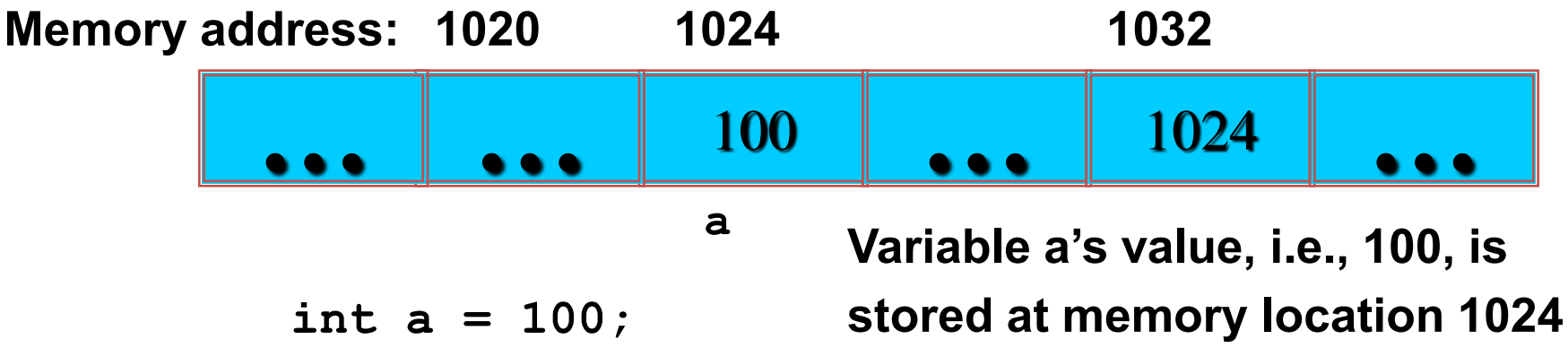
Pointers
and
dynamic objects

Topics

- Pointers
  - Memory addresses
  - Declaration
  - Dereferencing a pointer
  - Pointers to pointer
- Static vs. dynamic objects
  - `new` and `delete`

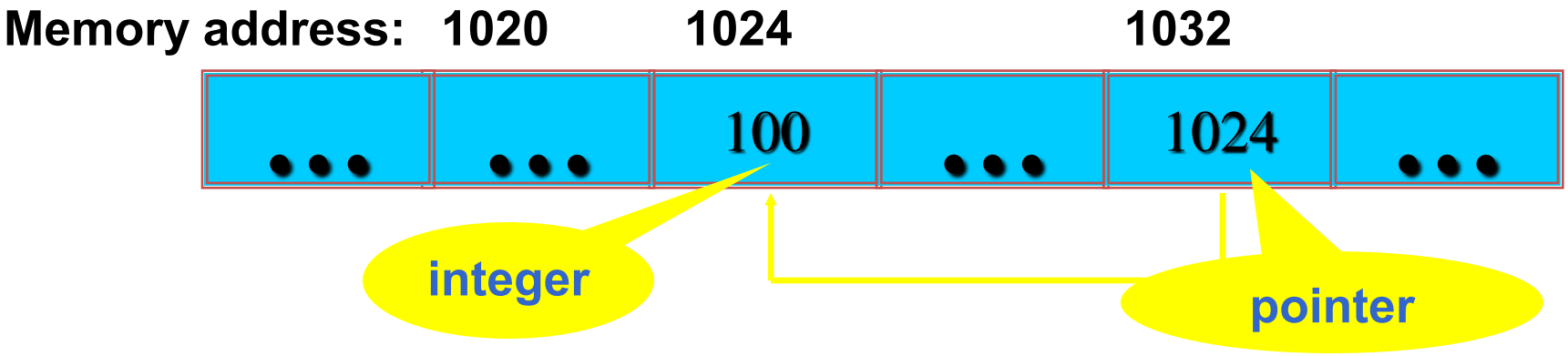
Computer Memory

- Each variable is assigned a memory slot (the size depends on the data type) and the variable's data is stored there



Pointers

- A pointer is a variable used to store the address of a memory cell.
- We can use the pointer to reference this memory cell



Pointer Types

- Pointer
  - C++ has pointer types for each type of object
    - Pointers to `int` objects
    - Pointers to `char` objects
    - Pointers to user-defined objects

      (e.g., `RationalNumber`)
  - Even pointers to pointers
    - Pointers to pointers to `int` objects

Pointer Variable

- Declaration of Pointer variables

```
type* pointer_name;
//or
type *pointer_name;
```

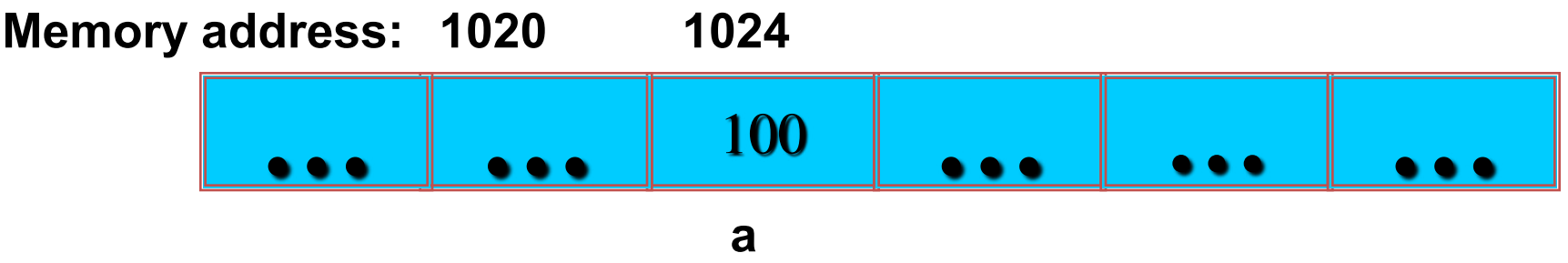
where *type* is the type of data pointed to (e.g. int, char, double)

Examples:

```
int *n;
RationalNumber *r;
int **p;    // pointer to pointer
```

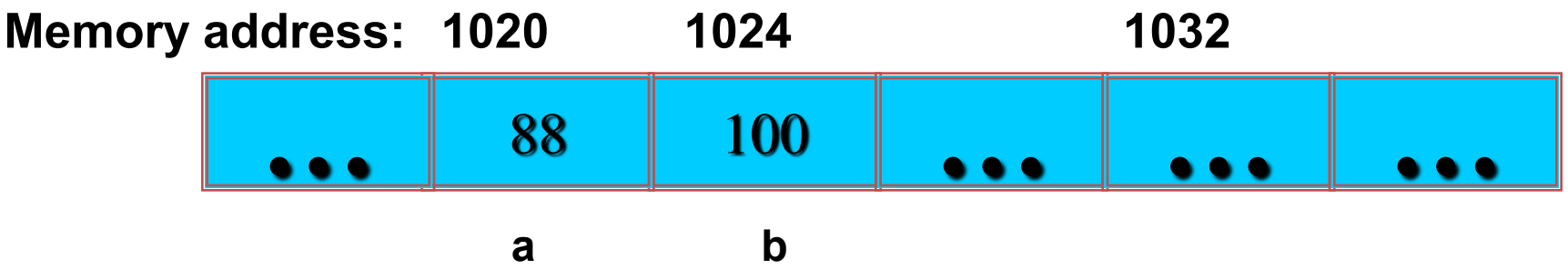
Address Operator &

- *The "address of" operator* (&) gives the memory address of the variable
  - Usage: **&variable_name**




```
int a = 100;
//get the value,
cout << a;     //prints 100
//get the memory address
cout << &a;    //prints 1024
```

Address Operator &



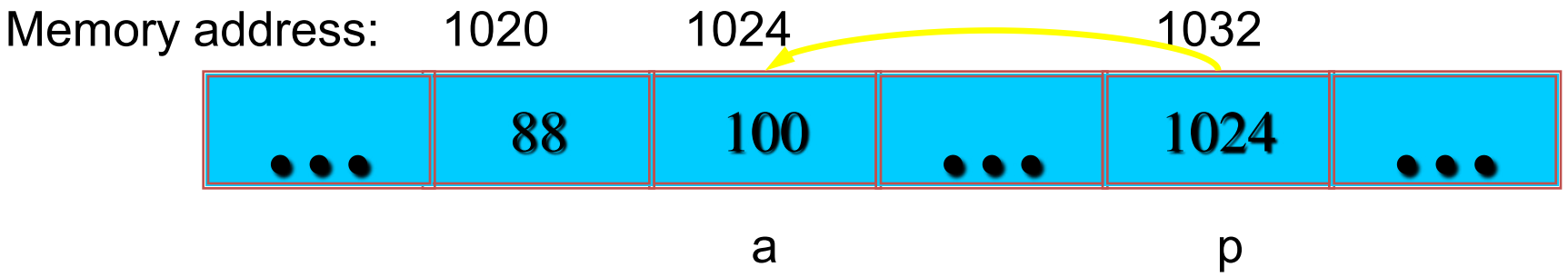

```
#include <iostream>
using namespace std;
void main(){
   int a, b;
   a = 88;
   b = 100;
   cout << "The address of a is: " << &a << endl;
   cout << "The address of b is: " << &b << endl;
}
```

- Result is:

The address of a is: 1020

The address of b is: 1024

Pointer Variables




```
int a = 100;
int *p = &a;
cout << a << " " << &a <<endl;
cout << p << " " << &p <<endl;
```

- Result is:

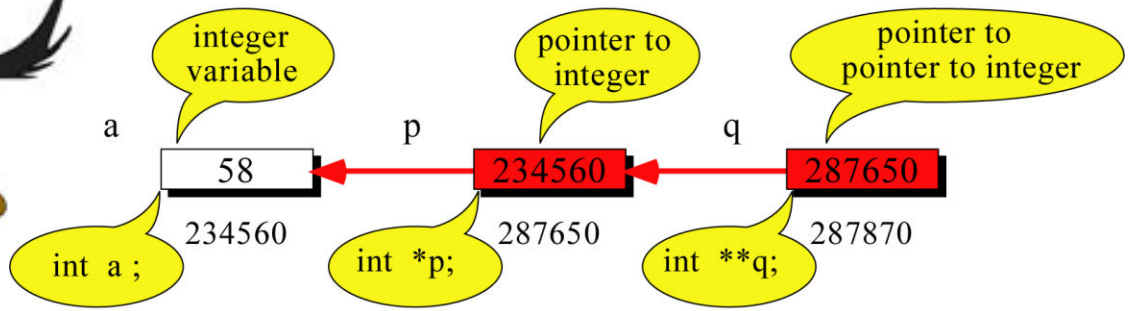
100 1024

1024 1032

- The value of pointer `p` is the address of variable `a`
- A pointer is also a variable, so it has its own memory address

Pointer to Pointer

```
// Local Declarations
int         a ;
int        *p ;
int       **q ;
```



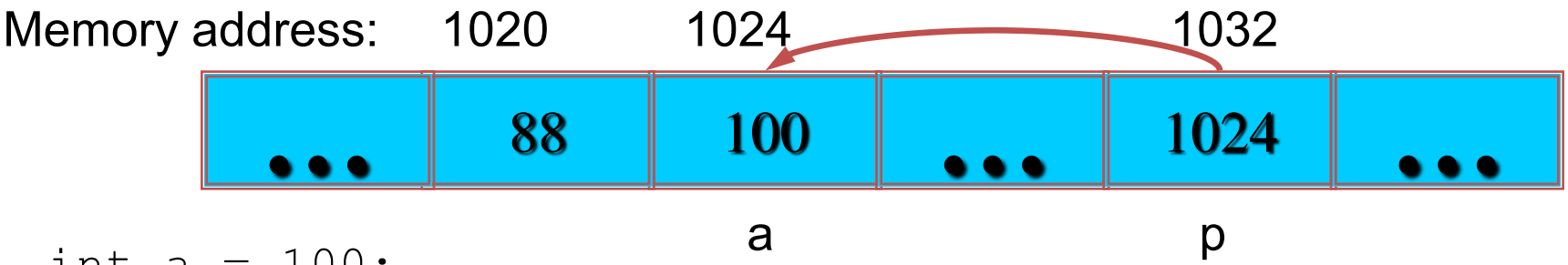
What is the output?

58 58 58

```
// Statements
a = 58 ;
p = &a ;
q = &p ;
cout <<        a << " ";
cout <<       *p << " ";
cout <<      **q << " ";
```

Dereferencing Operator *

- We can access to the value stored in the variable pointed to by using the dereferencing operator (*),




```
int a = 100;
int *p = &a;
cout << a << endl;
cout << &a << endl;
cout << p << " " << *p << endl;
cout << &p << endl;
```

- Result is:

100
1024
1024 100
1032

Don’t get confused

- Declaring a pointer means only that it is a pointer: `int *p;`
- Don’t be confused with the dereferencing operator, which is also written with an asterisk (`*`). They are simply two different tasks represented with the same sign

```
int a = 100, b = 88, c = 8;
int *p1 = &a, *p2, *p3 = &c;
p2 = &b;     // p2 points to b
p2 = p1;     // p2 points to a
b = *p3;     //assign c to b
*p2 = *p3;   //assign c to a
cout << a << b << c;
```

- **Result is: 888**

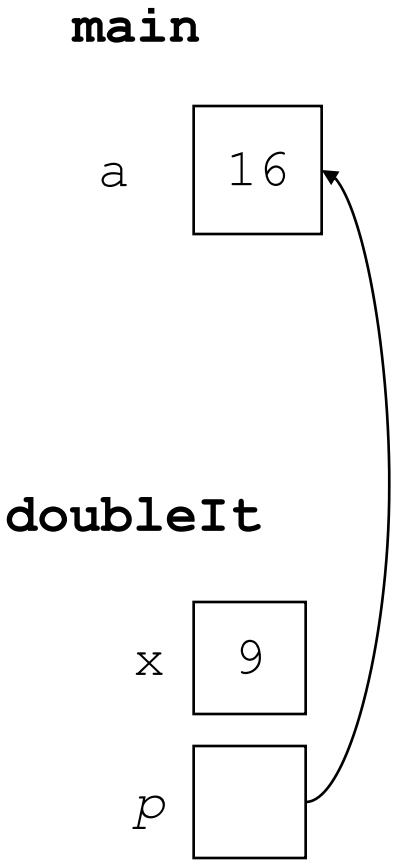
A Pointer Example

The code

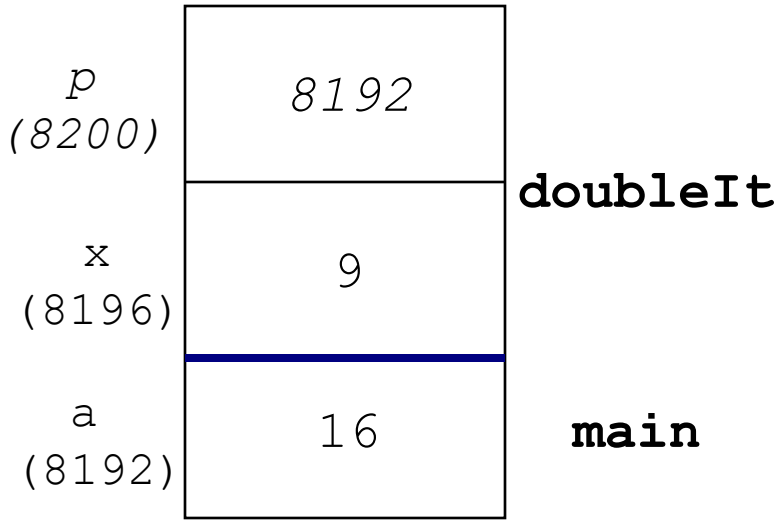
```
void doubleIt(int x,
              int * p)
{
   *p = 2 * x;
}
int main(int argc, const
   char * argv[])
{
   int a = 16;
   doubleIt(9, &a);
   return 0;
}
```

a gets 18

Box diagram



Memory Layout



Another Pointer Example

- **Let's figure out:**

value1==? / value2==?

Also, p1=? p2=?

```
#include <iostream>
using namespace std;
int main (){
   int value1 = 5, value2 = 15;
   int *p1, *p2;
   p1 = &value1; // p1 = address of value1
   p2 = &value2; // p2 = address of value2
   *p1 = 10;     // value pointed to by p1=10
   *p2 = *p1;    // value pointed to by p2= value
                 // pointed to by p1
   p1 = p2;      // p1 = p2 (pointer value copied)
   *p1 = 20;     // value pointed to by p1 = 20
   cout << "value1==" << value1 << "/ value2==" <<
   value2;
   return 0;
}
```

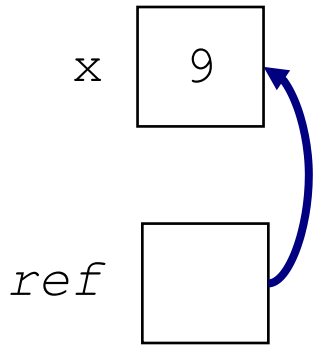
Another Pointer Example

```
    int a = 3;
    char s  = 'z';
    double d = 1.03;
    int *pa = &a;
    char *ps = &s;
    double *pd = &d;
% sizeof returns the # of bytes…
    cout << sizeof(pa) << sizeof(*pa)
         << sizeof(&pa) << endl;
    cout << sizeof(ps) << sizeof(*ps)
         << sizeof(&ps) << endl;
    cout << sizeof(pd) << sizeof(*pd)
         << sizeof(&pd) << endl;
```

Reference Variables

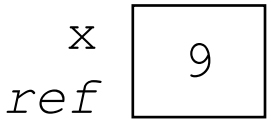
A reference is an additional name to an existing memory location

Pointer:




```
int x=9;
int *ref;
ref = &x;
```

Reference:




```
int x = 9;
int &ref = x;
```

Reference Variables

- A reference variable serves as an alternative name for an object

```
int m = 10;
int &j = m;  // j is a reference variable
cout << "value of m = " << m << endl;
                    //print 10
j = 18;
cout << "value of m = " << m << endl;
     // print 18
```

Reference Variables

- A reference variable always refers to the same object. Assigning a reference variable with a new value actually changes the value of the referred object.
- Reference variables are commonly used for parameter passing to a function

Traditional Pointer Usage

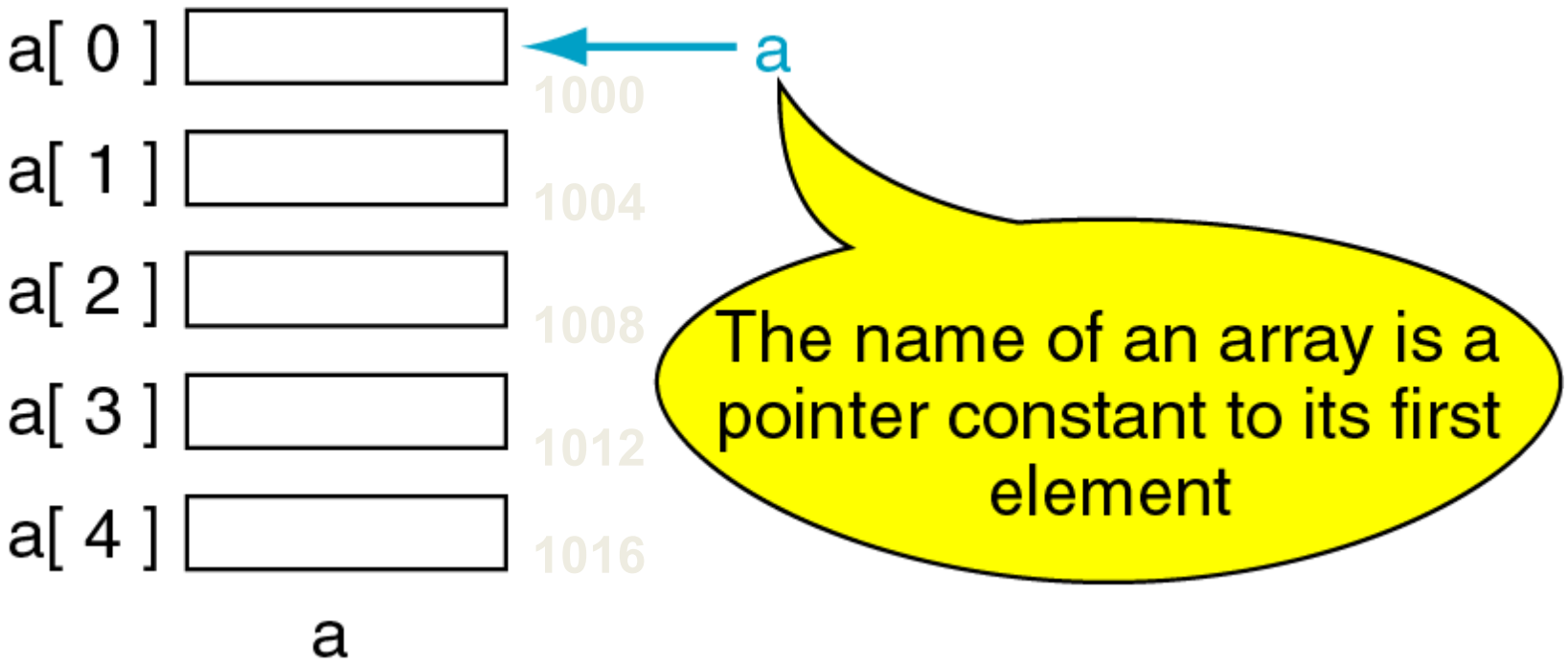
```
void IndirectSwap(char *Ptr1, char *Ptr2){
    char temp = *Ptr1;
    *Ptr1 = *Ptr2;
    *Ptr2 = temp;
}
int main() {
    char a = 'y';
    char b = 'n';
    IndirectSwap(&a, &b);
    cout << a << b << endl;
    return 0;
}
```

Pass by Reference

```
void IndirectSwap(char& y, char& z) {
   char temp = y;
   y = z;
   z = temp;
}
int main() {
   char a = 'y';
   char b = 'n';
   IndirectSwap(a, b);
   cout << a << b << endl;
   return 0;
}
```

Pointers and Arrays

- The name of an array points only to the first element not the whole array.



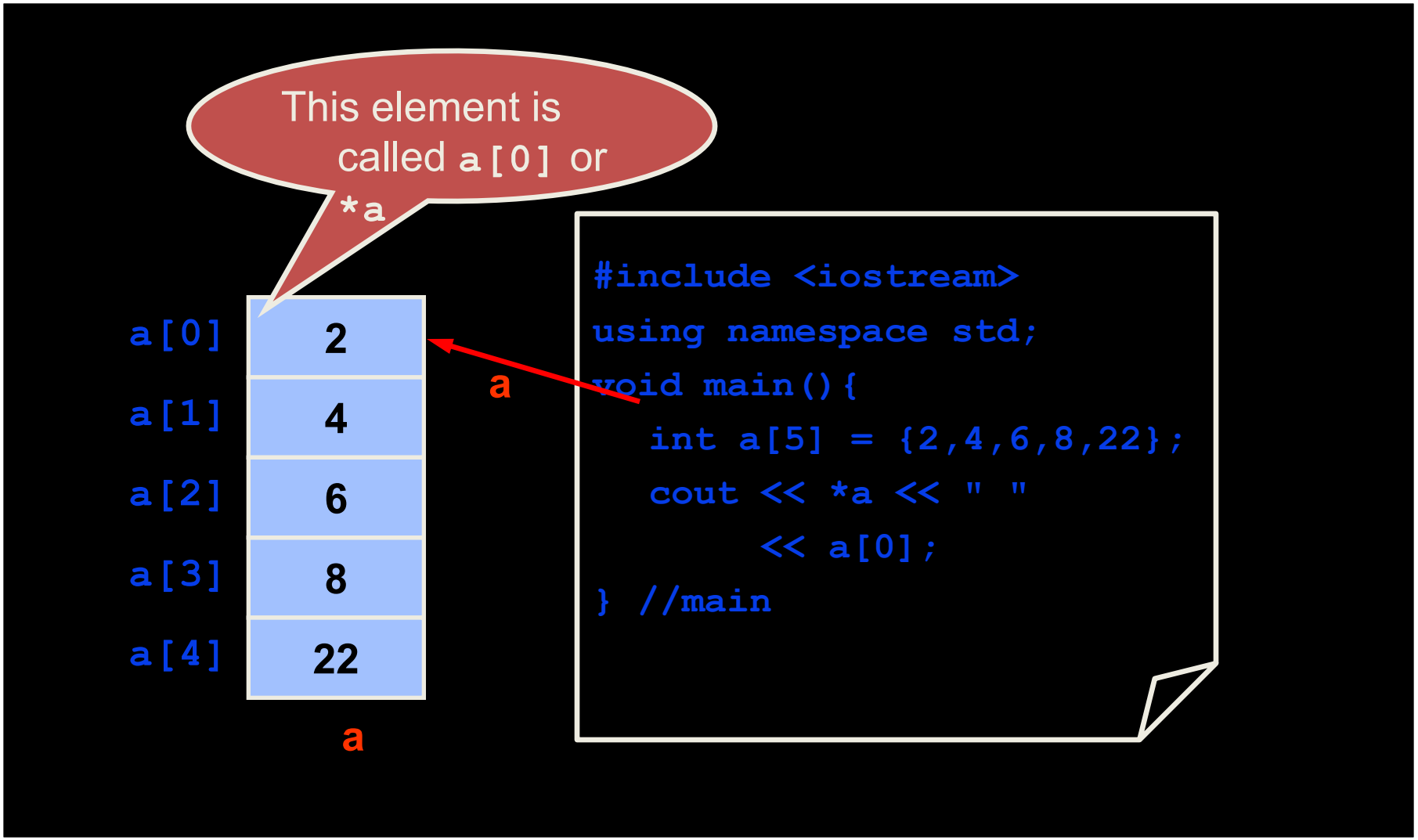
Array Name is a pointer constant

```
#include <iostream>
using namespace std;

void main (){
    int a[5];
    cout << "Address of a[0]: " << &a[0] << endl
         << "Name as pointer: " << a << endl;
}
```

Result:

```
Address of a[0]: 0x0065FDE4
Name as pointer: 0x0065FDE4
```

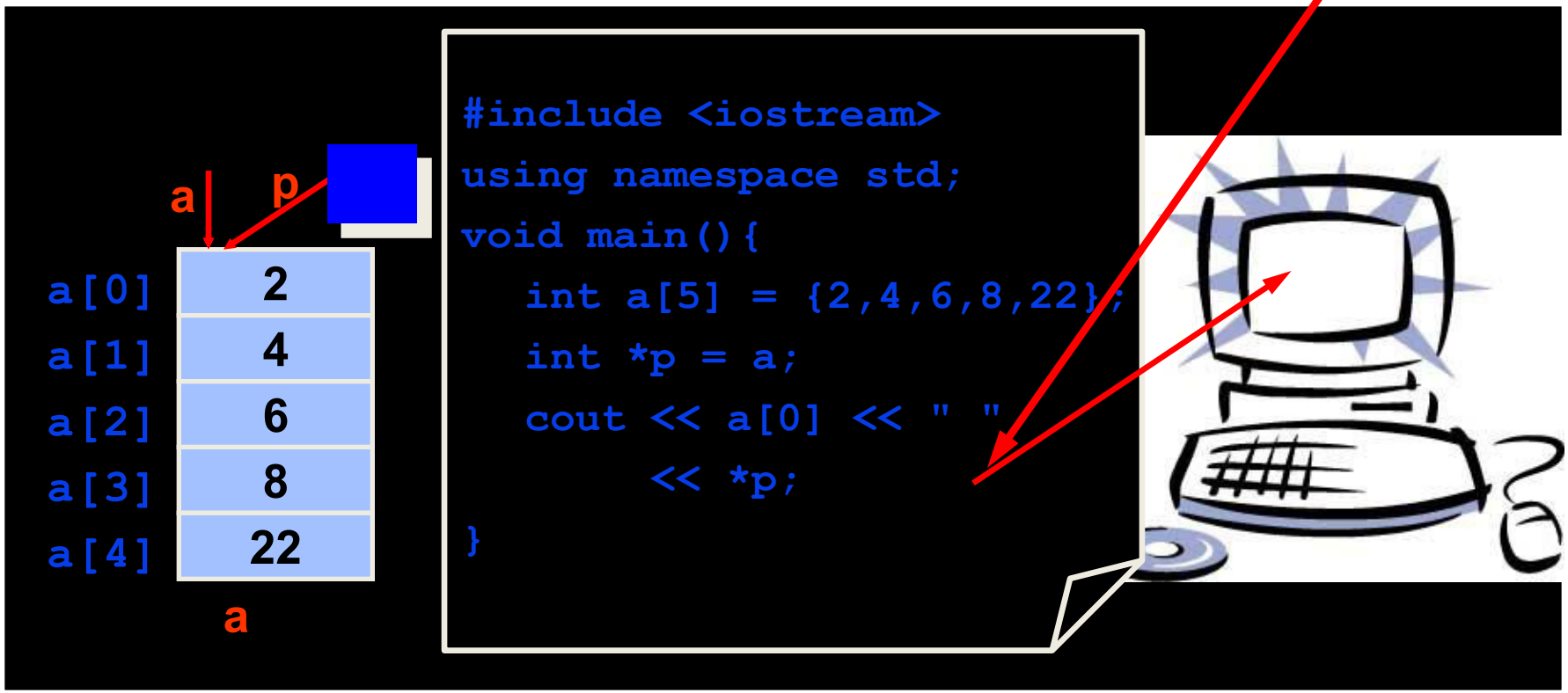
Dereferencing An Array Name



Array Names as Pointers

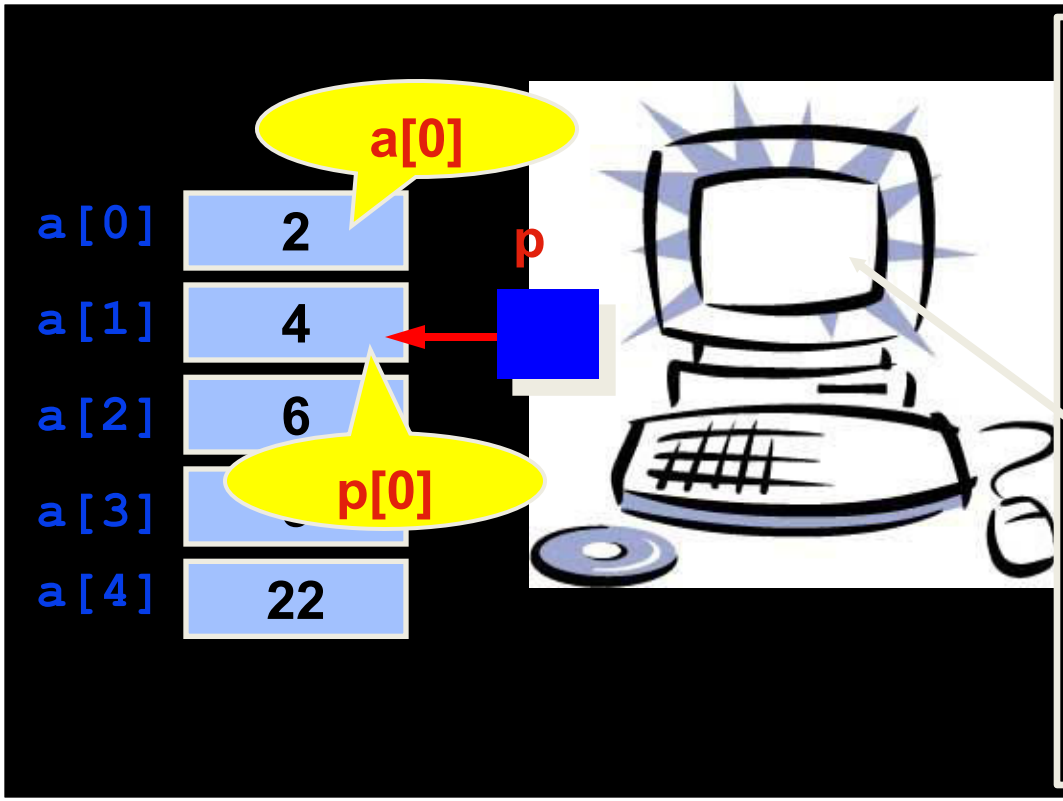
- To access an array, any pointer to the first element can be used instead of the name of the array.

We could replace `*p` by `*a`



Multiple Array Pointers

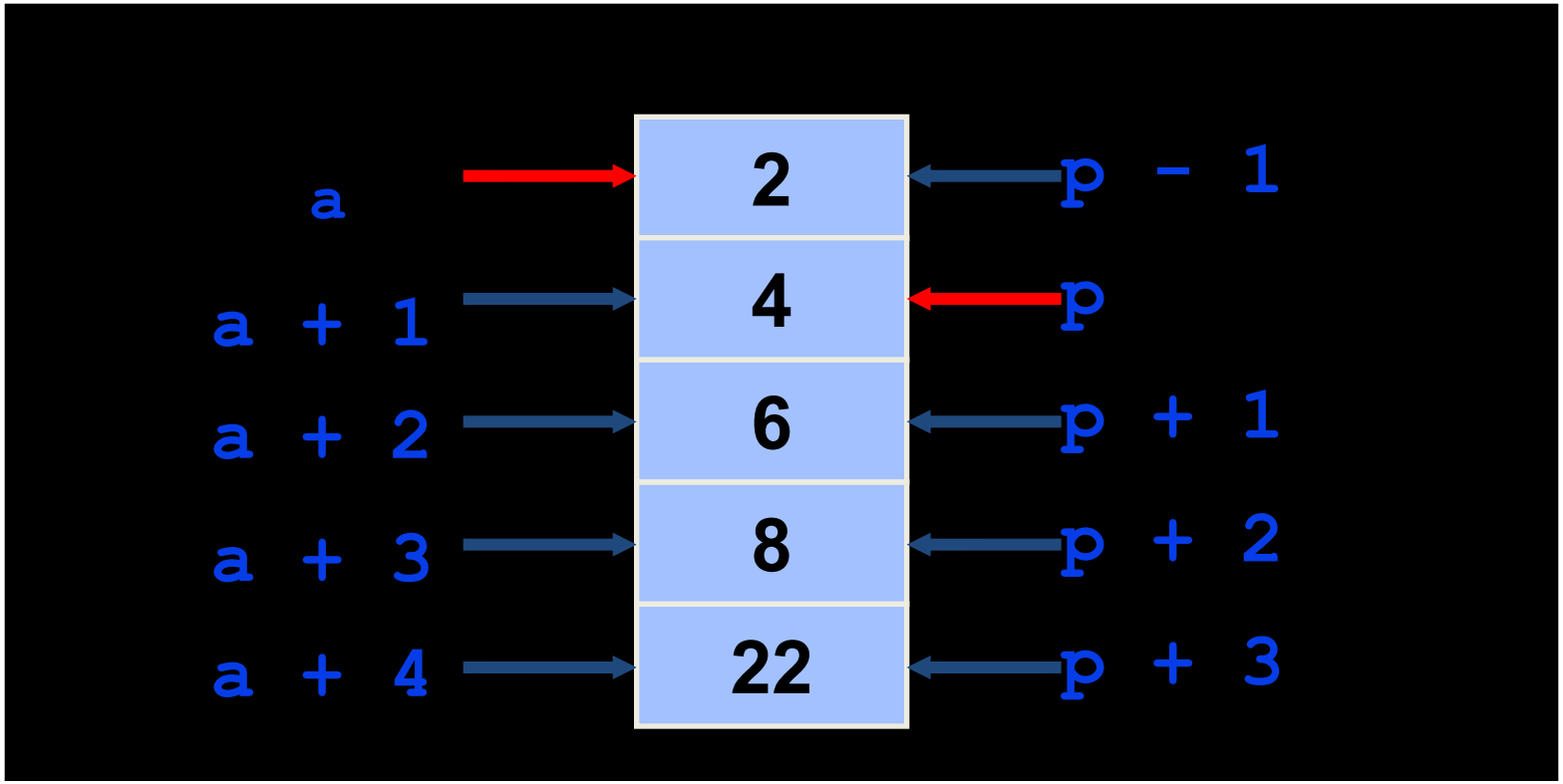
- Both `a` and `p` are pointers to the same array.



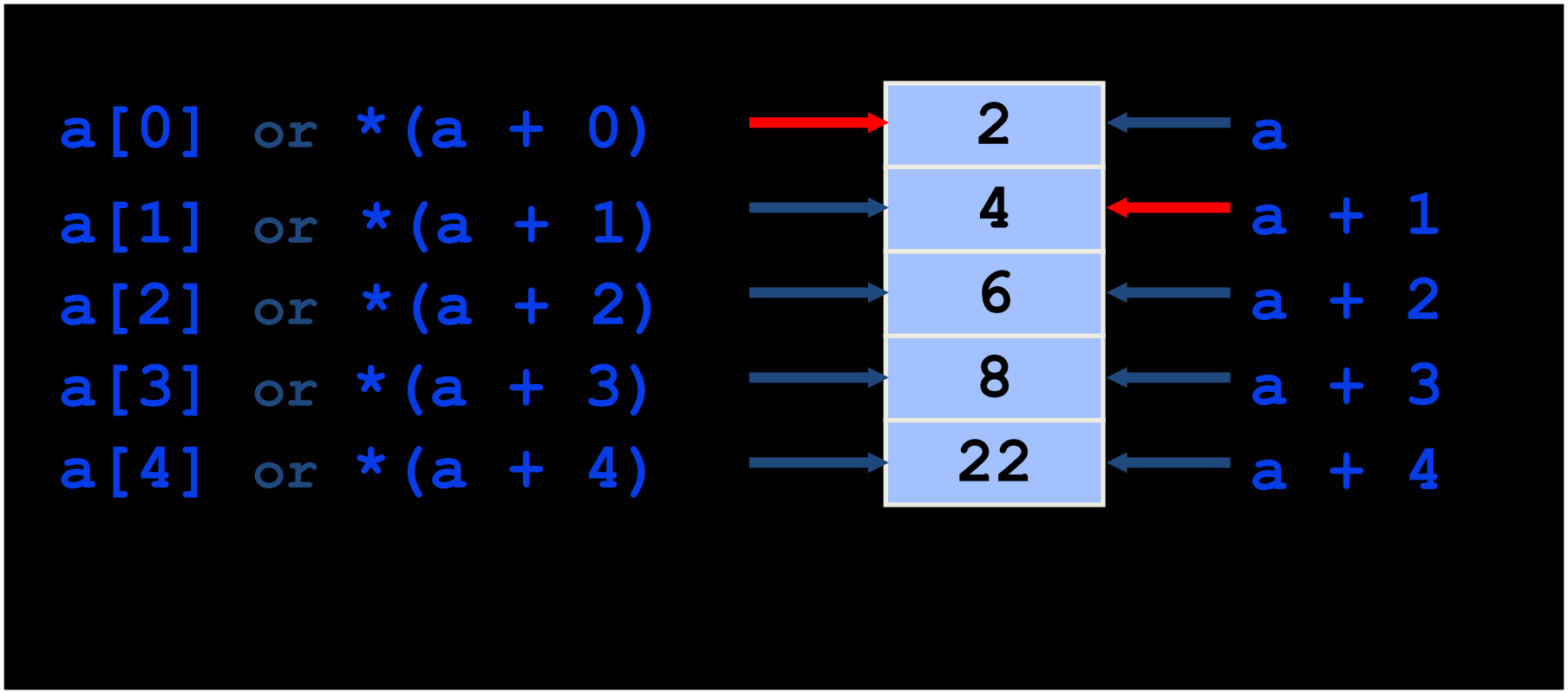

```
#include <iostream>
using namespace std;
void main(){
   int a[5] = {2,4,6,8,22};
   int *p = &a[1];
   cout << a[0] << " "
        << p[-1];
   cout << a[1] << " "
        << p[0];
}
```

Pointer Arithmetic

- Given a pointer `p`, `p+n` refers to the element that is offset from `p` by `n` positions.



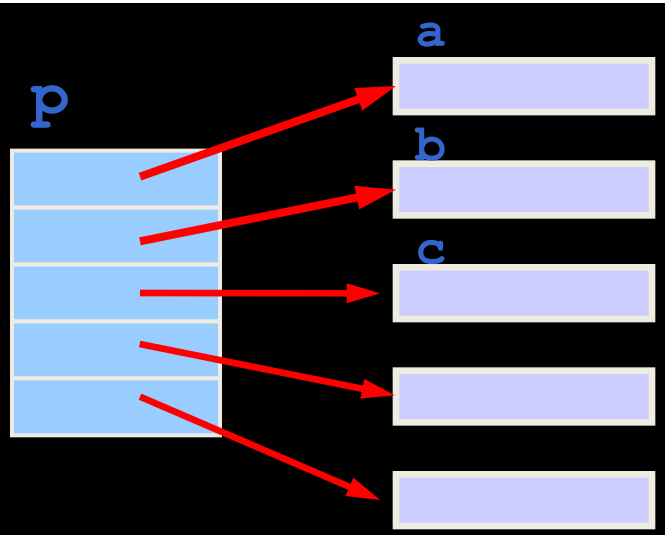
Dereferencing Array Pointers



`*(a+n)` is identical to `a[n]`

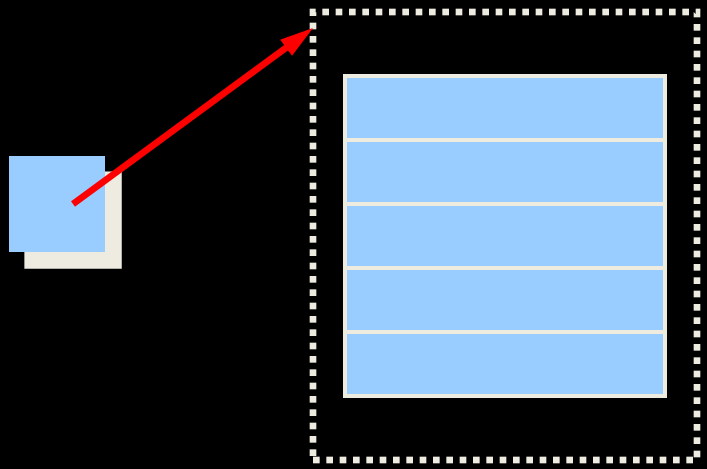
- Note: flexible pointer syntax

Array of Pointers & Pointers to Array



An array of Pointers

```
int a = 1, b = 2, c = 3;
int *p[5];
p[0] = &a;
p[1] = &b;
p[2] = &c;
```



A pointer to an array

```
int list[5] = {9, 8, 7, 6, 5};
int *p;
P = list;//points to 1st entry
P = &list[0];//points to 1st entry
P = &list[1];//points to 2nd entry
P = list + 1; //points to 2nd entry
```

NULL pointer

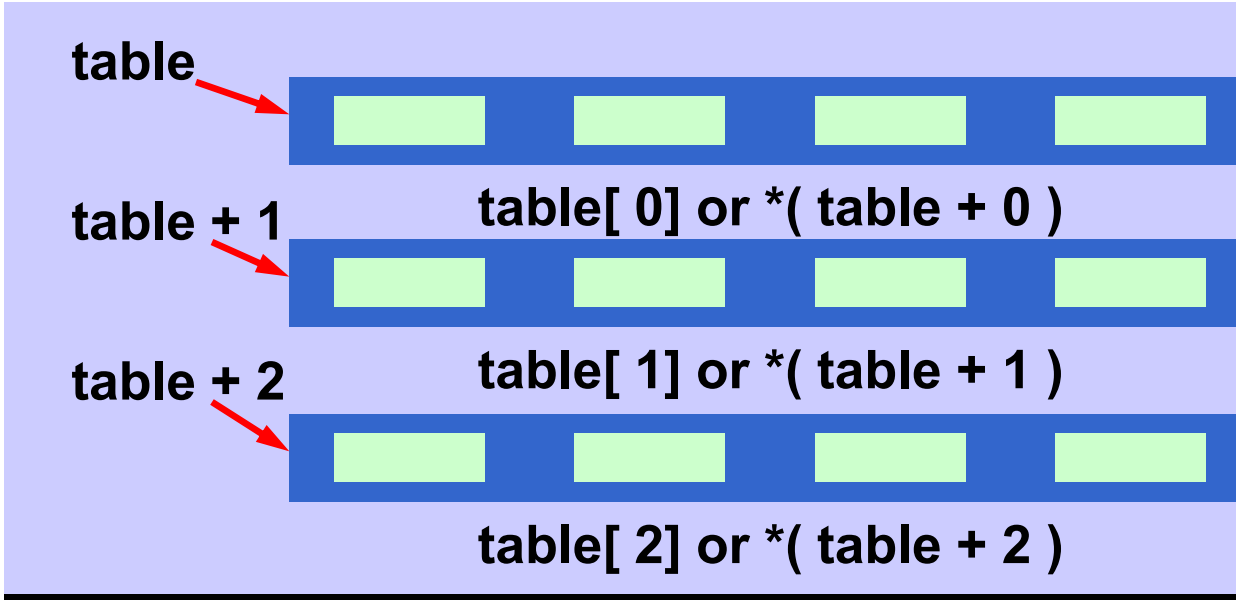
- NULL is a special value that indicates an empty pointer
- If you try to access a NULL pointer, you will get an error

```
int *p;
p = 0;
cout << p << endl; //prints 0
cout << &p << endl;//prints address of p
cout << *p << endl;//Error!
```

Storing 2D Array in 1D Array

```
int twod[3][4] = {{0,1,2,3}, {4,5,6,7},
                  {8,9,10,11}};

int oned[12];
for(int i=0; i<3; i++){
    for(int j=0; j<4 ; j++)
          oned[i*4+j] = twod[i][j];
}
```

Pointer to 2-Dimensional Arrays



`table[i] = &table[i][0]` refers to the *address* of the ith row

What is `**table` ?

```
int table[3][4] = {{1,2,3,4},
{5,6,7,8},{9,10,11,12}};

for(int i=0; i<3; i++){
   for(int j=0; j<4; j++)
        cout << *(*(table+i)+j);
   cout << endl;
}
```

`*(table[i]+j) = table[i][j]`

Dynamic Objects

Memory Management

- Static Memory Allocation
  - Memory is allocated at compilation time
- Dynamic Memory
  - Memory is allocated at running time

Static vs. Dynamic Objects

- Static object

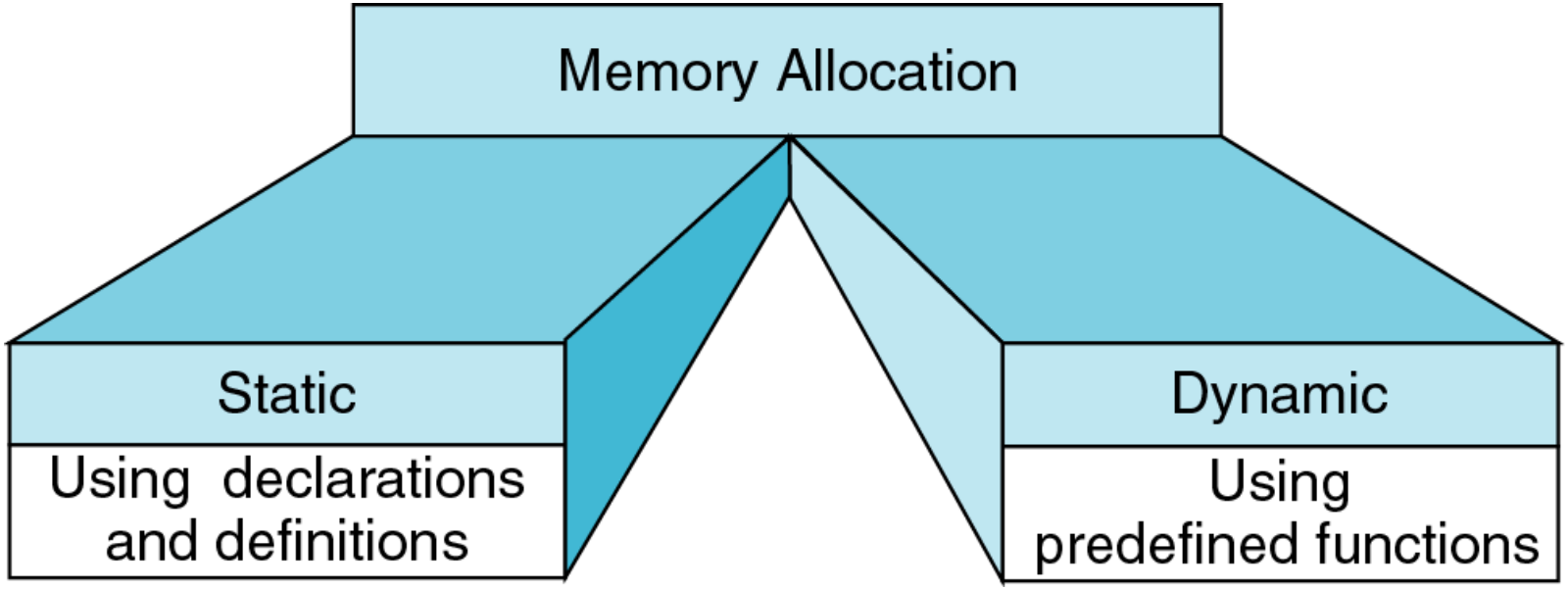
  (variables as declared in function calls)

  - Memory is acquired automatically
  - Memory is returned automatically when object goes out of scope

- Dynamic object

  - Memory is acquired by program with an allocation request
    - `new` operation
  - Dynamic objects can exist beyond the function in which they were allocated
  - Object memory is returned by a deallocation request
    - `delete` operation

Memory Allocation



new
delete

```
{
   int a[200];
   …
}
```

```
int* ptr;
ptr = new int[200];
…
delete [] ptr;
```

Object (variable) creation: `New`

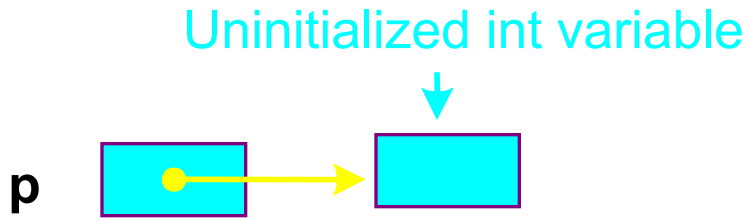
Syntax

```
ptr = new SomeType;
```

where `ptr` is a pointer of type `SomeType`

Example

```
int* p = new int;
```



Object (variable) destruction: Delete

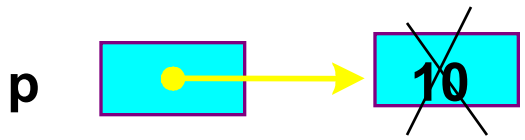
Syntax

```
delete p;
```

storage pointed to by p is returned to free store and p is now undefined

Example

```
int* p = new int;
*p = 10;
delete p;
```



Array of `New`: **dynamic arrays**

- Syntax

```
P = new SomeType[Expression];
```

  - Where
    - `P` is a pointer of type `SomeType`
    - `Expression` is the number of objects to be constructed -- we are making an array

- Because of the flexible pointer syntax, `P` can be considered to be an array

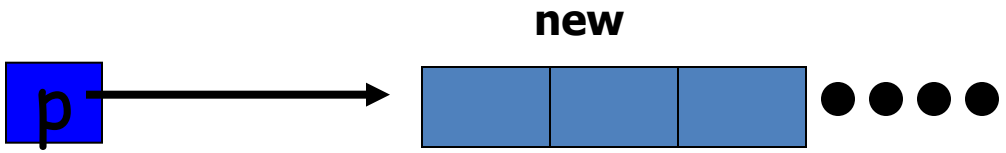
Example

Dynamic Memory Allocation

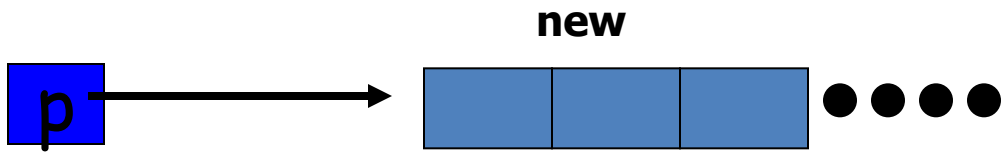
- Request for "unnamed" memory from the Operating System

- `int *p, n=10;`
  `p = new int;`

`p = new int[100];`



`p = new int[n];`

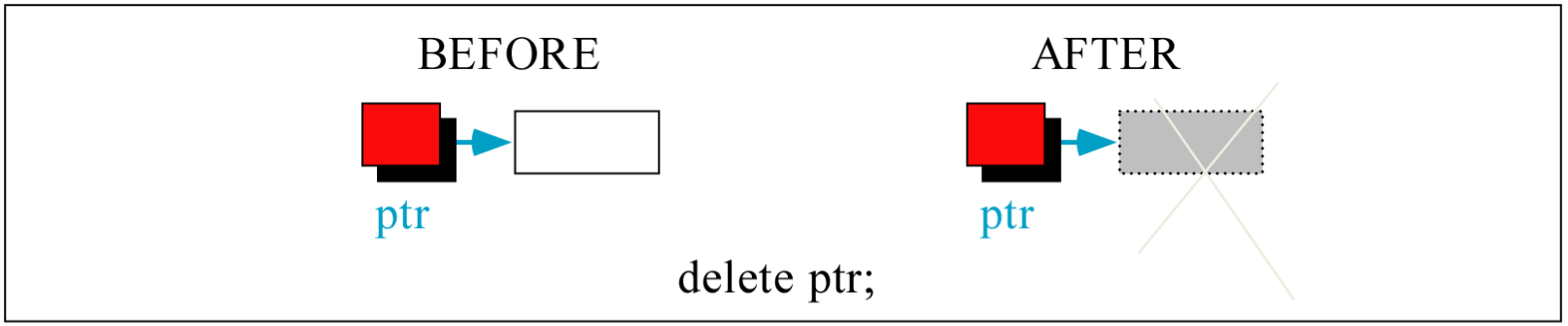


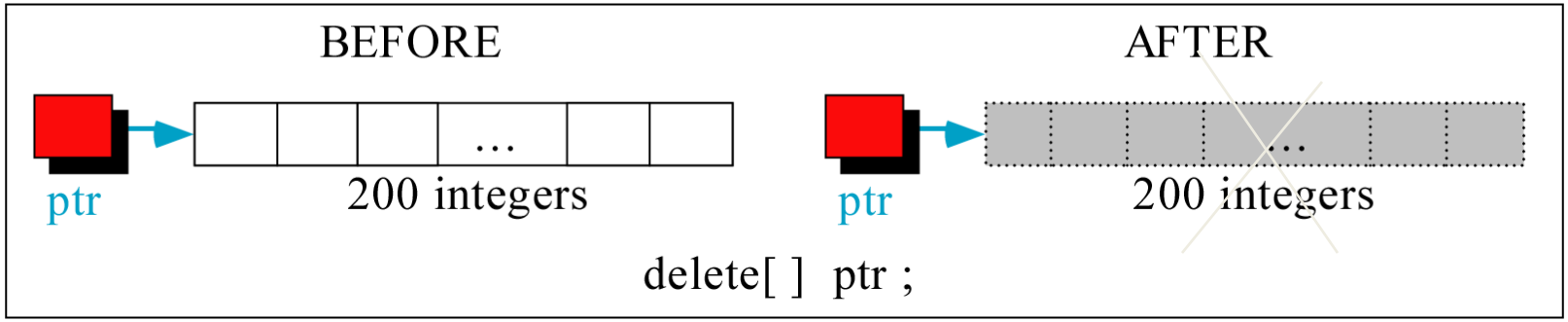
Memory Allocation Example

Want an array of unknown size

```
#include <iostream>
using namespace std;
void main()
{
      int n;
      cout <<  "How many students? ";
      cin   >> n;
      int *grades = new int[n];
      for(int i=0; i < n; i++){
          int mark;
          cout << "Input Grade for Student" << (i+1)  << " ? :";
          cin >> mark;
          grades[i] = mark;
      }
     . . .
     printMean( grades, n ); // call a function with dynamic array
     . . .
 }
```

Freeing (or deleting) Memory





A Simple Dynamic List Example

```
cout << "Enter list size: ";
int n;
cin >> n;
int *A = new int[n];
if(n<=0){
   cout << "bad size" << endl;
   return 0;
}
initialize(A, n, 0);  // initialize the array A with value 0
print(A, n);
A = addElement(A,n,5);  //add an element of value 5 at the end of A
print(A, n);
A = deleteFirst(A,n);  // delete the first element from A
print(A, n);
selectionSort(A, n);  // sort the array (not shown)
print(A, n);
delete [] A;
```

Initialize

```
void initialize(int list[], int size, int value){
   for(int i=0; i<size; i++)
       list[i] = value;

}
```

print()

```
void print(int list[], int size) {
   cout << "[ ";
   for(int i=0; i<size; i++)
      cout << list[i] << " ";
   cout << "]" << endl;
}
```

- **Remember in C++, array parameters are always passed by reference. That is,** void print(int list[], int size) {…} is the same as

  void print(int * list , int size) {…}

  Note: no & used here, so, the pointer itself is passed by value

Adding Elements

```
// for adding a new element to end of array
int* addElement(int list[], int& size, int value){
   int* newList = new int [size+1]; // make new array
   if(newList==0){
        cout << "Memory allocation error for addElement!" << endl;
        exit(-1);
   }
   for(int i=0; i<size; i++)
        newList[i] = list[i];
   if(size) delete [] list;
   newList[size] = value;
   size++;
   return newList;
}
```

Delete the first element

```
// for deleting the first element of the array
int* deleteFirst(int list[], int& size){
   if(size <= 1){
        if( size) delete list;
        size = 0;
        return NULL;
   }
   int* newList = new int [size-1]; // make new array
   if(newList==0){
        cout << "Memory allocation error for deleteFirst!" << endl;
        exit(-1);
   }
   for(int i=0; i<size-1; i++)    // copy and delete old array
        newList[i] = list[i+1];
   delete [] list;
   size--;
   return newList;
}
```

Adding Element (version 2)

```
// for adding a new element to end of array
// here "list" is a reference to a pointer variable: if the value of
   the pointer is changed in function, the change is global.
void addElement( int * & list, int & size, const int value ){

  int * newList = new int [size + 1];

  if( newList == NULL ){
    cout << "Memory allocation error for addElement!" << endl;
    exit(-1);
  }

  for( int i = 0; i < size; i++ )
    newList[ i ] = list[ i ];

  if( size ) delete [] list;

  newList[ size ] = value;
  size++;
  list = newList;
  return;
}
```

Deleting Element (version 2)

```
void deleteFirst( int * & list, int & size ){

  if( size <= 1 ){
   if( size )
     delete list;
   list = NULL;
   size = 0;
   return;
  }

  delete list; // delete the first element
  list++;
  size--;
  return;
}
```

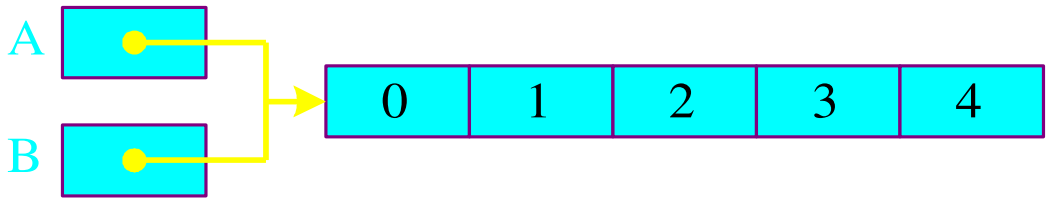
Another Main program

```
int main(){

  int * A = NULL;
  int size = 0;
  int i;

  for( i = 0; i < 10; i++ )
    addElement( A, size, i );

  for( i = 0; i < 10; i++ )
    cout << A[i] << " ";
  cout << endl;

  for( i = 0; i < 4; i++ )
    deleteFirst( A, size );

  for( i = 0; i < 6; i++ )
    cout << A[i] << " ";
  cout << endl;

  return 0;
}
```

0 1 2 3 4 5 6 7 8 9
4 5 6 7 8 9

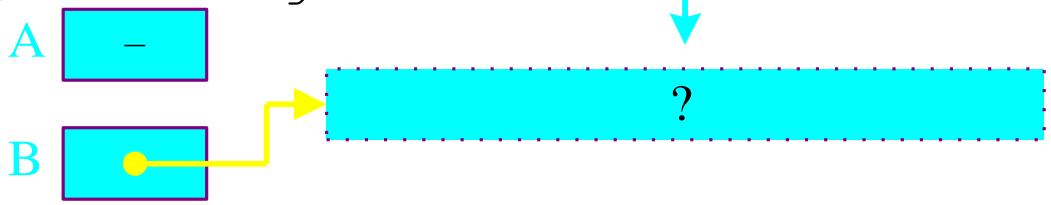
Dangling Pointer Problem

```
int *A = new int[5];
for(int i=0; i<5; i++)
   A[i] = i;
int *B = A;
```



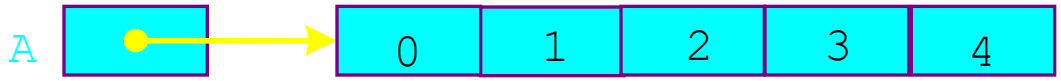

```
delete [] A;
B[0] = 1; // illegal!
```

Locations do not belong to program



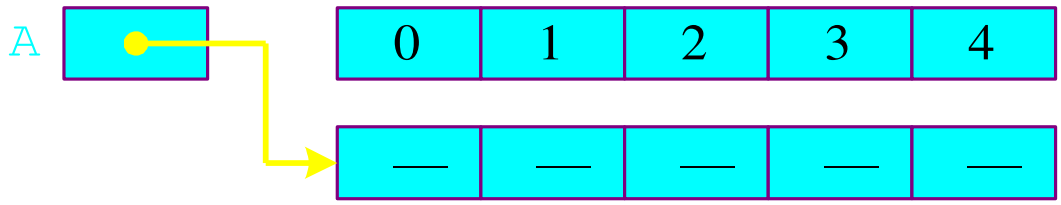
Memory Leak Problem

```
int *A = new int [5];
for(int i=0; i<5; i++)
    A[i] = i;
```



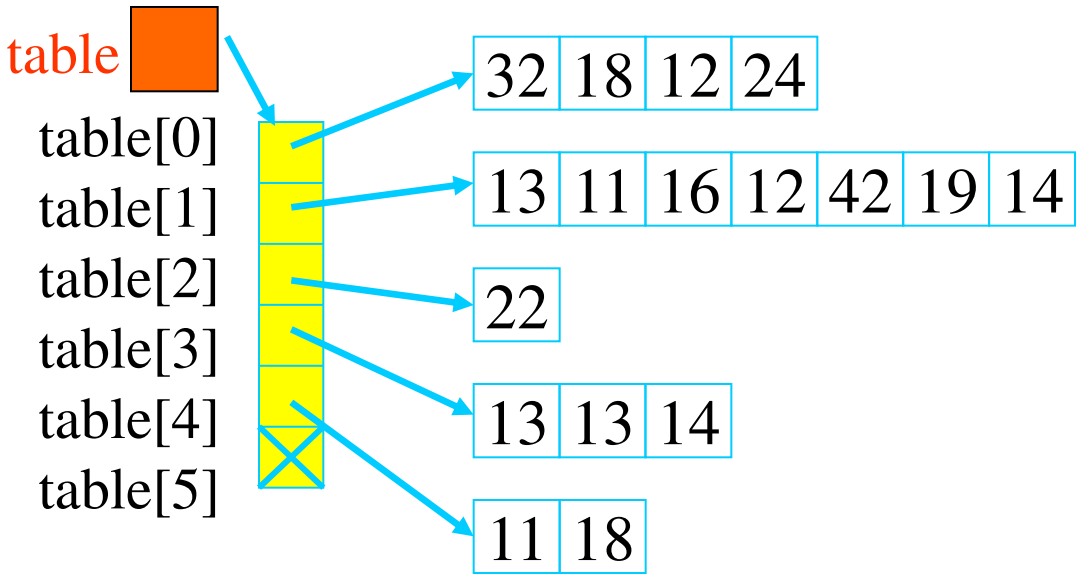

```
A = new int [5];
```

These locations cannot be accessed by program



A Dynamic 2D Array

- A dynamic array is an array of pointers to save space when not all rows of the array are full.
- `int **table;`




```
table = new int*[6];
…
table[0] = new int[4];
table[1] = new int[7];
table[2] = new int[1];
table[3] = new int[3];
table[4] = new int[2];
table[5] = NULL;
```

Memory Allocation

```
int **table;

table = new int*[6];

table[0]= new int[3];
table[1]= new int[1];
table[2]= new int[5];
table[3]= new int[10];
table[4]= new int[2];
table[5]= new int[6];

table[0][0] = 1; table[0][1] = 2; table[0][2] = 3;
table[1][0] = 4;
table[2][0] = 5; table[2][1] = 6; table[2][2] = 7; table[2][3]
= 8; table[2][4] = 9;

table[4][0] = 10; table[4][1] = 11;
cout << table[2][5] << endl;
```

Memory Deallocation

- Memory leak is a serious bug!
- Each row must be deleted individually
- Be careful to delete each row before deleting the table pointer.
  - `for(int i=0; i<6; i++)`

```
        delete [ ] table[i];
    delete [ ] table;
```

Create a matrix of any dimensions, m by n:

```
int m, n;
cin >> m >> n >> endl;

int** mat;

mat = new int*[m];

for (int i=0;i<m;i++)
  mat[i] = new int[n];
```

Put it into a function:

```
int m, n;
cin >> m >> n >> endl;
int** mat;
mat = imatrix(m,n);
…

int** imatrix(nr, nc) {
   int** m;
   m = new int*[nr];
   for (int i=0;i<nr;i++)
       m[i] = new int[nc];
   return m;
}
```