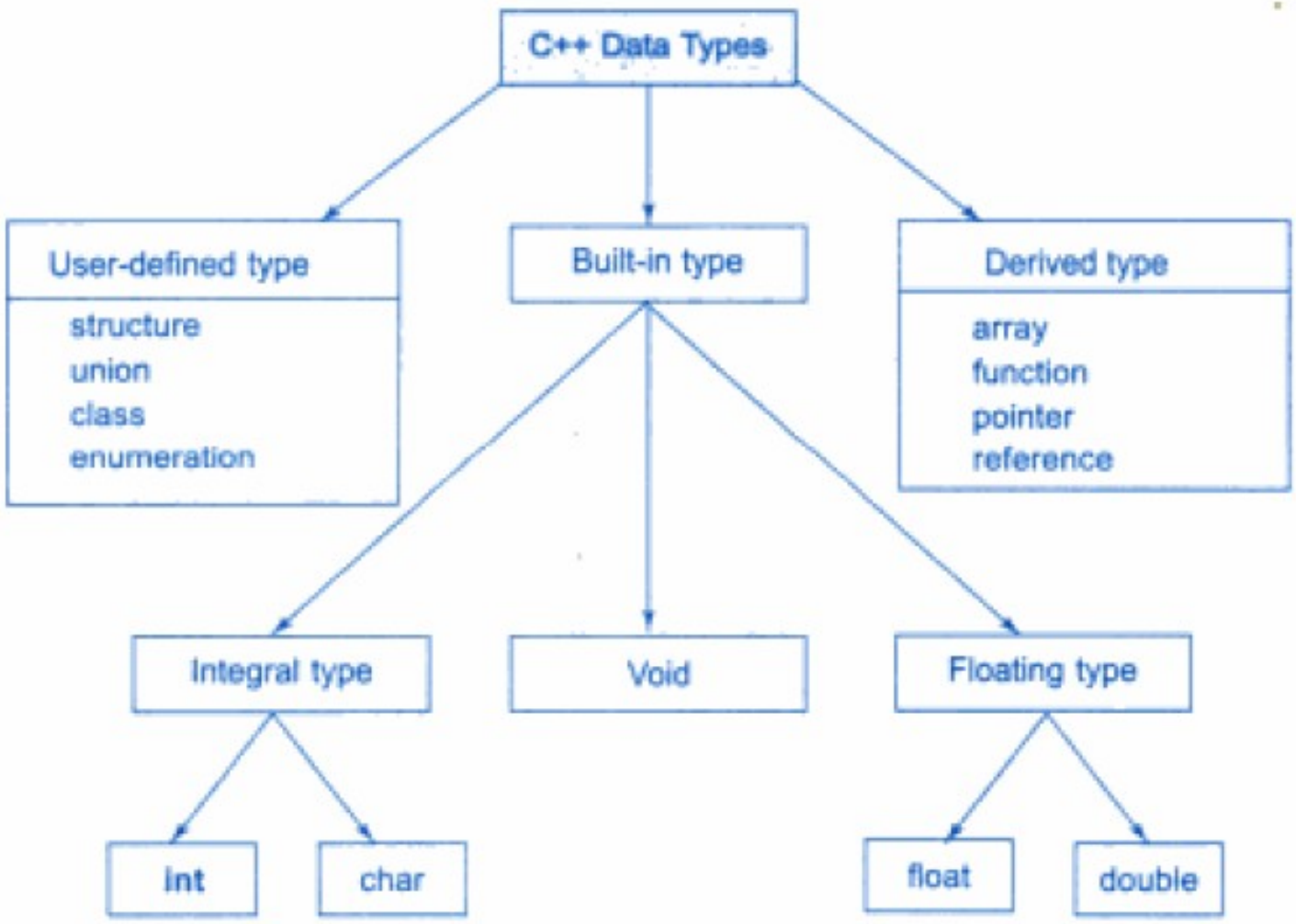
Pointer data type

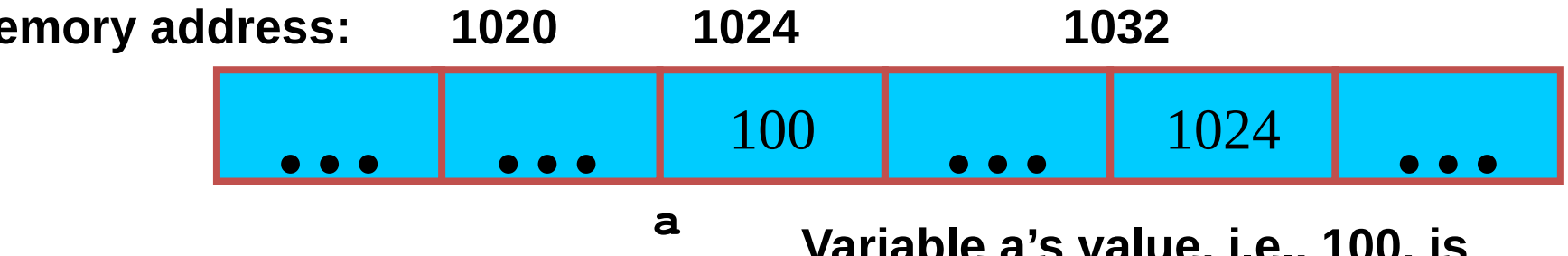


Topics

- 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

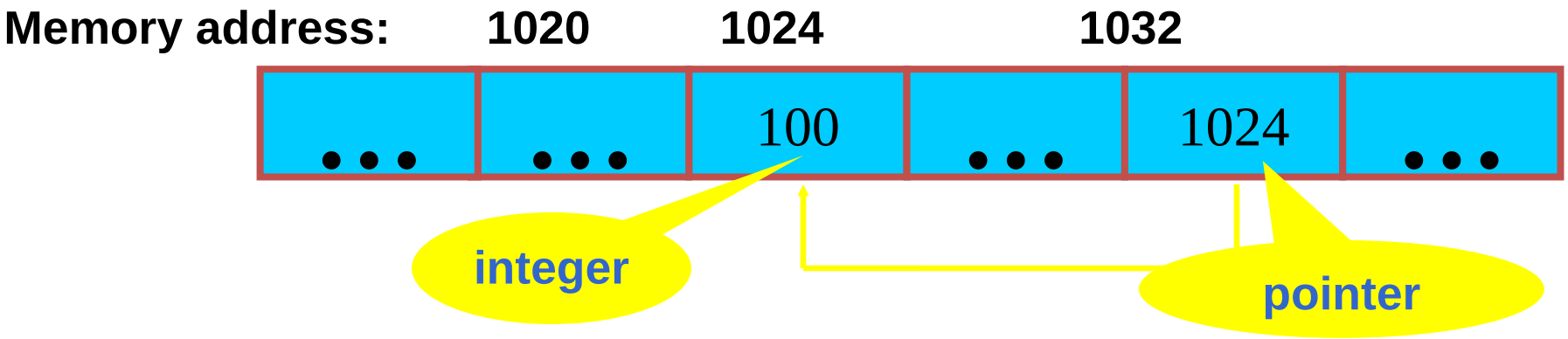



```
int a = 100;
```

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

Pointer Variable

- Declaration of Pointer variables

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

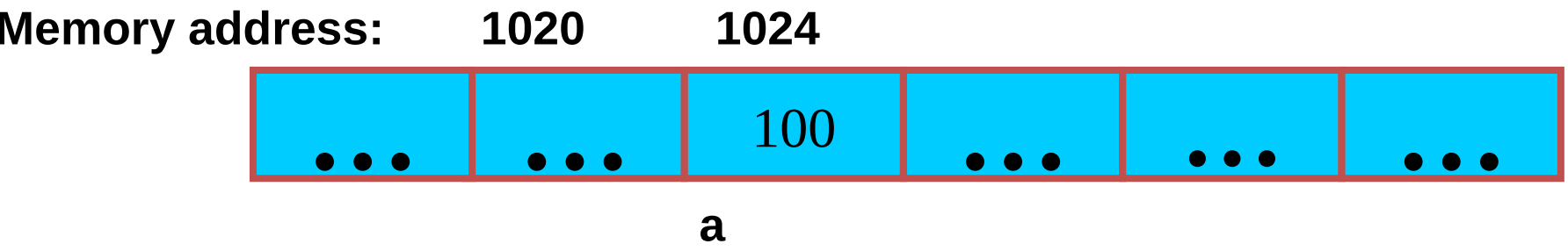
where *type* is the type of data pointed to (e.g. int, char, double)

Examples:

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

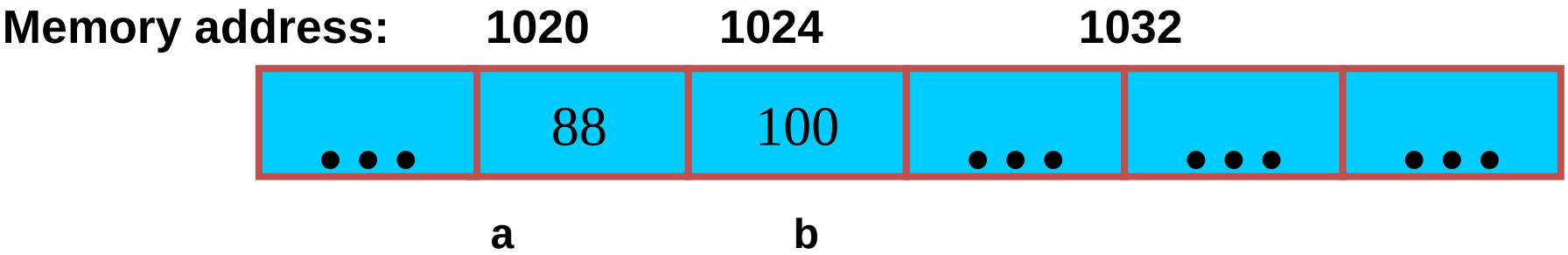
Address Operator &

- *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 &



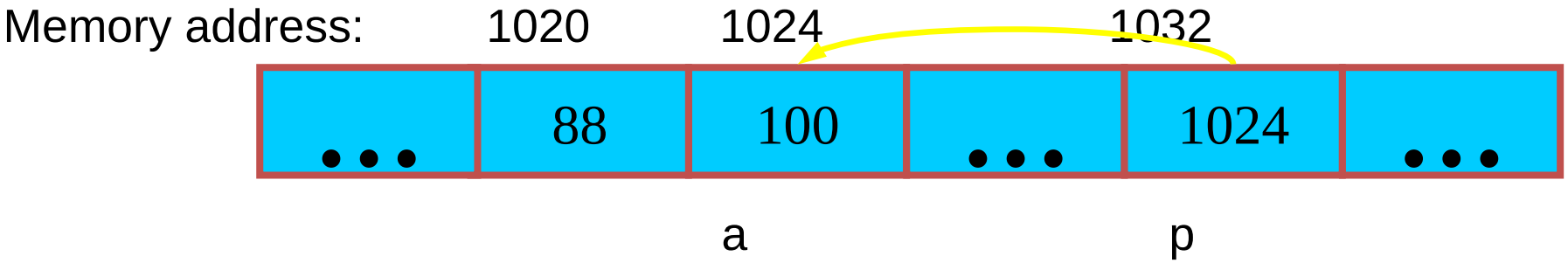

```
#include <iostream>
using namespace std;
void main(){
   int a, b;
   a = 88;
   b = 100;
   cout << "The address of a is: " << &a << endl;
   cout << "The address of b is: " << &b << endl;
}
```

- 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:

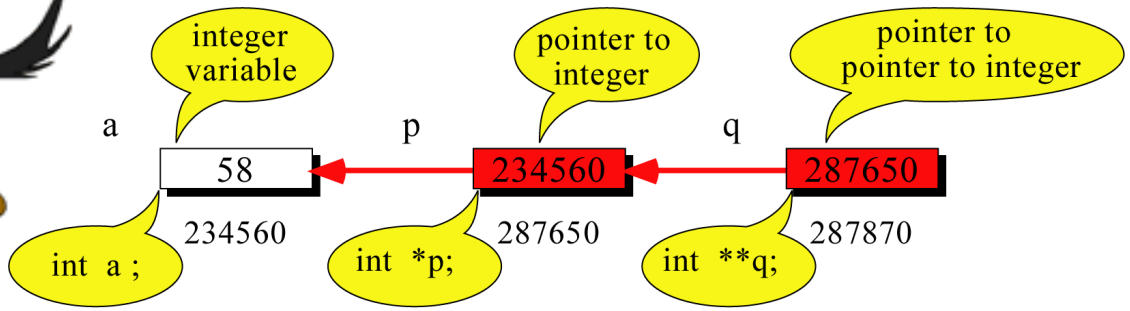
100 1024

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 ;
```



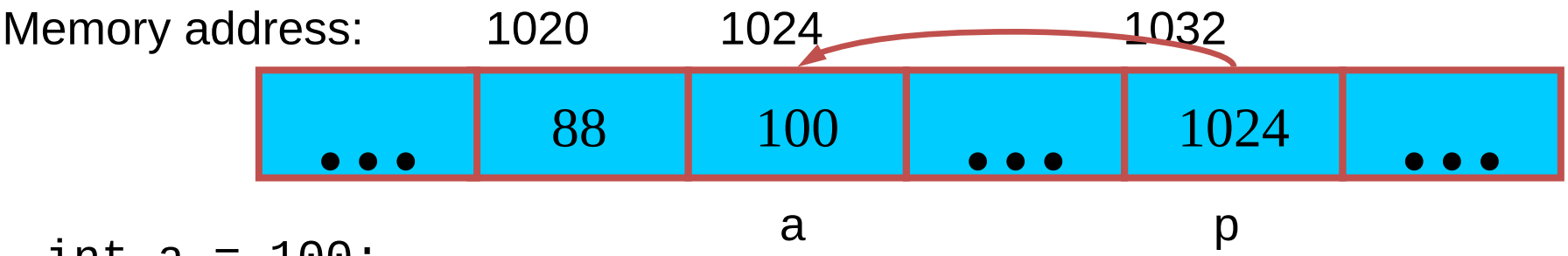

```
// Statements
a = 58 ;
p = &a ;
q = &p ;
cout <<     a << " ";
cout <<    *p << " ";
cout <<   **q << " ";
```

What is the output?

58 58 58

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

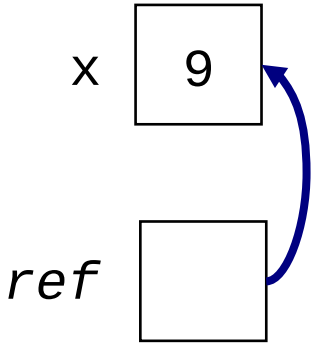
```
int a = 100, b = 88, c = 8;
int *p1 = &a, *p2, *p3 = &c;
p2 = &b; // p2 points to b
p2 = p1; // p2 points to a
b = *p3; //assign c to b
*p1 = *p3;  //assign c to a
cout << a << b << c;
```

- **Result is: 888**

Reference Variables

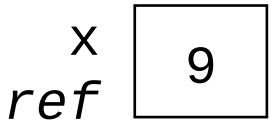
A reference is an additional name to an existing memory location

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

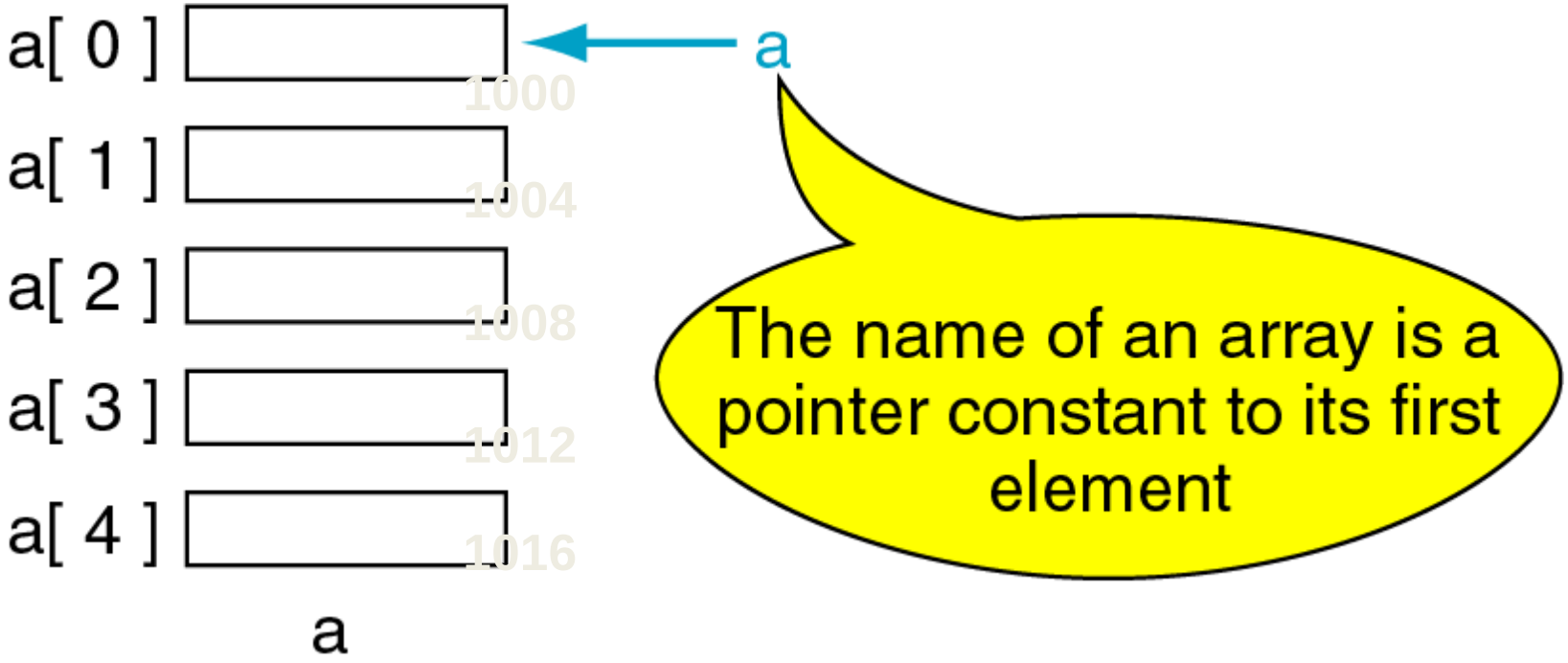
- A reference variable always refers to the same object. Assigning a reference variable with a new value actually changes the value of the referred object.
- Reference variables are commonly used for parameter passing to a function

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

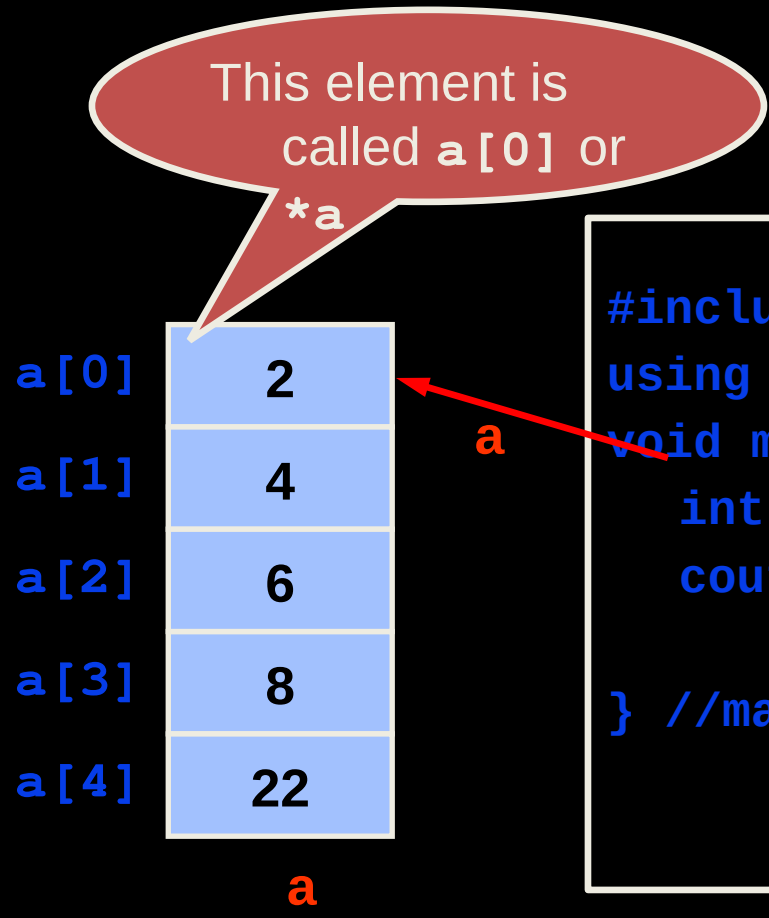
```
#include <iostream>
using namespace std;

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

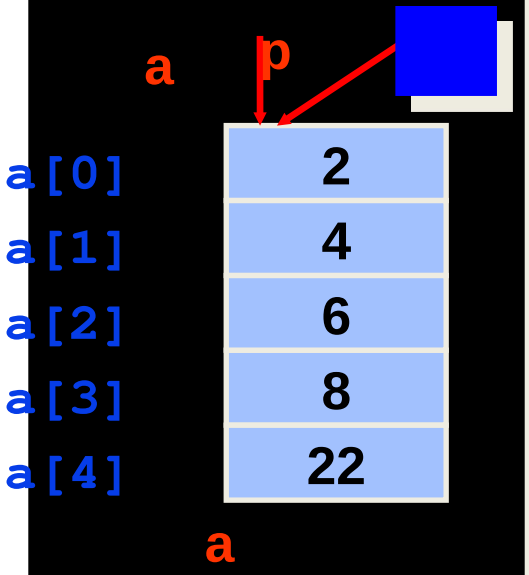



```
#include <iostream>
using namespace std;
void main(){
   int a[5] = {2,4,6,8,22};
   cout << *a << " "
        << a[0];
} //main
```

Array Names as Pointers

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

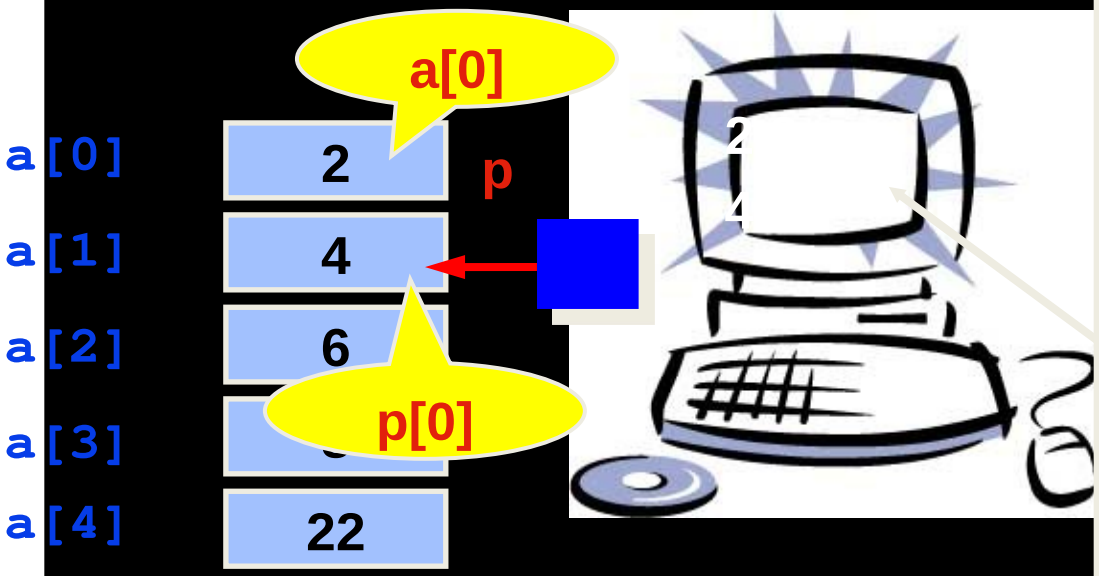
We could replace *p by *a




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

Multiple Array Pointers

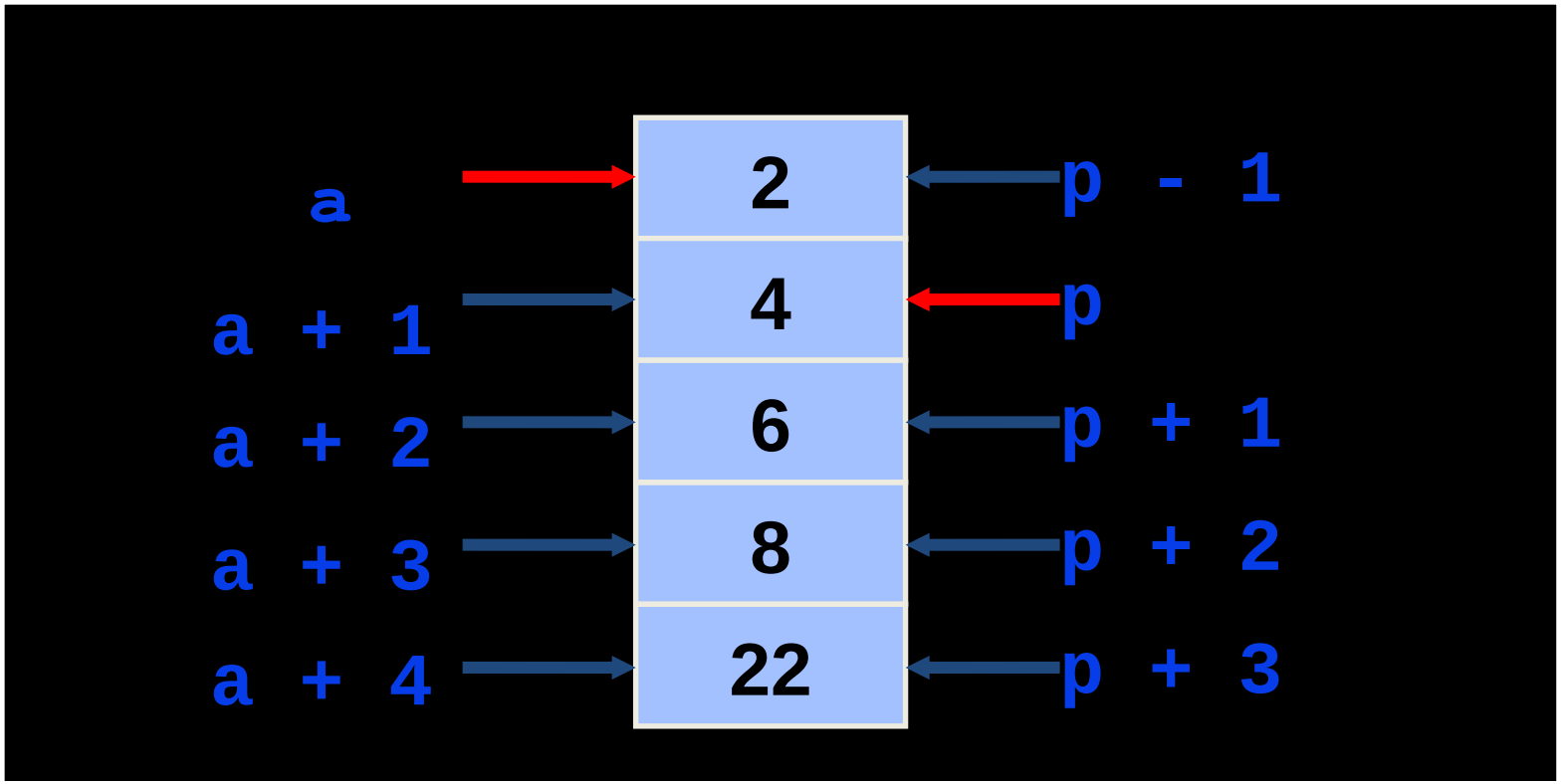
- Both `a` and `p` are pointers to the same array.



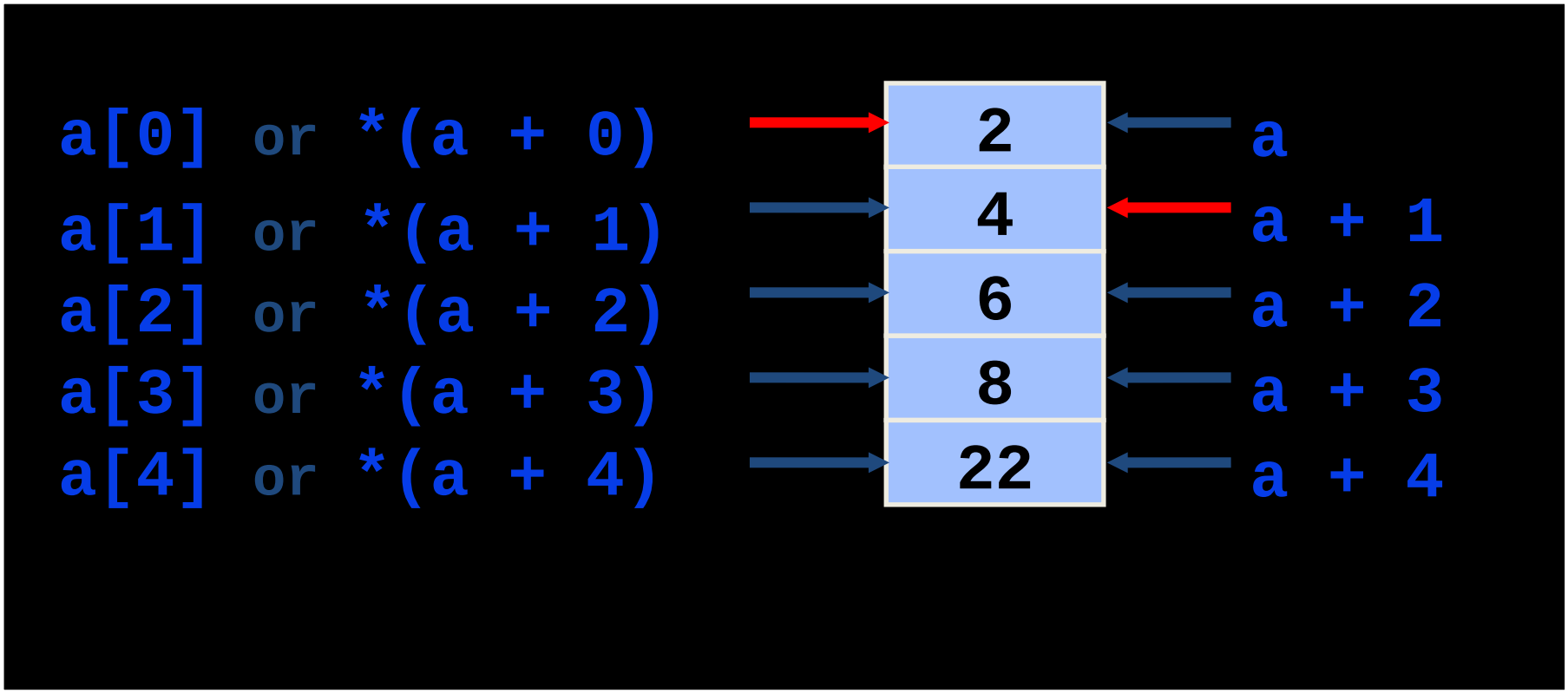

```
#include <iostream>
using namespace std;
void main(){
   int a[5] = {2,4,6,8,22};
   int *p = &a[1];
   cout << a[0] << " "
        << p[-1];
   cout << a[1] << " "
        << p[0];
}
```

Pointer Arithmetic

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



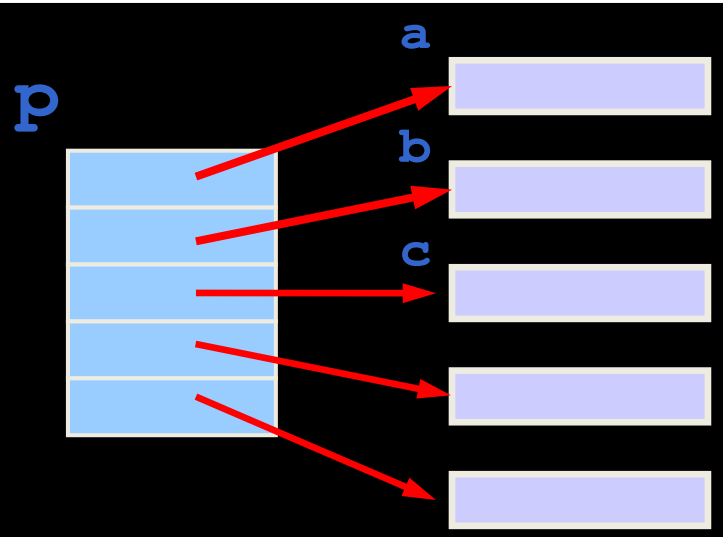
Dereferencing Array Pointers



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

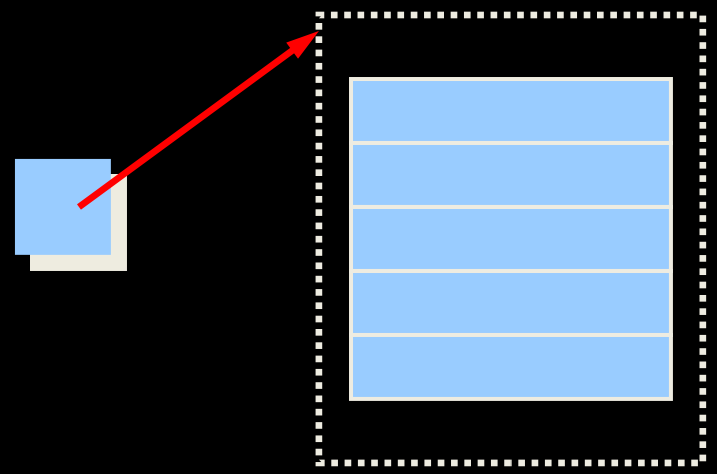
- Note: flexible pointer syntax

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
```