pointer

  Pointer—, or --Pointer

- Other arithmetic operations are illegal. An operation like "Integer – Pointer" is not also allowed.

Pointer Arithmetic

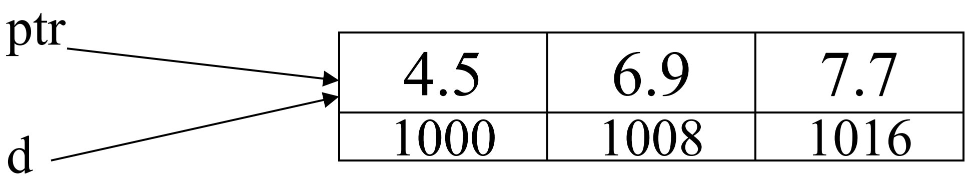
- "pointer + n" refers to the address of nth element , from the current address. In other words:

  pointer +n = current address + n * sizeof (type)

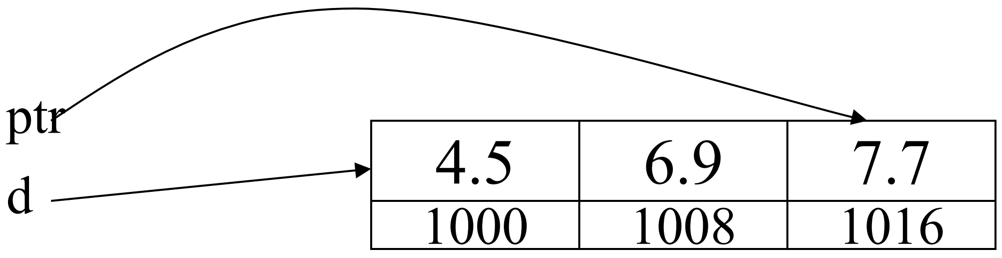
  E.g:

  double d[3] = {4.5, 6.9, 7.7};

  double* ptr = &d[0];



ptr = ptr + 2; // moves **ptr** to the address1000 + 2 * 8



Likewise, pointer –n refers to the current address – n * sizeof(type)

Pointer Arithmetic

- pointer1 – pointer2 refers to: address1 minus address2 divided by sizeof(type)
- In other words, "Pointer1 – Pointer2", results in an integer value that represents the number of elements-types between the two pointers:

```
int a[5] = {2, 6, 4, 7, 9};
int *ptr;
ptr = a+2;
int diff;
diff = ptr – a;
```

  – In this example the value of diff will be 2.

Copying Objects

Copying Object

- An instance of a class can be initialized with another instance of the same class:

```
Aclass  a1;
Aclass  a2 = a1;  // Initialization
Aclass  a3;
a1  = a3;          // Assignment
```

- Every data member of instance a1 will be copied into instance a2.

Copying Objects

- Consider the following C++ class definition:

```
class String {
        char *storageM;   // pointer to allocated memory on the heap
        int lengthM;         // represents length of string
        public:
        String(char *s);   // ctor
        String(const String& src);   // copy ctor
        String& operator =(String& rhs); // assignment op.
        ~String();   // dtor
        void display();
};
```

- **Details of the copy ctor and overloaded assignment operator will be reviewed during the lectures**

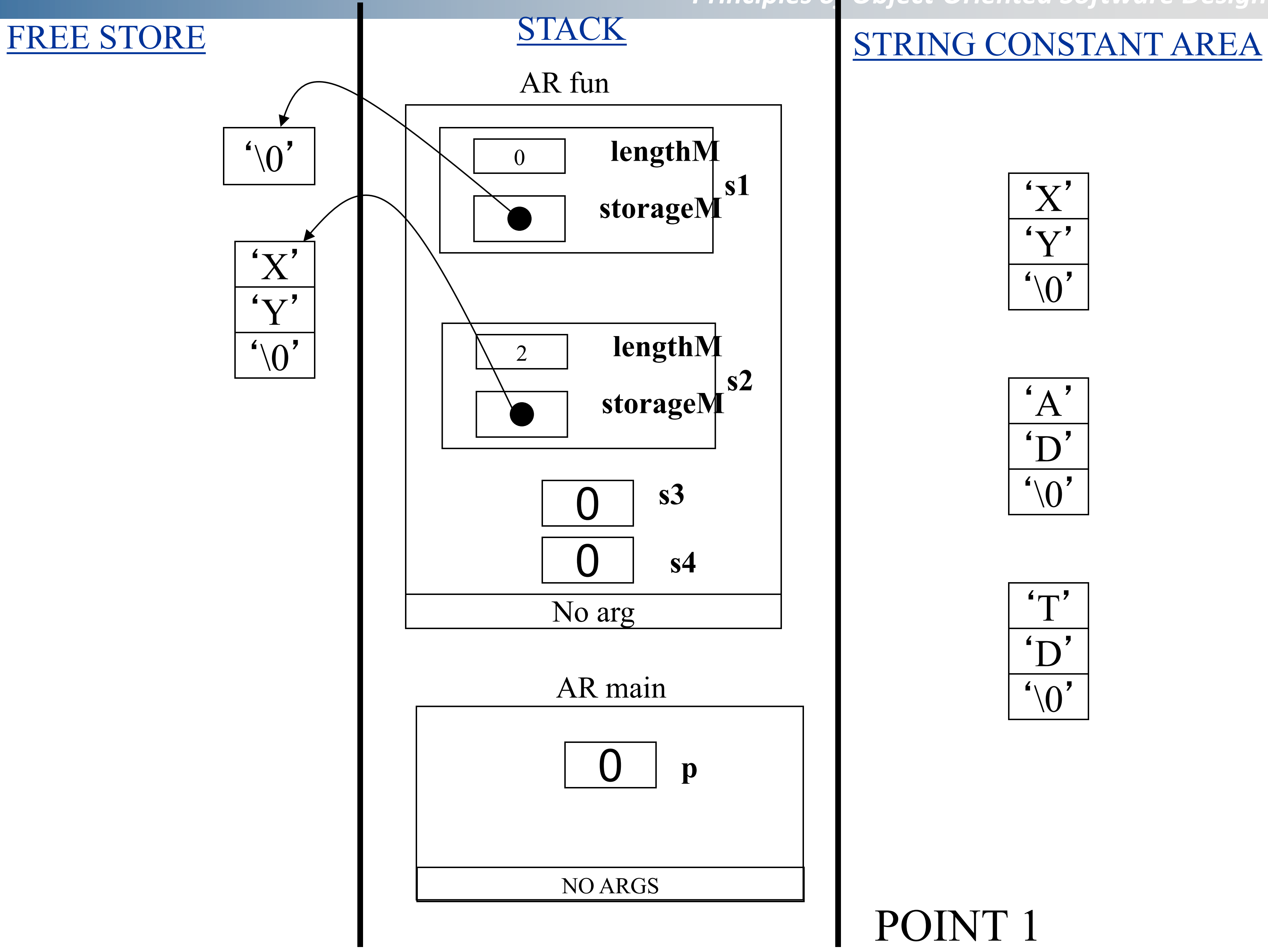
Dynamic Allocation of Objects in C++

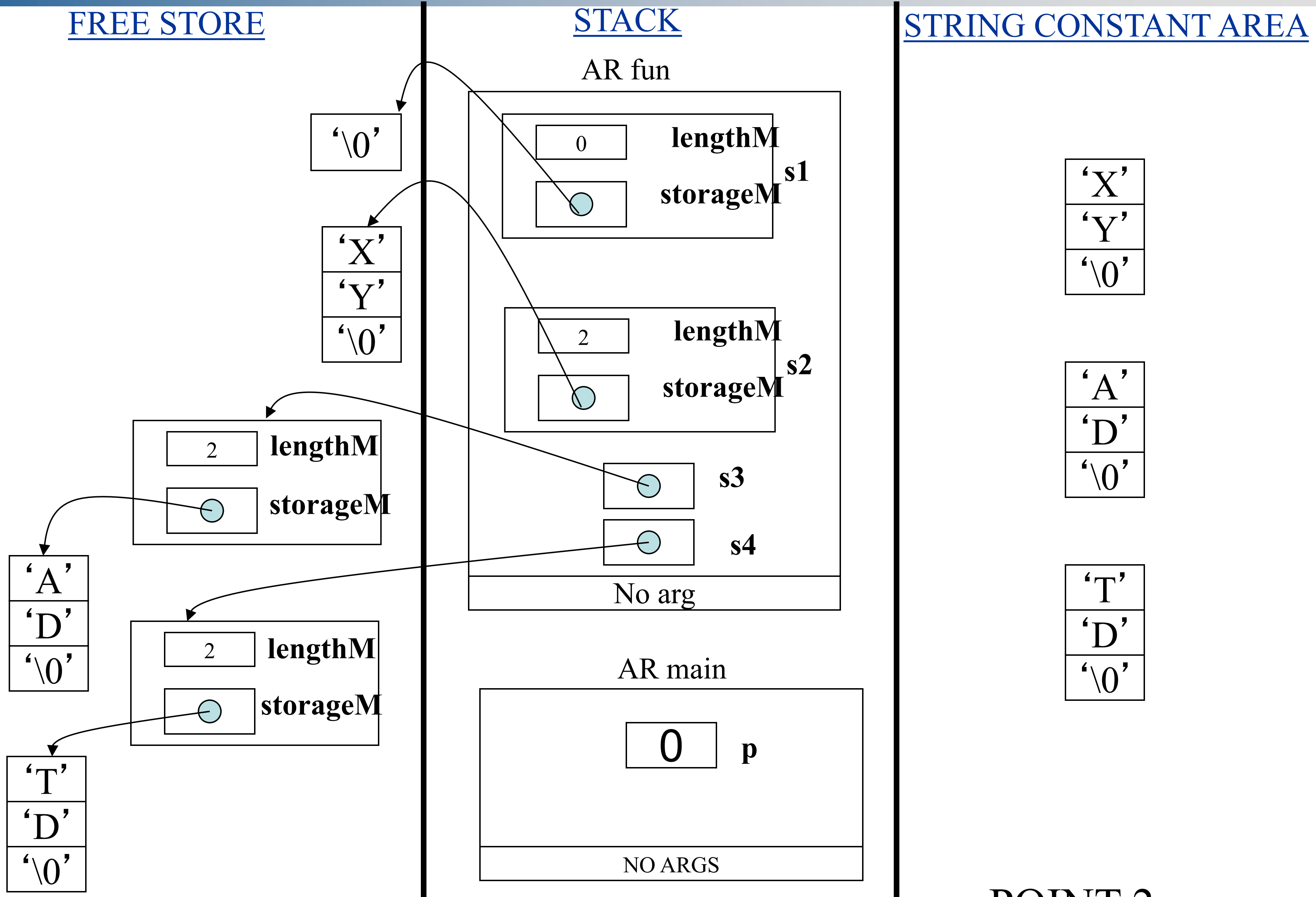
Dynamic Allocation of Objects

```
#include <iostream.h>
#include <string.h>
#include <assert.h>
void main()
{
     String * p = NULL;
     p = fun();
    // point 3
    delete p;
    // Point 4
    return 0;
}
```

```
String* fun()
{
        String s1;
        String s2(“XY”) ;
        String *s3 = NULL;
        String *s4 = NULL;
        // Point 1
        s3= new String (“AD”);
        s4= new String (“TD”);
        // Point 2
        return s3
}
```

AR diagrams at point 1 and 2 are given. Your job is to draw ARs for points 3 and 4 try to


FREE STORE
STACK
STRING CONSTANT AREA
AR fun
'\0'
0
lengthM
s1
storageM
'X'
'Y'
'\0'
2
lengthM
s2
storageM
0
s3
0
s4
No arg
AR main
0
p
NO ARGS
'X'
'Y'
'\0'
'A'
'D'
'\0'
'T'
'D'
'\0'
POINT 1


FREE STORE
STACK
STRING CONSTANT AREA
AR fun
'\0'
0
lengthM
storageM
s1
'X'
'Y'
'\0'
2
lengthM
storageM
s2
2
lengthM
storageM
s3
s4
'A'
'D'
'\0'
No arg
2
lengthM
storageM
AR main
0
p
'T'
'D'
'\0'
NO ARGS
'X'
'Y'
'\0'
'A'
'D'
'\0'
'T'
'D'
'\0'

POINT 2

Questions to be asked when designing a C++ class. “

- When do we need a destructor?
- When do we need assignment operator?
- When do we need copy constructor?
- When do we need default constructor?
- When is constructor called?
- When is destructor called?
- What is the law of Big 3?

- The answers to this questions have discussed in ENSF 337, and we will review them during the lectures in ENSF 480.

- Please look at the following example that uses objects of String class, and indicate: How many times constructor (ctor), destructor (dtor), assignment operator, default constructor (default ctor), and copy constructor are called:

```
int main(void) {
        String s1("ABC");
        String s2("XY");
        {
                String s3 ("KLM");
                String *s4;
                s4 = new String("BAR");
                String s5 =s1;
                s3 = s2;
                String s6[2];
                delete s4;
                //Point one
        }
        // point two
        String s7 = fun(s1, s2, &s1);
        S2 = fun(s1, s2, &s7);
        // point three
        Return 0
}
```

```
String fun (String x, String& y, String *z)
{
   MyString w;
  // Some code…

  return w;
}
```

Answers will be discussed during the lecture

Different Application of const Identifier in C++

Different usage of `const` keyword

- The const keyword might be used in different forms in C++. Here are some examples
  - Pointer to constant. Example:

```
const char* s= “ABCD”;        // s is pointing to a constant area
s[0] = ‘M’;                   // Illegal operation
s++;                                      // OK
```

  - Constant Pointer. Example:

```
char a[4] = “XYZ”;
char* const cp = a;    // cp is a constant pointer
cp++;                                     // Illegal operation
cp[0] = ‘M’;                              // OK
```

  - Constant Pointer to a constant

```
char a[4] = “XYZ”;
const char* const cpc = a;
cpc++;                                    // Illegal operation
cpc[0] = ‘M’;                              // Illegal operation
```

`const` Member Function and Member Functions that Return `const` Type

`const` Member Functions

- If a member function is supposed to be used as a Read-Only function, or simply the function is supposed to serve as a "getter", the function is better to be declared as a const member:

```
class Student
{
  public:
        Student(const char* &name, const int id);
        char* get_name() const; // read-only function
  private:
         char nameM[50];
         int idM;
};
```

Member Functions with `const` Return Type

- Sometimes, it is necessary to protect the values returned from member functions. If a function returns a pointer or reference to a member variable. Those cases may allow the program to change the value of a private data member (which defeats the purpose of information hiding). Let's have a close look the following example and find out what may go wrong with such a program.

```
class Student
{
    public:
      Student(const char* &name, const int id);
      char* get_name() const {return nameM;}
    private:
      char nameM[50];
      int idM;
};

Student::Student(const char* name, const int id)
{
   strcpy(nameM, name);
   idM = id;
}
```

Protecting Data Members

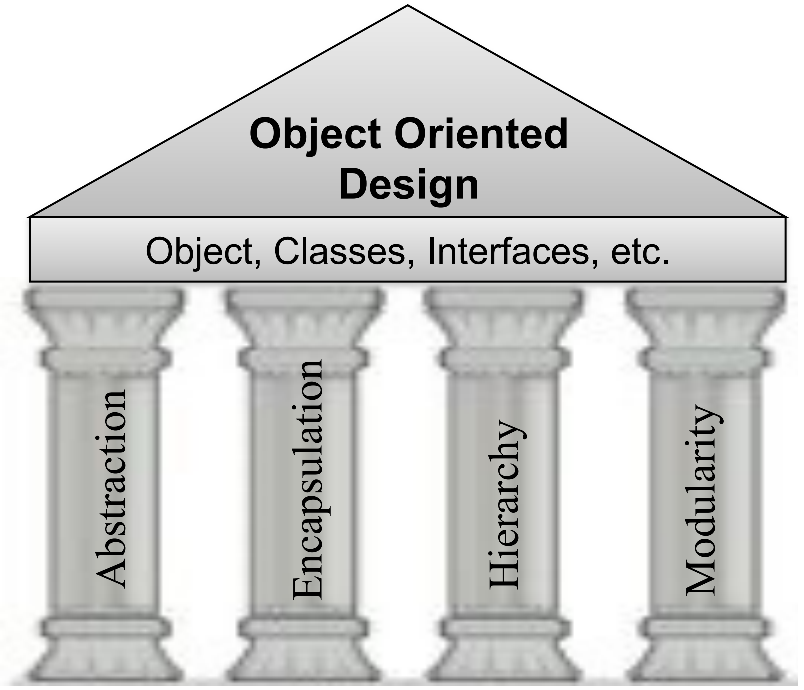
- Assuming that the above code compiles, consider the following code segment:

```
Student One(“Jane”,123456);
char* trouble = One.get_name();
trouble [0] = ‘P’;
```

- **What is wrong about this code, and what is this issue. Do your best to find out an answer.**
  - **During the lecture, the detail and the possible solution will be discussed.**

Pillars of Object-Oriented Design

- Four major elements of the object design includes:
  - Abstraction
  - Encapsulation/Information Hiding
  - Hierarchy
  - Modularity



Abstraction

- Abstraction is a technique of dealing with complex system. We make a simplified model of a complex system.
- Deciding upon the right set of abstractions for a given domain is the central problem in object-oriented analysis and design.
  - By abstraction, we ignore the inessential details.
  - An abstraction focuses on the outside view of an object.
    - Properties
    - Outside view of behavior
- What is “Abstraction” in context of Object-Oriented Programming (OOP)?
- Answer is: Class Data Type

Abstraction Example in C++

- Lets design a calculator in C++ that adds and subtracts numbers and displays the result:

Outside view Abstraction

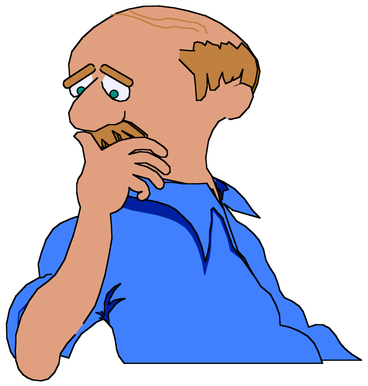
Outside view Abstraction in C++

```
class Calculator
{
    private:
       char* expression;
       char**   parsed_expression;
       double result;
        …
    public:

         Calculator();
         double add();
         double subtract();
         …
};
```

C++ Code for class Book

- Let’s develop a C++ class called “Book” for a Library Application.



One possible solution in C++

```
class Book{
private:
   char* title;
   char* publisher;
   char*  datePublished;
   char* bookState;

   char** authors;
  int numberOfAuthors;
public:
   Book(…);      // ctor to allocate memory
   ~Book();      // dtor to deallocate memory
   void bookInfo();
   // a set of getters and setter
};
```

A set of pointers to allocate memory for

A pointer to pointer to create a two dimensional array

Class Discussion

Is there a better solution for class Book

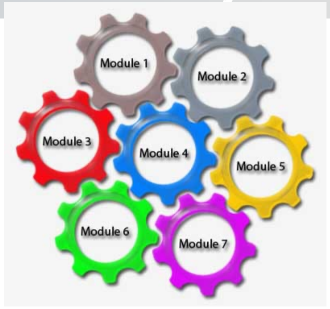
Encapsulation/Information Hiding

Information Hiding

- C++ achieves information hiding at two different levels:
  - Separation of outside view of an object from its inside view (secretes).
    - Although the interface of the methods are public, but the implementation detail of the objects and the current values of its data are hidden.
  - Second by keeping data hidden and invisible to other objects.
    - We can re-implement anything inside the object's capsule without affecting other objects that interact with it.

Modularity

Modularity



- Modularity is the property of a system decomposed into a set of cohesive and ***loosely*** coupled modules.
- A class/object is the lowest level of modularity in an object-oriented paradigm.
- At the higher-level modules are *physical* containers in which classes and objects (the logical design) are placed.
- A module has an *interface* and a *body* (implementation).
  - Changing the body requires recompiling just that module.
  - Changing the interface requires recompiling the module, plus all other modules that depend on the interface.

Example:

```
class Company {
    private:
        string name;
        string address;
        string dateStablished
    public:
        string getName() const;
        void setName(string name);
        …
        …
};
```

```
class Project {
    private:
        string title;
        string address;
        string dateStablished
    public:
        string getName() const;
        void setName(string name);
        …
        …
};
```

```
class Employee{
    private:
        string name;
        string address;
        string birthday
    public:
        …
};
```

- **Can we make it more modular?**
- **In other words, is there any data member in this definition that is a good candidate to be separated as another object**
- **The answer will be discussed during the lecture**

Another Example

- Reconsider class Book in one of the previous slides, discuss the possible options to improve its modularity.

```
class Book{
 private:
      string title;
      vector<string> authors;
      string publisher;
      string datePublished;
      string bookState;
  public:
      Book(…);
      void bookInfo();
      // assume a setters of getters
};
```

Please do your best to find out the right answer. The answer(s) will be discussed during the lecture.