Introduction to Programming (2)

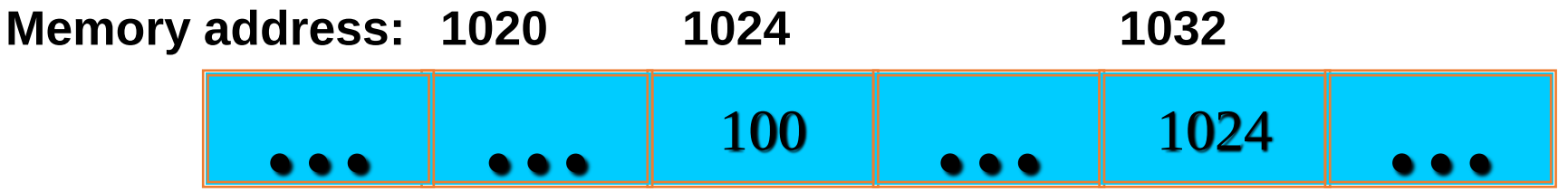
Pointers and Dynamic Objects

국립서울과학기술대학교 SEOULTECH

Laihyuk Park (lhpark@seoultech.ac.kr)

Computer Memory

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

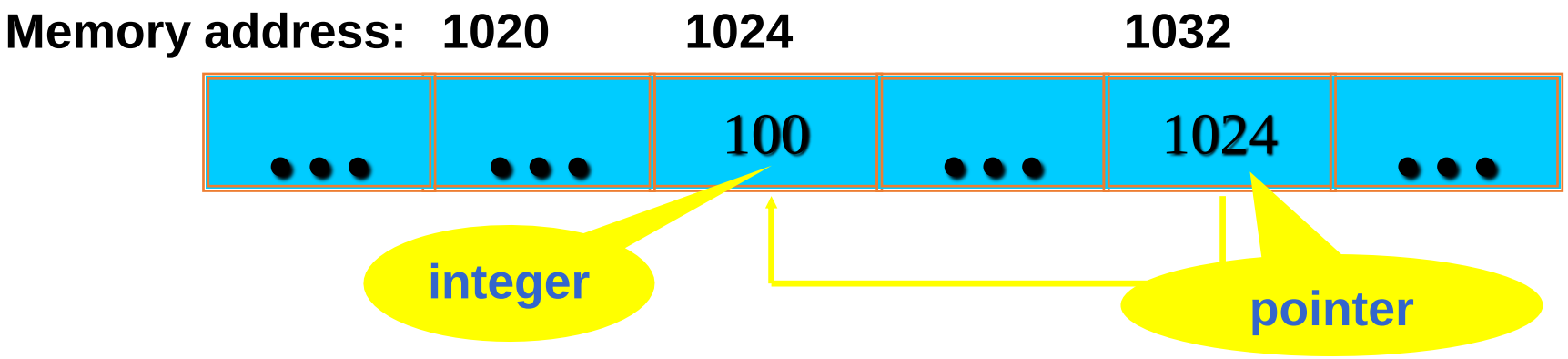



```
int a = 100;
```

Variable a's value, i.e., 100, is stored at memory location 1024

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., `Box`)
  - Even pointers to pointers
    - Pointers to pointers to `int` objects

Pointer Variable

- Declaration of Pointer variables

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

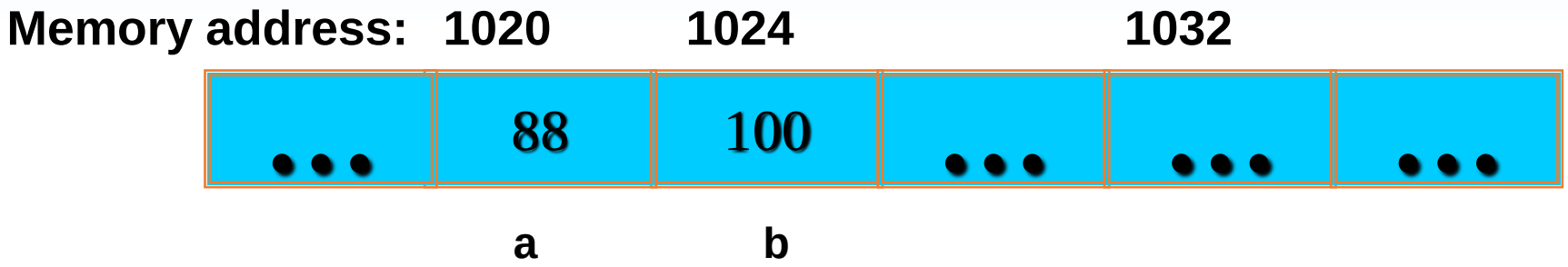
where *type* is the type of data pointed to (e.g. int, char, double)

Examples:

```
int *n;
Box *b;
int **p;     // pointer to pointer
```

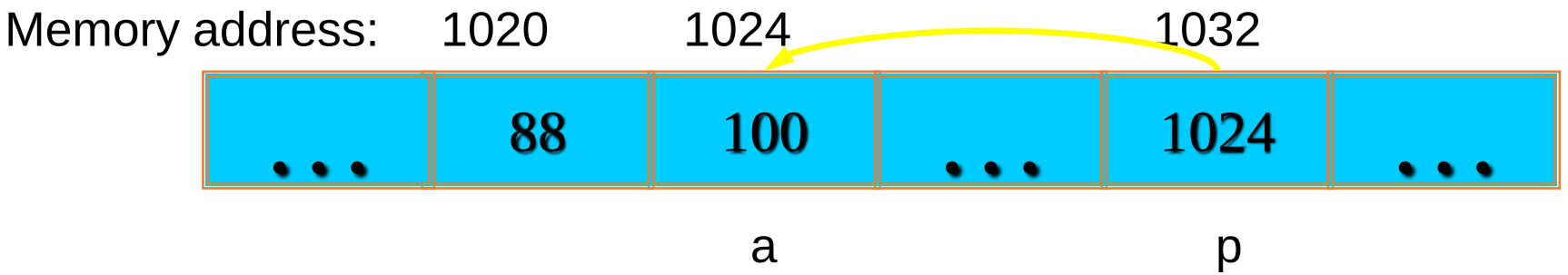

Laihyuk Park (lhpark@seoultech.ac.kr)

Address Operator &




```
#include <iostream>

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

Pointer Variables

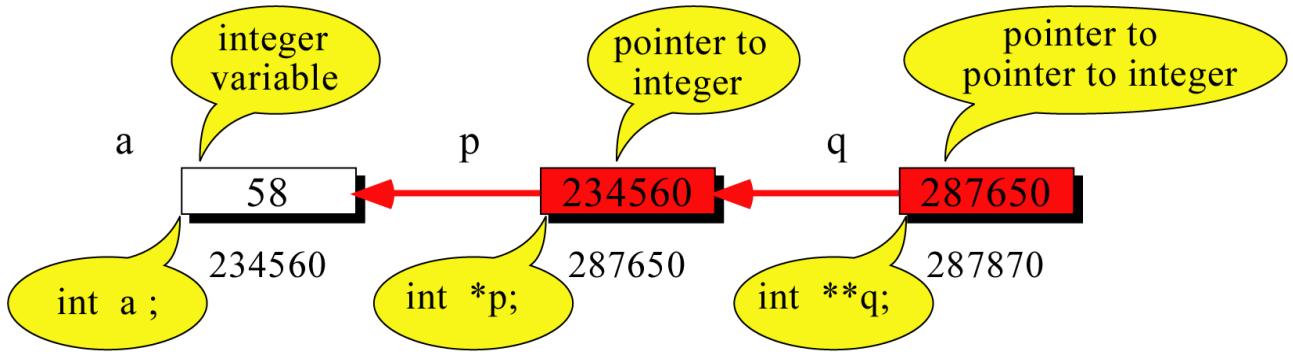



```
#include <iostream>

void main() {
    int a = 100;
    int* p = &a;
    std::cout << a << " " << &a << std::endl;
    std::cout << p << " " << &p << std::endl;
}
```

- 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

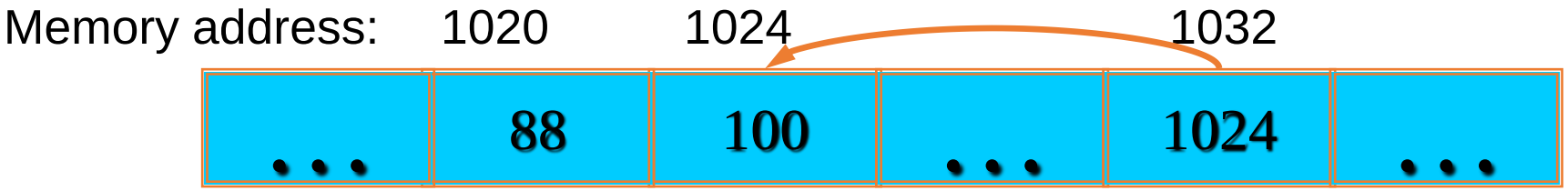
```
#include <iostream>

void main() {
    int a;
    int* p;
    int** q;

    a = 58;
    p = &a;
    q = &p;

    std::cout << a << " ";
    std::cout << *p << " ";
    std::cout << **q << " ";
}
```



- What is the output?

Dereferencing Operator *

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




```
#include <iostream>

void main() {
int a = 100;
int* p = &a;
    std::cout << a << std::endl;
    std::cout << &a << std::endl;
    std::cout << p << " " << *p << std::endl;
    std::cout << &p << std::endl;
}
```

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

```
#include <iostream>

void main() {
    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
    std::cout << a << b << c;
}
```

- What is the output?

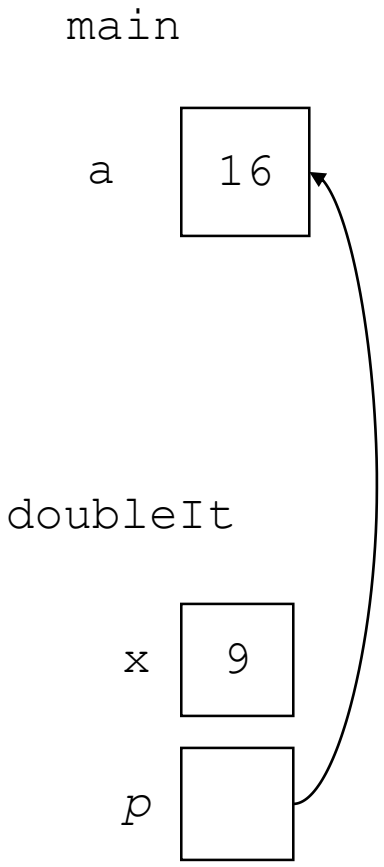
Pointer Example - 1

The code

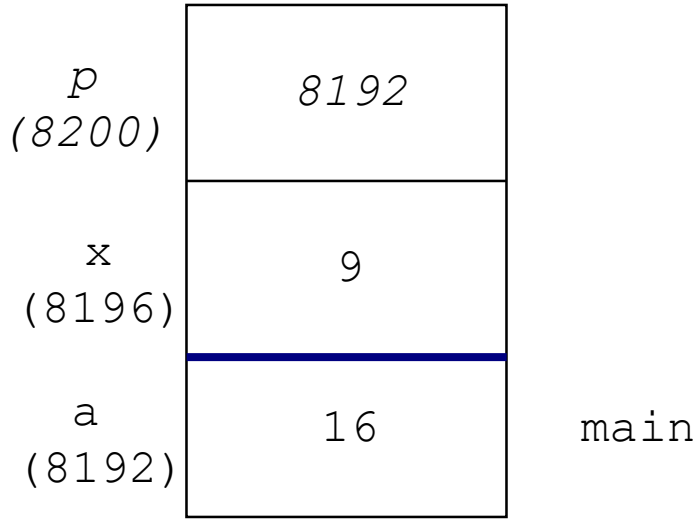
```
void doubleIt(int x,
              int * p)
{
  *p = 2 * x;
}
int main(int argc, const
  char * argv[])
{
  int a = 16;
  doubleIt(9, &a);
  return 0;
}
```

a gets 18

Box diagram



Memory Layout



Laihyuk Park (lhpark@seoultech.ac.kr)

Pointer Example - 2

- Let's figure out: value1==? / value2== ?
- Also, p1=? p2=?

```
#include <iostream>

void 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
    std::cout << "value1==" << value1 << "/ value2==" << value2;
}
```

Pointer Example - 3

- What is the output?

```
#include <iostream>

void main() {
    int a = 3;
    char s = 'z';
    double d = 1.03;
    int* pa = &a;
    char* ps = &s;
    double* pd = &d;
    //sizeof returns the # of bytes…
    std::cout << sizeof(pa) << sizeof(*pa) << sizeof(&pa) << std::endl;
    std::cout << sizeof(ps) << sizeof(*ps) << sizeof(&ps) << std::endl;
    std::cout << sizeof(pd) << sizeof(*pd) << sizeof(&pd) << std::endl;
}
```

Usage of const qualifier with pointers

- Pointers and the `const` Qualifier
  - `const` qualifier can apply to pointers in two ways.
    - Pointers to `const` Objects
    - `const` Pointers

```
#include <iostream>
void main() {
      const double d0 = 10.0;
      const double* ptr_c = &d0;
      *ptr_c = 20.0;     // Compilation error
                         // You cannot assign to a variable that is const
      double* ptr = &d0;    // Compilation error
                            // cannot convert from 'const double *' to 'double *'
}
```

Pointer to const object

```
#include <iostream>
void main() {
      double d0 = 10.0, d1 = 20.0;
      double* const ptr_c = &d0;

      ptr_c = &d1;// Compilation error
       // ptr_c is a constant pointer
}
```

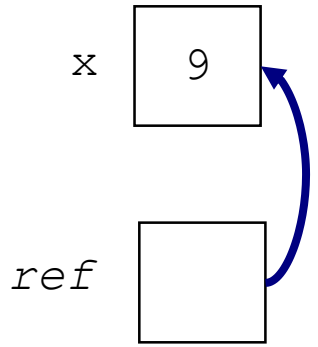
const Pointer

Laihyuk Park (lhpark@seoultech.ac.kr)

Reference Variables

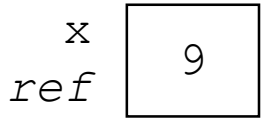
A reference is an additional name to an existing memory location

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

```
#include <iostream>

void main() {
    int m = 10;
    int& j = m;  // j is a reference variable
    std::cout << "value of m = " << m << std::endl;
    //print 10
    j = 18;
    std::cout << "value of m = " << m << std::endl;
    // print 18
}
```

- 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

Pointer vs. Reference

- A reference must refer a pre-existing object.
  - To define a reference without initializing it is an error.

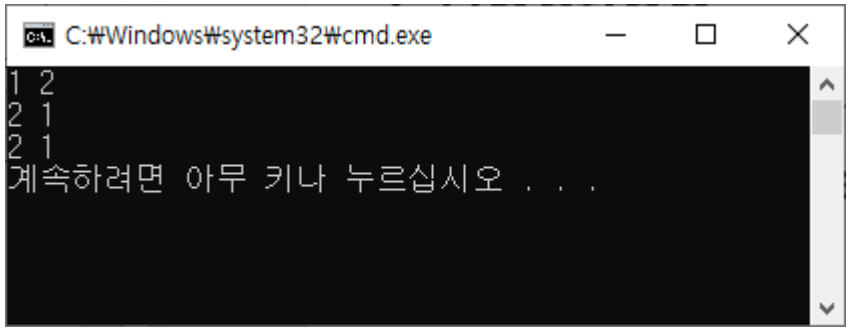
```
#include <iostream>

void main() {
    double* d_ptr;// No problem
    double& d_ref;  // Compilation Error
                    // reference must be initialized
}
```

- Note that assignment is done differently.

```
#include <iostream>

void main() {
    double d0 = 10.0, d1 = 10.0;
    double* d_ptr = &d0;
    double& d_ref = d0;

    d_ptr = &d1;  // d_ptr now points to d1
    d_ref = 20.0; // assign 20.0 to d0
}
```

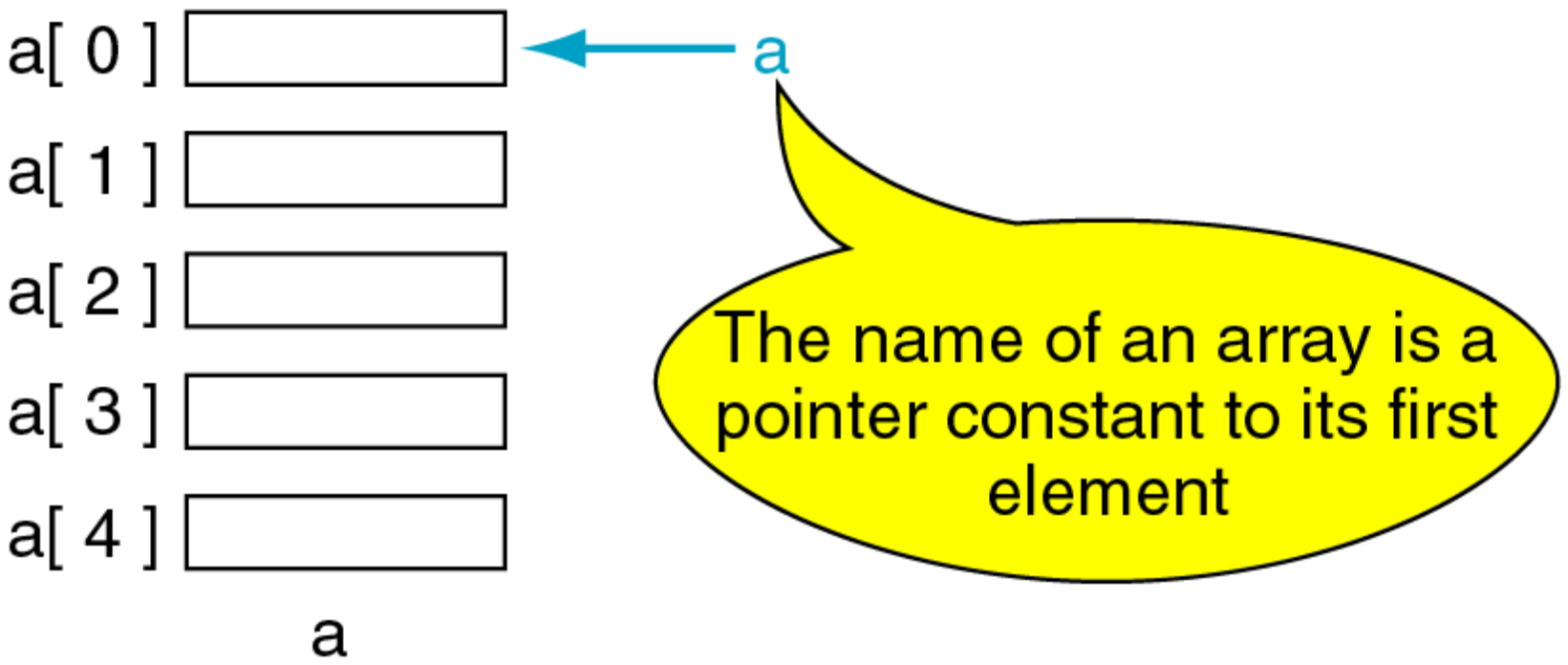
Call by value, call by reference, call by pointer

```
#include <iostream>

void swap_using_value(int a, int b) { int tmp = a; a = b; b = tmp; }
void swap_using_ref(int& a, int& b) { int tmp = a; a = b; b = tmp; }
void swap_using_ptr(int* a, int* b) { int tmp = *a; *a = *b; *b = tmp; }

void main() {
    int u, v;
    u = 1, v = 2; swap_using_value(u, v);
    std::cout << u << " " << v << std::endl;
    u = 1, v = 2; swap_using_ref(u, v);
    std::cout << u << " " << v << std::endl;
    u = 1, v = 2; swap_using_ptr(&u, &v);
    std::cout << u << " " << v << std::endl;
}
```



Pointers and Arrays

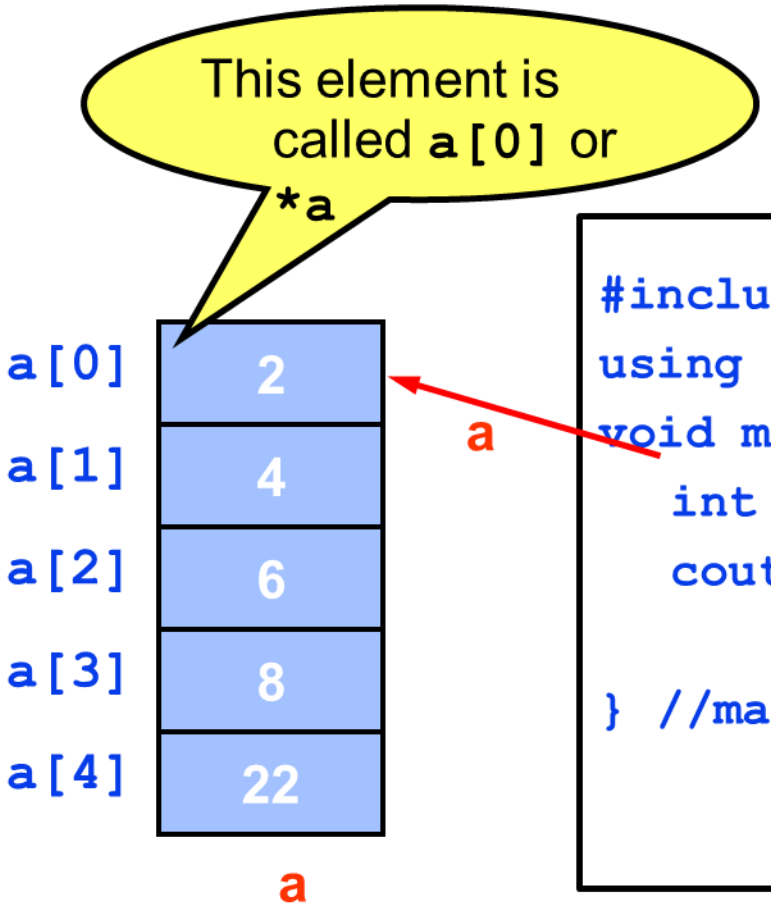
- The name of an array points only to the first element not the whole array.



Array name is a pointer constant

```
#include <iostream>

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

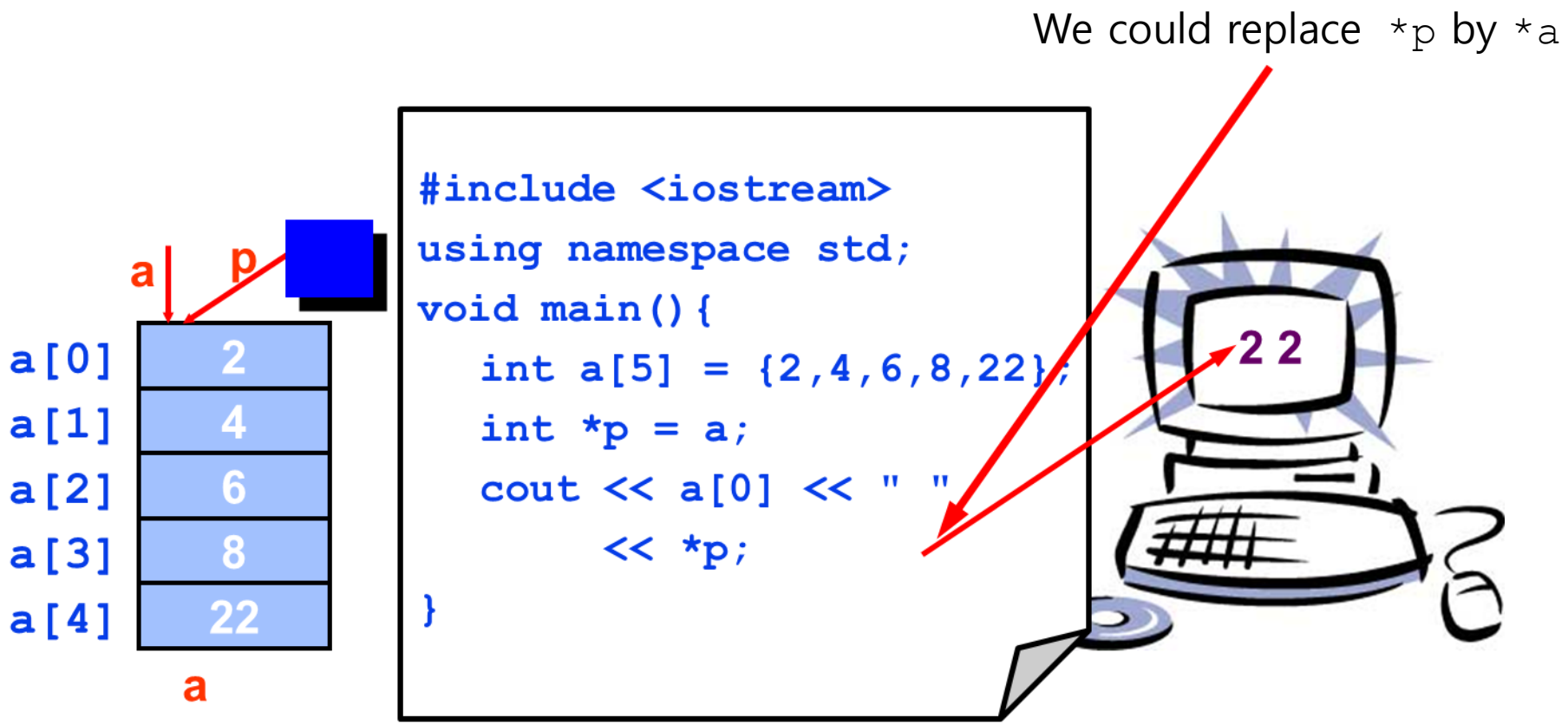
Dereferencing an Array Name




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

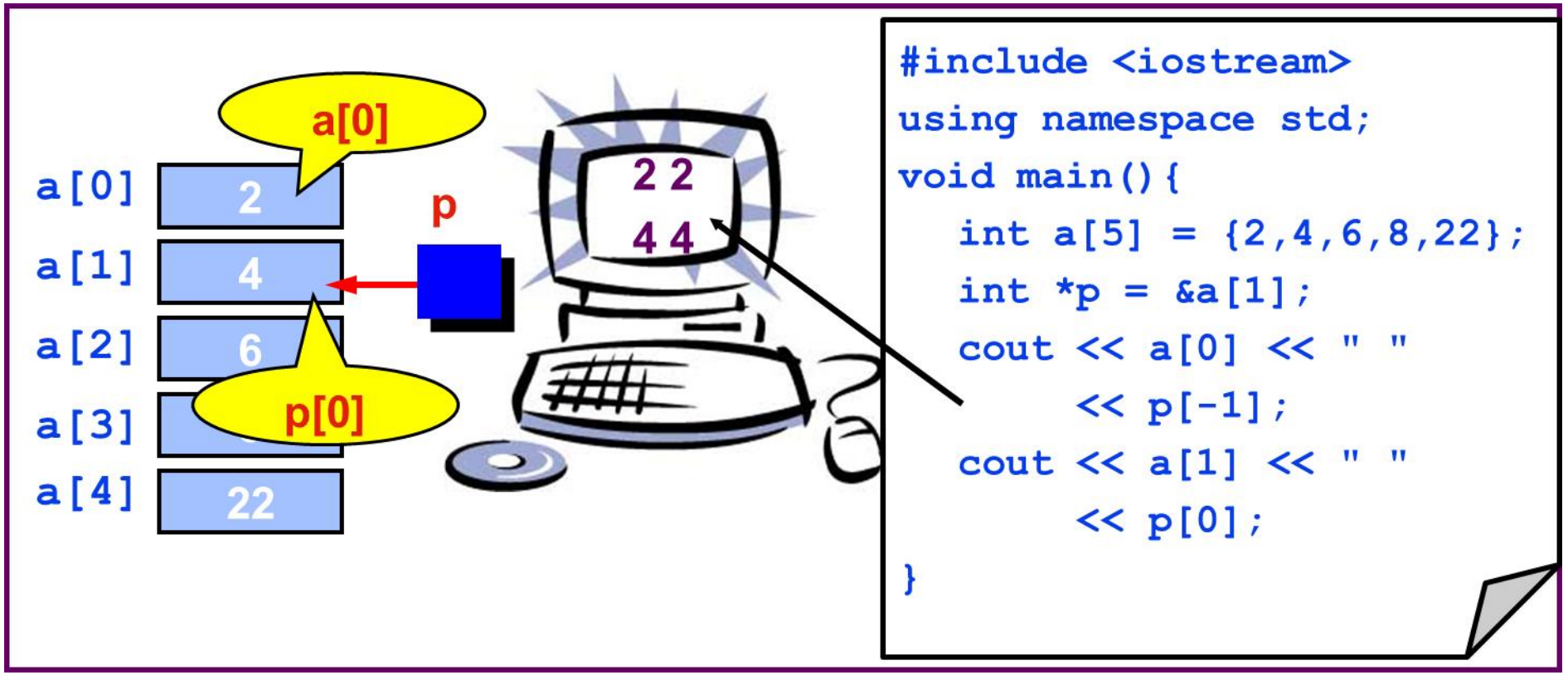
Array Name as Pointers

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



Multiple Array Pointers

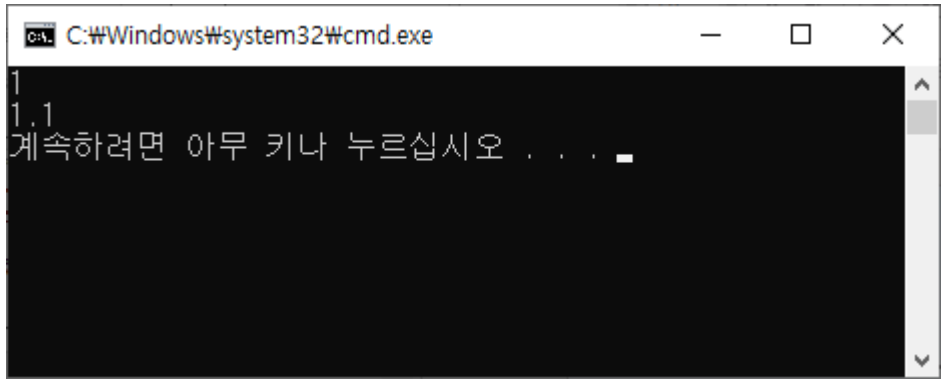
- Both `a` and `p` are pointers to the same array.



Pointer Arithmetic

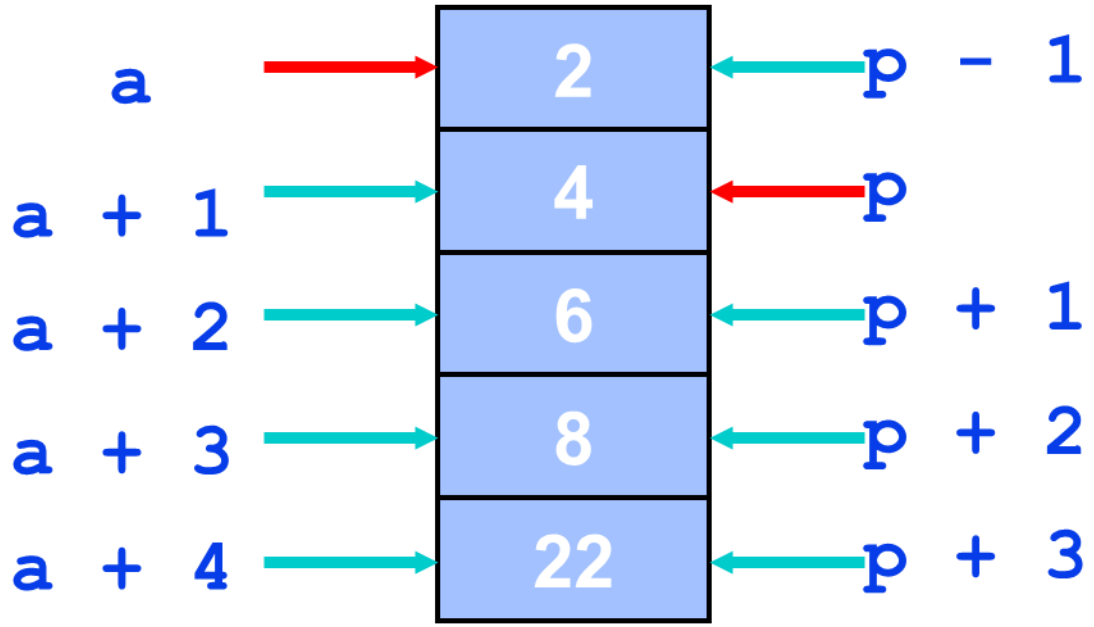
- Adding an integer n to a pointer variable x does not produce the address displaced from n **bytes** from x, but produces the address displaced n **elements** of the data type from x.

```
#include <iostream>

void main() {
    int ia[] = { 6,5,4,3,2,1 };
    int* ptr_i = &ia[3];

    double da[] = { 6.1,5.1,4.1,3.1,2.1,1.1,0.1 };
    double* ptr_d = &da[3];

    std::cout << *(ptr_i + 2) << std::endl;
    std::cout << *(ptr_d + 2) << std::endl;
}
```

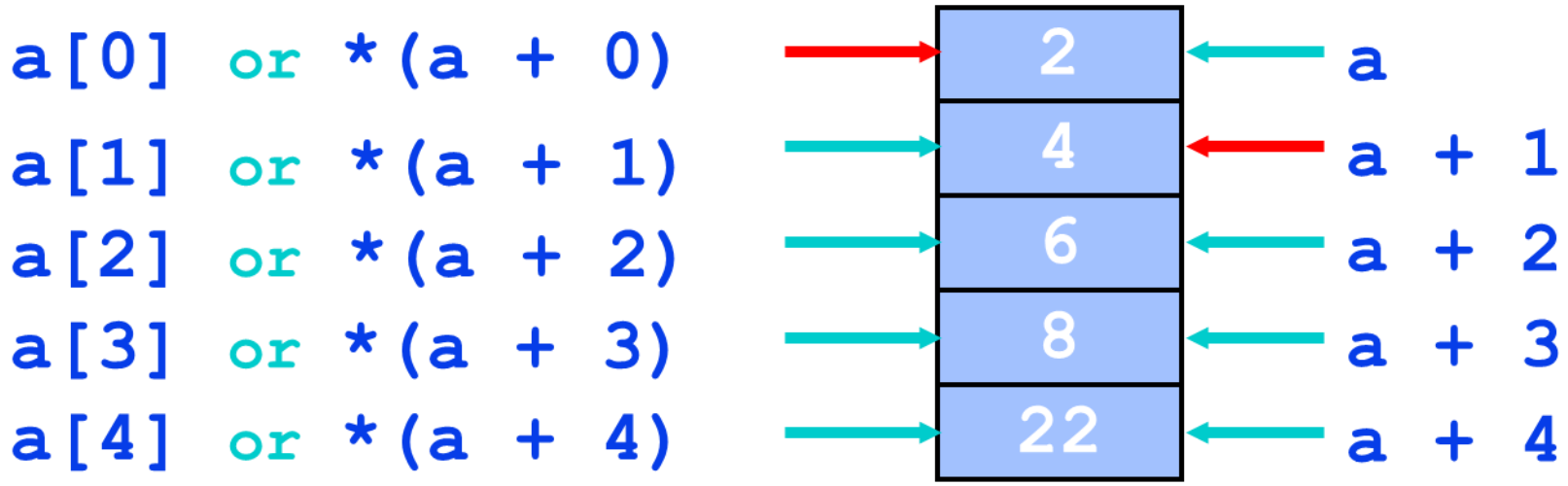


Pointer Arithmetic

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



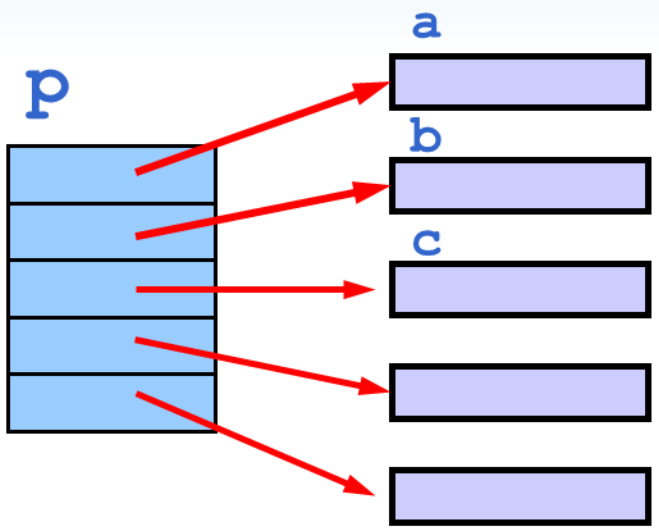
Dereferencing Array Pointers



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

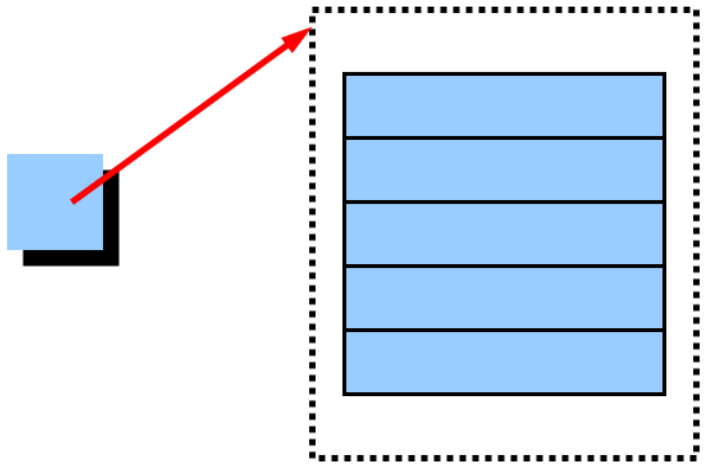
Note: flexible pointer syntax

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 Pointers

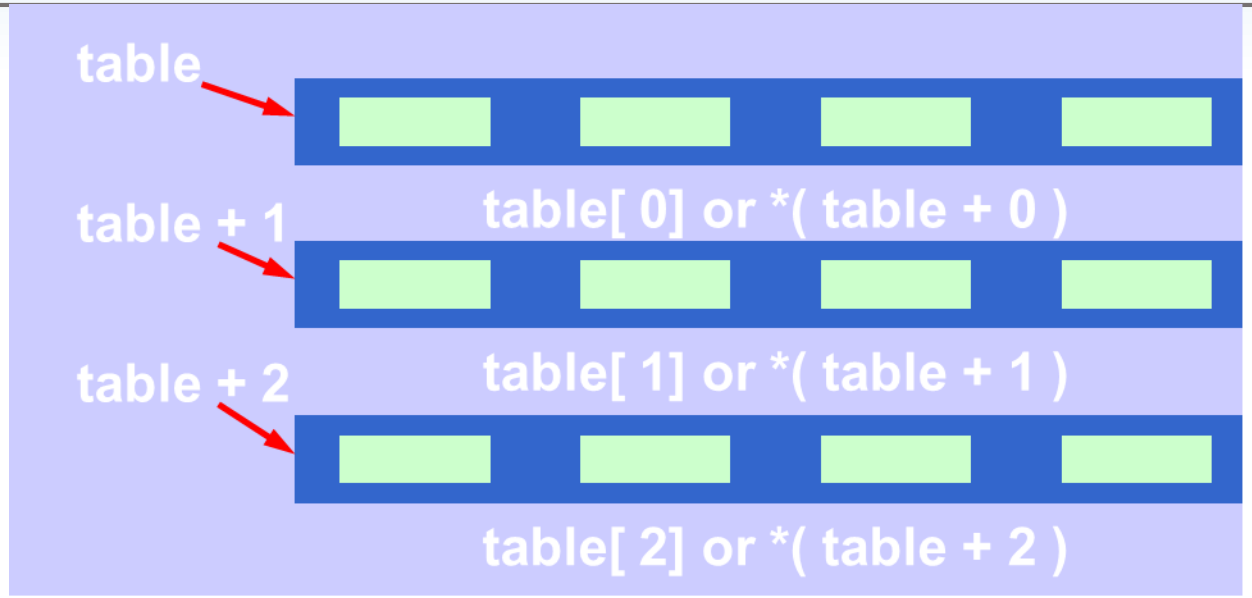
- NULL is a special value that indicates an empty pointer
- If you try to access a NULL pointer, you will get an error

```
#include <iostream>

void main() {
    int* p;
    p = 0;
    std::cout << p << std::endl; //prints 0
    std::cout << &p << std::endl;//prints address of p
    std::cout << *p << std::endl;//Error!
}
```

Storing 2D Array in 1D Array

```
#include <iostream>

void main() {
    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

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

What is `**table` ?

```
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]`

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

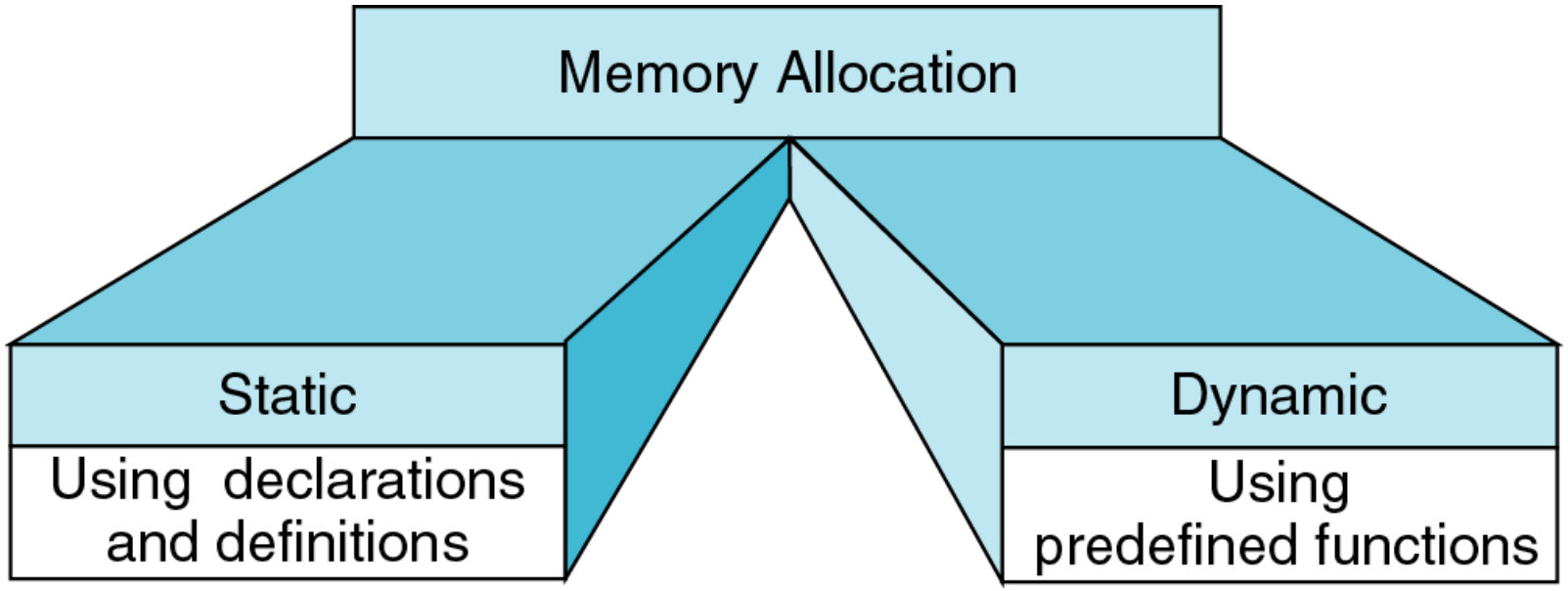
  (variables as declared in function calls)

  - 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

Laihyuk Park (lhpark@seoultech.ac.kr)

Memory Allocation



new
delete

```
{
   int a[200];
   …
}
```

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

Object (variable) Creation : New

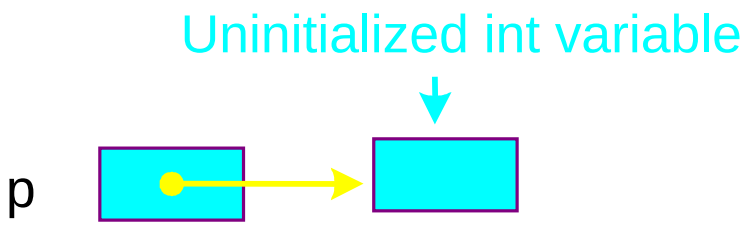
Syntax

```
ptr = new SomeType;
```

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

Example

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



Object (variable) Destruction :Delete

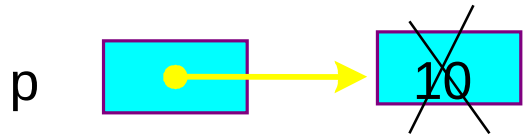
Syntax

```
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 arr ay

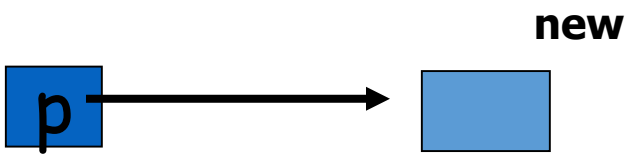
- Because of the flexible pointer syntax, `P` can be considered to be an array

Example

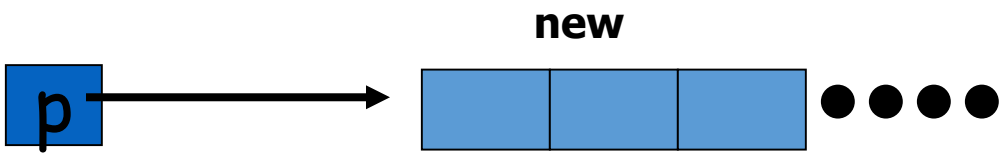
Dynamic Memory Allocation

- Request for “unnamed” memory from the Operating System

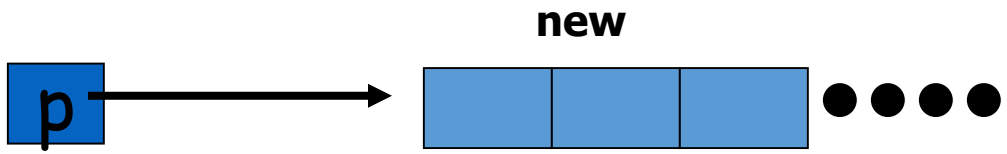
```
int *p, n=10;
p = new int;
```



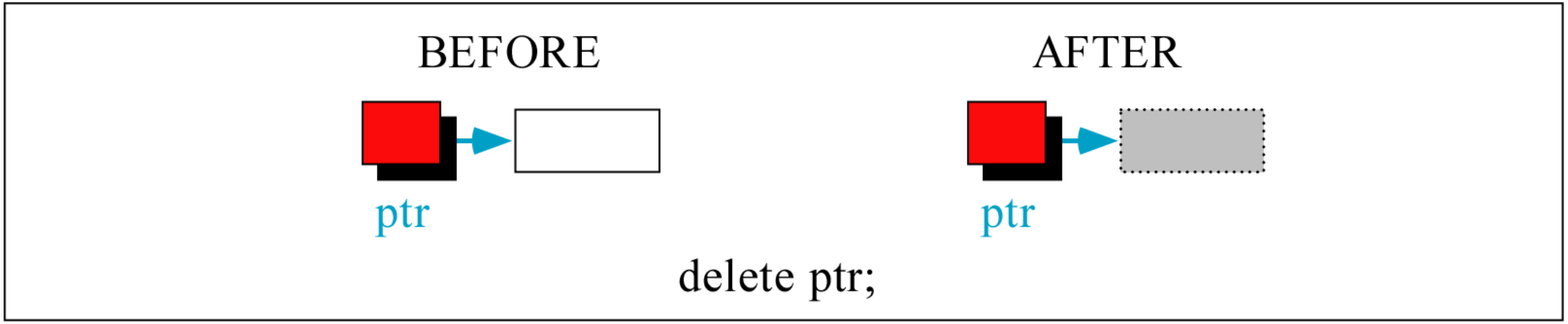

```
p = new int[100];
```

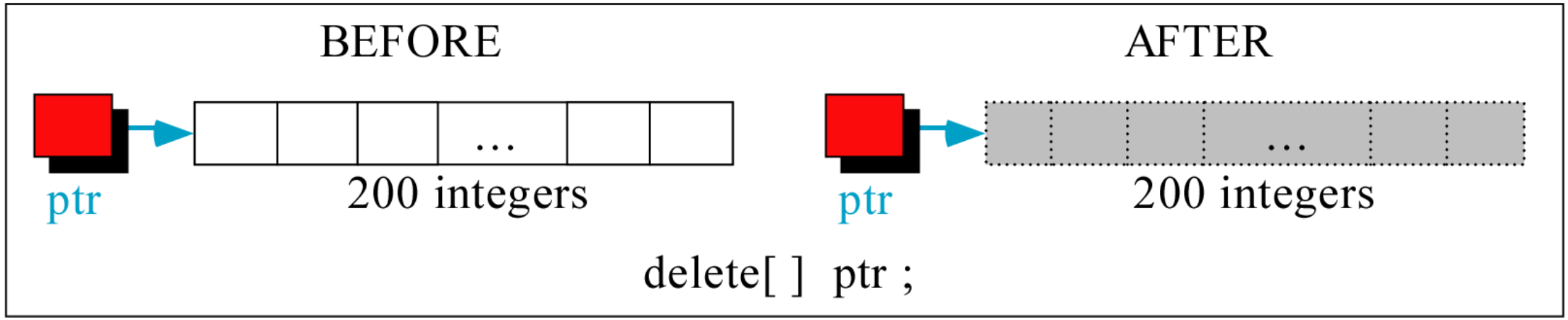



```
  p = new int[n];
```



Freeing (or Deleting) Memory





A Simple Dynamic List Example

```
int main()
{
    std::cout << "Enter list size: ";
    int n;
    std::cin >> n;
    int* A = new int[n];
    if (n <= 0) {
        std::cout << "bad size" << std::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);
    delete[] A;
    return 0;
}
```

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) {
    std::cout << "[ ";
    for (int i = 0; i < size; i++)
        std::cout << list[i] << " ";
    std::cout << "]" << std::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

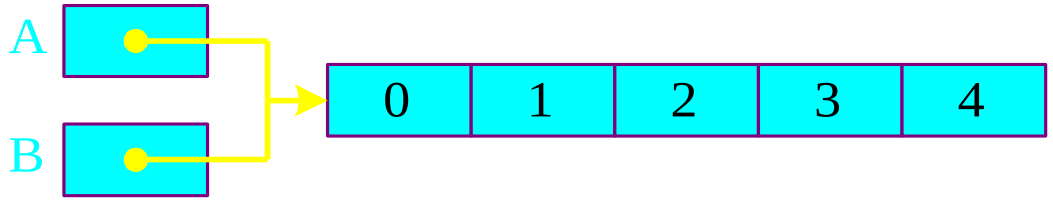
```
// for adding a new element to end of array
int* addElement(int list[], int& size, int value) {
    int* newList = new int[size + 1]; // make new array
    if (newList == 0) {
        std::cout << "Memory allocation error for addElement!" << std::endl;
        exit(-1);
    }
    for (int i = 0;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

Deleting the first element

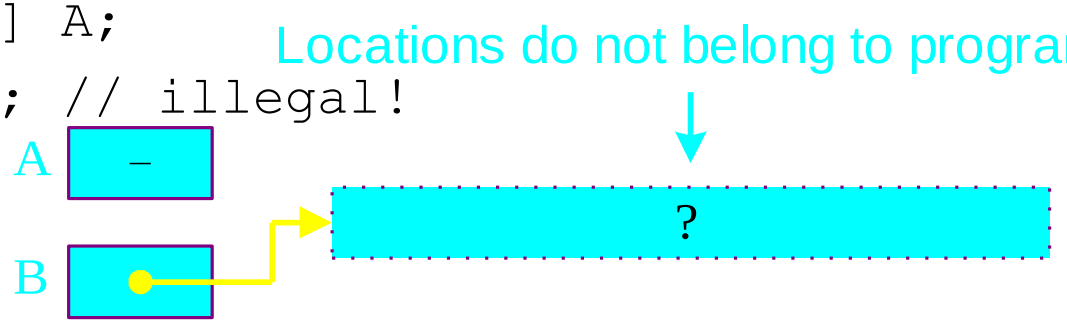
```
// for deleting the first element of the array
int* deleteFirst(int list[], int& size) {
    if (size <= 1) {
        if (size) delete list;
                size = 0;
        return NULL;
    }
    int* newList = new int[size - 1]; // make new array
    if (newList == 0) {
        std::cout << "Memory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

Dangling Pointer Problem

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

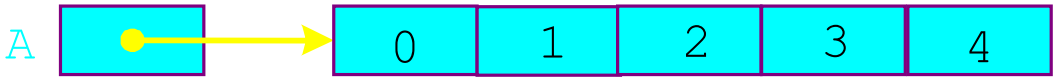



```
delete [] A;
B[0] = 1; // illegal!
```



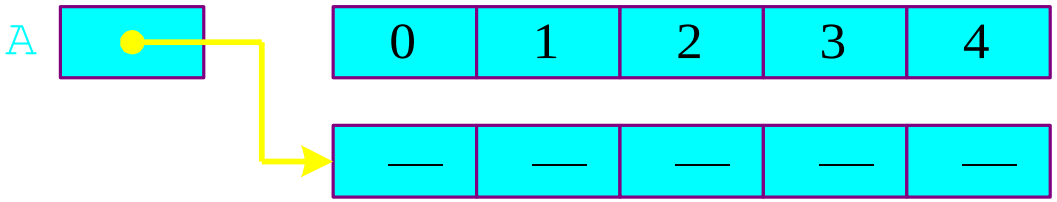
Memory Leak Problem

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




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

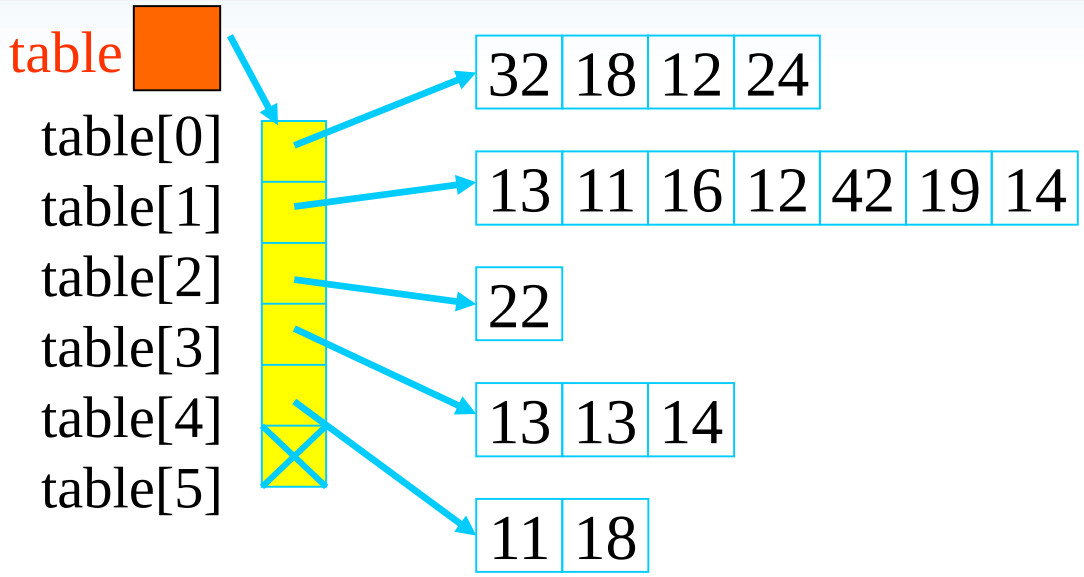
These locations cannot be accessed by program



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 main(){
    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;
    std::cout << table[2][5] << std::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;
```