Sunbeam Infotech

Exploring new ideas, Reaching new heights!

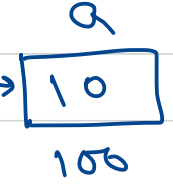
Reference is another name for variable (alias).
pointer is a var that store addr of another var.

```
int a = 10;
int &r = a;
cout << r;
```

must be initialized when declared.

automatically dereferenced

sizeof(r) = 4

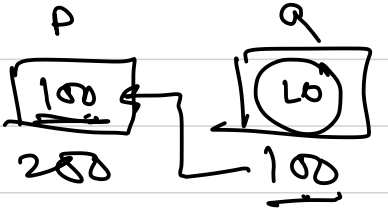



```
int a = 10;
int *p;
p = &a;
pf("%d", *p);
```

pointer is dangling

dereferencing op.

sizeof(p) = 8



```
void swap(int a, int b)
{
    int t = a;
    a = b;
    b = t;
}
```

a & b will be swapped

```
main()
{
    int x = 10, y = 20;
    swap(x, y);
    cout << x << y;
}
```

10 20

```
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

main()
{
    int x = 10, y = 20;
    swap(&x, &y);
    cout << x << y;
}
```

20 10

```
void swap(int &a, int &b)
{
    int t = a;
    a = b;
    b = t;
}

main()
{
    int x = 10, y = 20;
    swap(x, y);
    cout << x << y;
}
```

a & b will be swapped

x & y

20 10

```
int fun() {
    int a = 10;
    =
    return a;
}

main() {
    int x;
    x = fun();
    cout << x;
}
```

copy

```
int* fun() {
    static int a = 10;
    =
    return &a;
}

main() {
    int *x;
    x = fun();
    cout << *x;
}
```

Never return local vars by addr, bcoz they will be destroyed when fn return. & the pointer will be dangling.

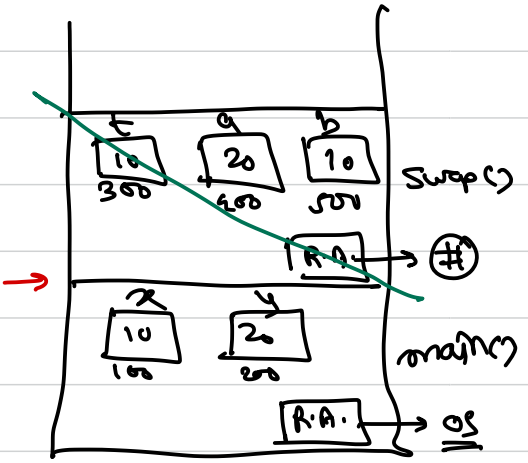
```
int& fun() {
    static int a = 10;
    =
    return a;
}

main() {
    int &x = fun();
    cout << x;
}
```

Never return local var by ref.

```
void swap(int a, int b)
{
    int t = a;
    a = b;
    b = t;
}
main()
{
    int x = 10, y = 20;
    swap(x, y);
    cout << x << y;
}
```

10 20



working of function call.

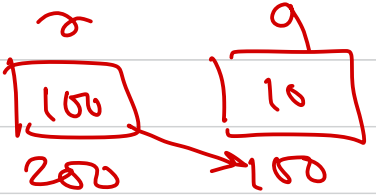
FAR = local vars + Args + Return Addr.
SF

Internally, ref is a pointer that is automatically dereferenced (no * operator).

size of pointer = 8 bytes.
size of refence = 8 bytes.

```
char ch;
class A {
{       charR a;
public:
        A() : a(ch) {
        }
};
```

```
int a=10;
int &r=a;
```




```
main() {
    A obj;
    Cout<< sizeof(obj);
}
```

```
cout << r;
```

8

Inline fns

typesafe

```
inline int Sqr(int n) {
    return n*n;
}
```

→ not preprocessor

request Compiler to replace fn defn at its call.

- accepted for small & simple fns.
- ignored if fn have loops, fn is big, fn is recursive

```
main() {
    cout << Sqr(5);
```

inline member fns must be defined in .h file of the class (not in .cpp).

not type-safe

macros

```
#define SQR(n) n*n
main() {
    cout << SQR(5);      // 5*5
    cout << SQR(2+3);    // 2+3*2+3
}
```

C++ DMA

In C → malloc(), free()

* malloc() is a library fn - <stdlib.h>.
* person *p = (person*) malloc (sizeof (person));
  - ① alloc mem
* return NULL on failure.
* release mem using free()

In C++ → new, delete

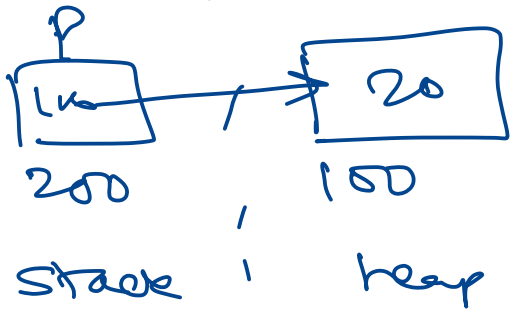
* new is a operator.

class person

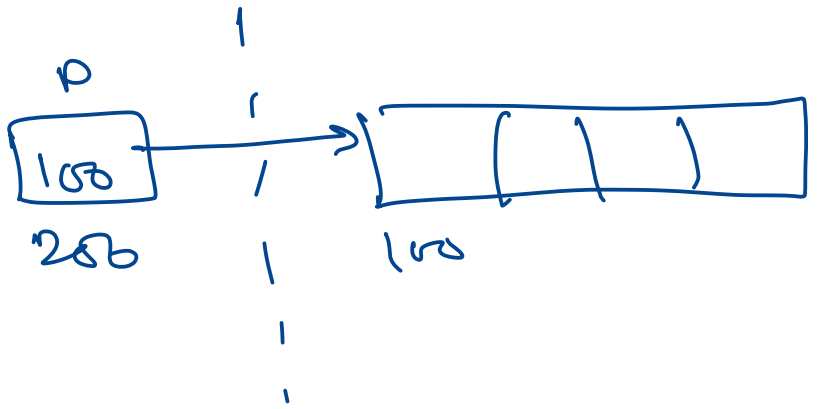
* person *p = new person;
  - ① alloc mem
  - ② Call Constructor
* throw exception bad_alloc. on failure.
* release mem using delete..

dynamic mem alloc. on heap.

```
main() {
    int * p = new int;
    *p = 20;
    cout << *p;
    delete p;
}
```




```
main () {
    int *p = new int[4];
    for(i=0; i<4; i++)
        cin >> p[i];
    for(i=0; i<4; i++)
        cout << p[i];
    delete[] p;
}
```

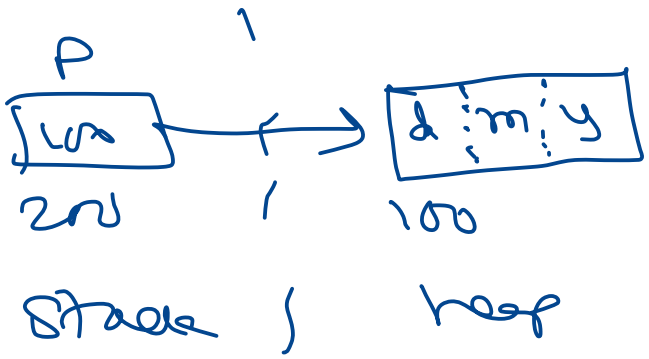


class date { ... }; → 12 bytes

```
main()
{
    date *p = new date;
    p->accept();
    p->display();
    delete p;
}
```

↳ ① alloc 12 bytes
② call paramless ctor.

① Calls destructor.
② dealloc 12 bytes.



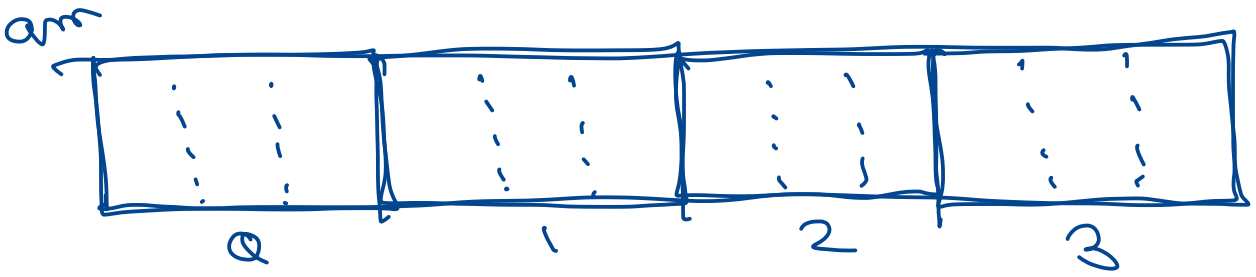

```
main()
{
    date *p = new date(1, 1, 2000);
    p->accept();
    p->display();
    delete p;
}
```

↳ ① alloc 12 bytes
② call param ctor.

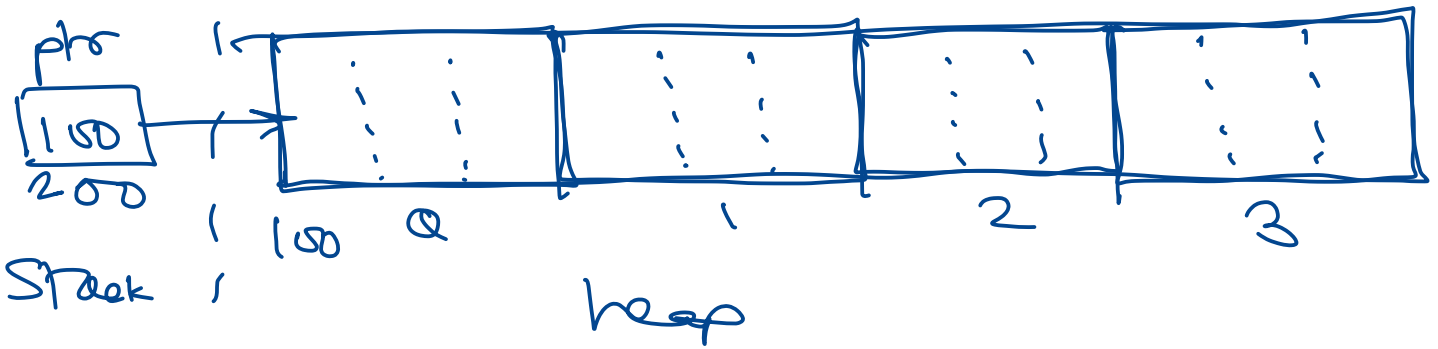
① Calls destructor.
② dealloc 12 bytes.

param less ctor called for each object.

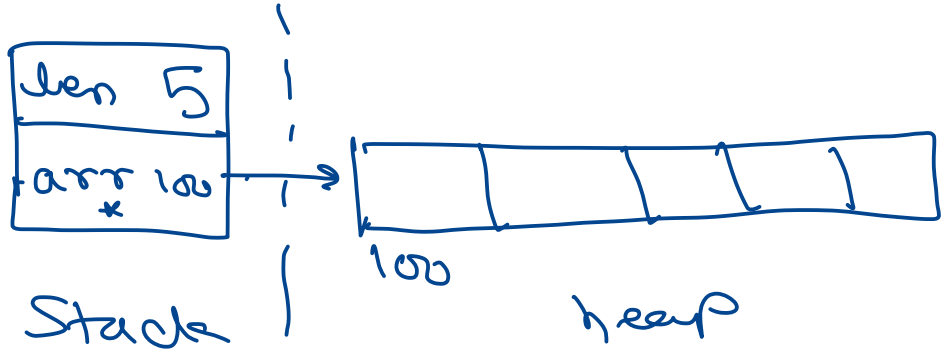
```
main() {
    Date arr[4];
    for(i=0; i<4; i++)
        arr[i].display();
    return 0;
}
```




```
main () {
    Date *ptr = new Date[4];
    for(i=0; i<5; i++)
        ptr[i].display();
    delete [] ptr;
}
```



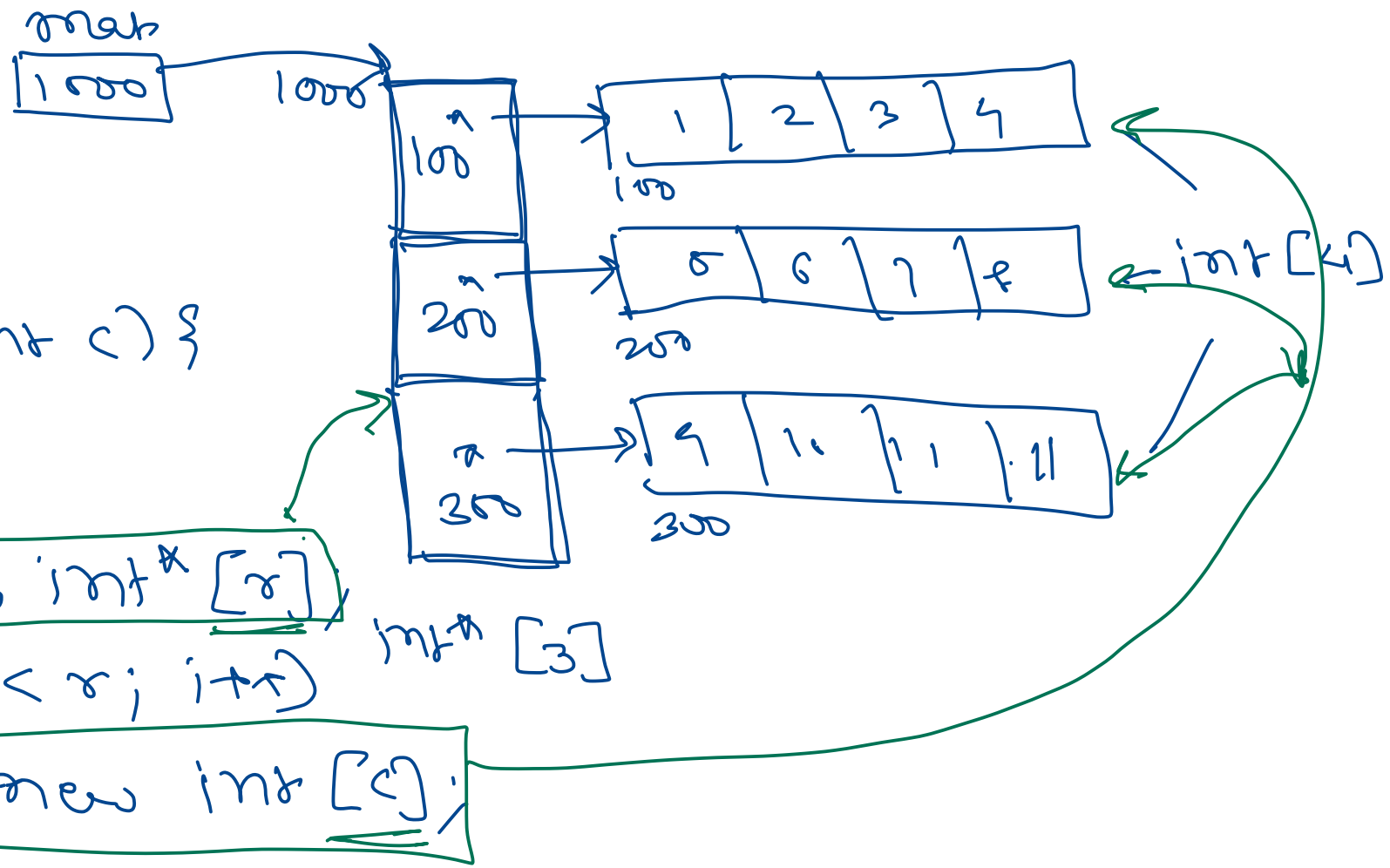
```
class array {
    int len;
    int *arr;
public:
    array(int len) {
        this->len = len;
        this->arr = new int[len];
    }
    ~array() {
        delete[] this->arr;
    }
    // accept() | display()
};
```




```
main() {
    array obj(5);
    obj.accept();
    obj.display();
}
```

3 x 4

```
class matrix {
    int *arr;
    int rows;
    int cols;
matrix(int r, int c) {
    rows = r;
    cols = c;
    mat = new int*[r];
    for(i=0; i<r; i++)
        mat[i] = new int[c];
};  }
```



```
~matrix() {
    for(i=0; i<r; i++)
        delete[] mat[i];
    delete[] mat;
}
void display() {
    for(i=0; i<rows; i++) {
        for(j=0; j<cols; j++)
            cout << mat[i][j];
        cout << endl;
    }
}
```

```
try {
    cin >> num >> den;
    if (den == 0)
        throw den;
    res = num / den;
    cout << res;
}
catch (int den) {
    cout << "div by 0";
}
```

```
class account {
    int id;
    ---------
    double bal;
public:
    //ctor
    //dtor
    //getter/setter
    void withdraw(double amt) {
        if( bal - amt <= 0) {
            ex_cls ex_obj(id, amt, -, -);
            throw ex_obj;
        }
        bal = bal - amt;
    }
};
```

```
class ex_cls {
    int id;
    double amt;
    ==
public:
    ex_cls(int id, double amt, -, -);
    ==
    this->id = id;
    this->amt = amt;
    ==
};
```

```
main() {
    account a1;
    ==
    try {
        a1.withdraw(5000);
    } catch (--- &ex) {
        //email..
    }
}
```

OOP Concepts → Grady Booch / UML - Rambaug

- Major pillars
  - Abstraction
  - Encapsulation
  - Hierarchy
  - Modularity
- Minor pillars
  - Typing
  - Concurrency
  - Persistence

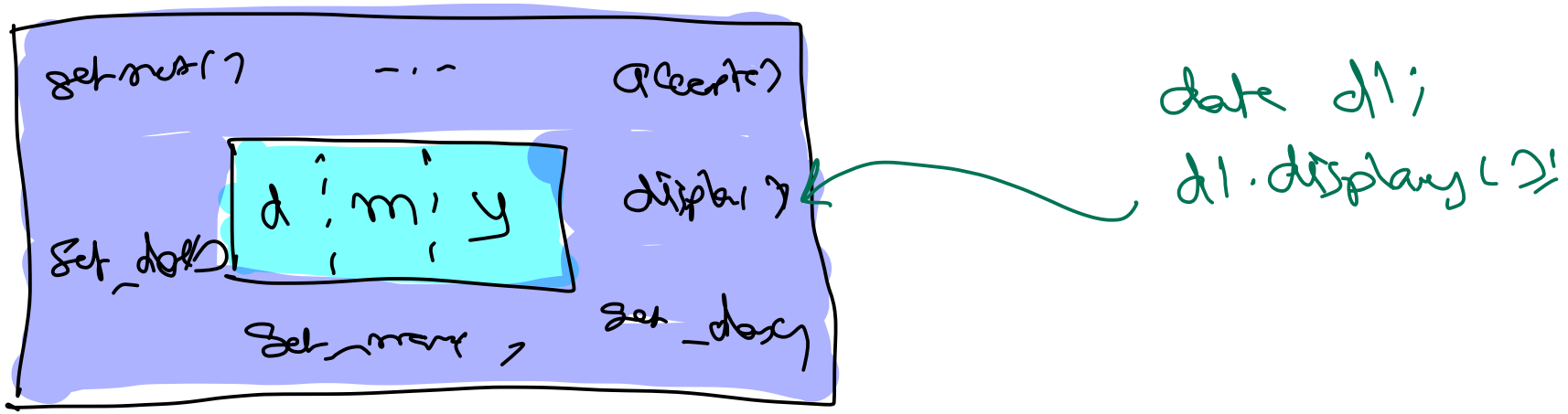
- If language supports only abstraction & encapsulation, it is said to be “Object based language”. VB, JS
- Programming language supporting all major pillars, is called as “OOP language”. → C++
- Programming language supporting all major & minor pillars, is called as “Pure OOP language”. → C#
  - Every data must be object (including primitive types). x C++, Java
  - No global variables/functions allowed.
  - No violation OO concepts. x C++ → macro

x C++ : friend, ...

Abstraction & Encapsulation

- Abstraction is getting essential details of the system.
- Deals with the interface provided by the system.
- Abstraction will change as per perspective of user.

- Encapsulation is binding of internal mechanism with external interface.
- Deals with internal complexity of the system.
- Binding data members & member functions together in a class.



Data hiding & security

	in class	in derived	out class
private:	✓	✗	✗
protected:	✓	✓	✗
public:	✓	✓	✓

- OOP languages provides access specifiers like private, public & protected. They control visibility of the member.
- Data not to be accessed directly from outside the class, should be done private.
- These members can be accessed from outside using inspectors (if implemented).
- These members can be modified from outside using mutators (if implemented, with appropriate value checks).

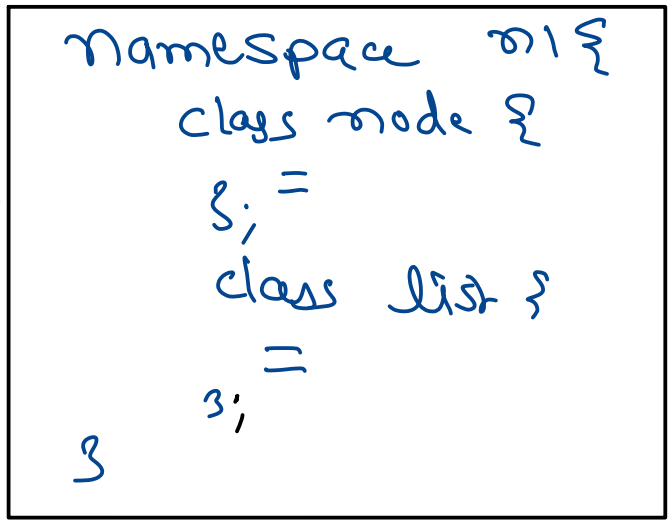
```
class date {
private:
    int day,mon,yr;
public:
    int get_mon(){
        return mon;
    }
    void set_mon(int m){
        if(m<1 || m >12)
            throw m;
        mon = m;
    }
};
```

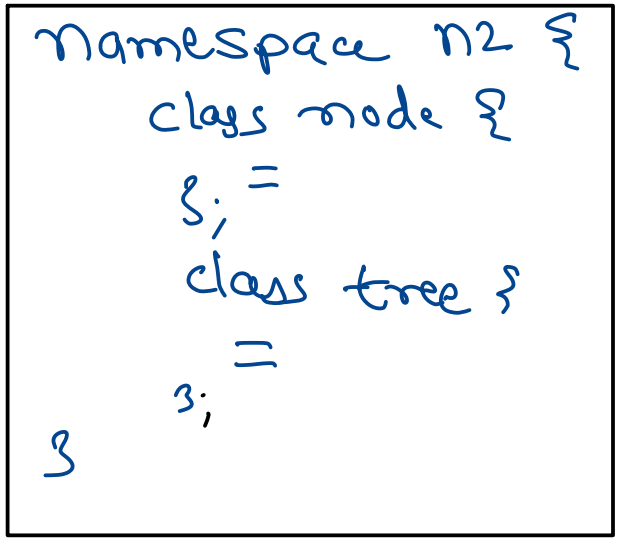
getters

setters

Modularity

- To make the development & maintenance simplified for complex systems, solution must be modularized.
- Modularization can be logical or physical.
- Possible modularation methods:
  - Method
  - Class
  - Namespace/Package
  - Libraries
    - .dll
    - .so
  - files





Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>