COMP 2710
Software Construction

Chapter 8
Pointer and Dynamic Objects

AUBURN
UNIVERSITY

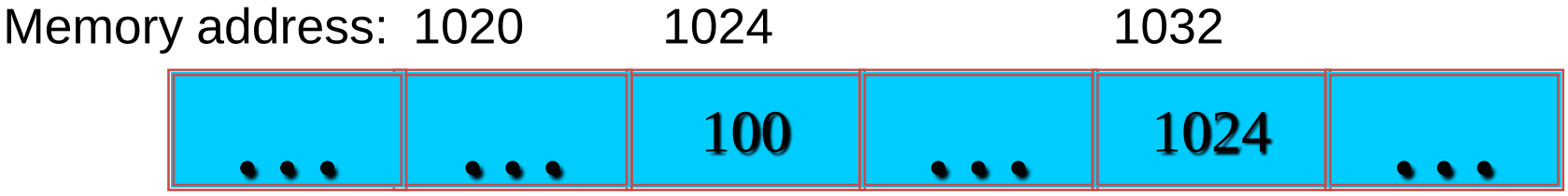
SAMUEL GINN
COLLEGE OF ENGINEERING

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

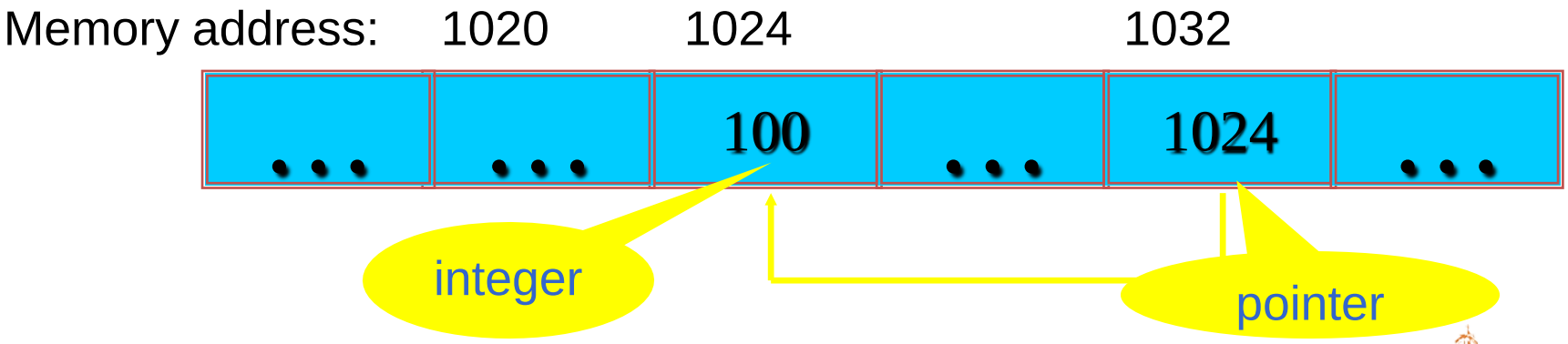



```
int a = 100;
```

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

Pointer Variable

- Declaration of Pointer variables

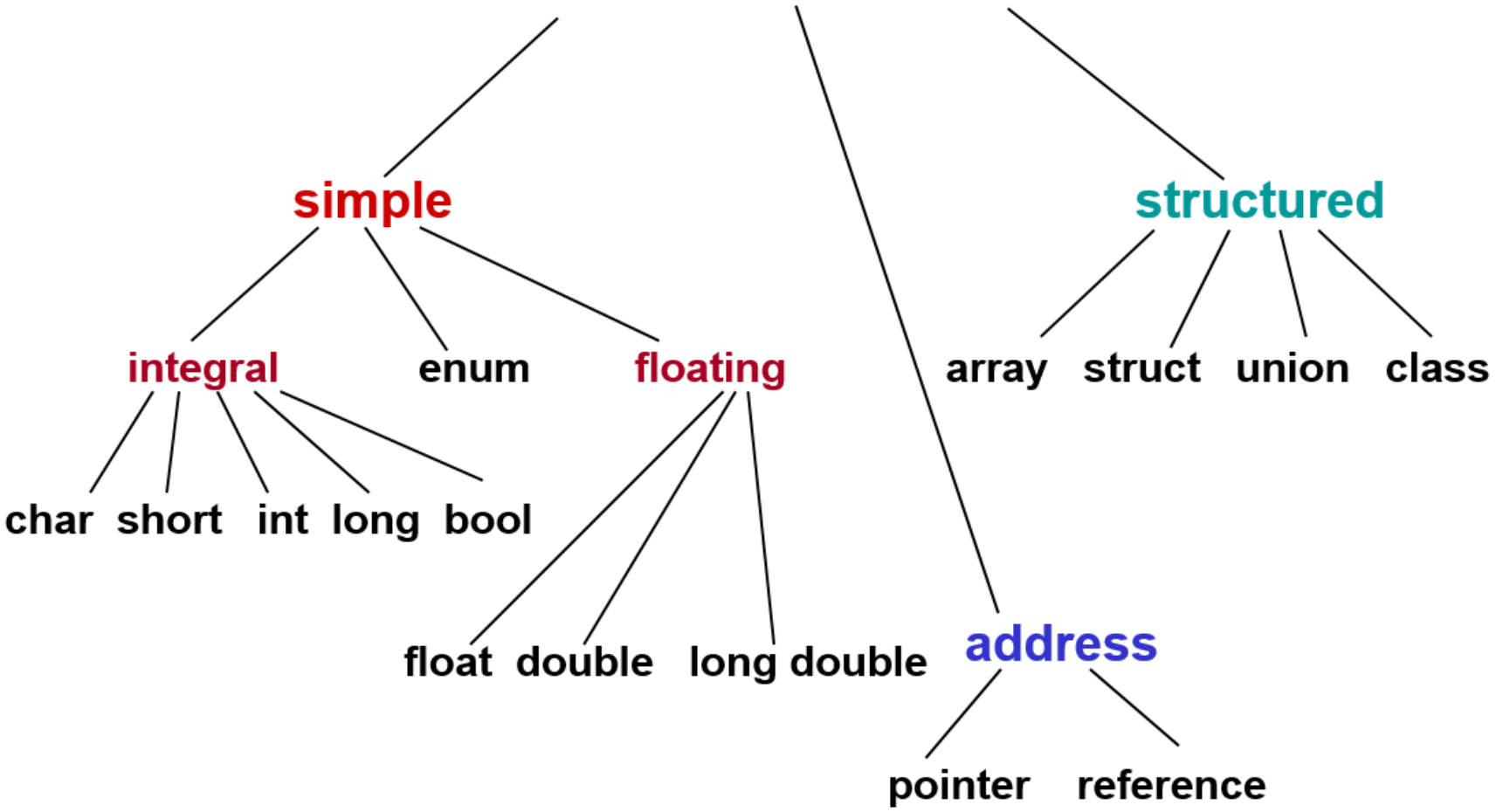
```
type* pointer_name;
//or
type *pointer_name;
```

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

Examples:

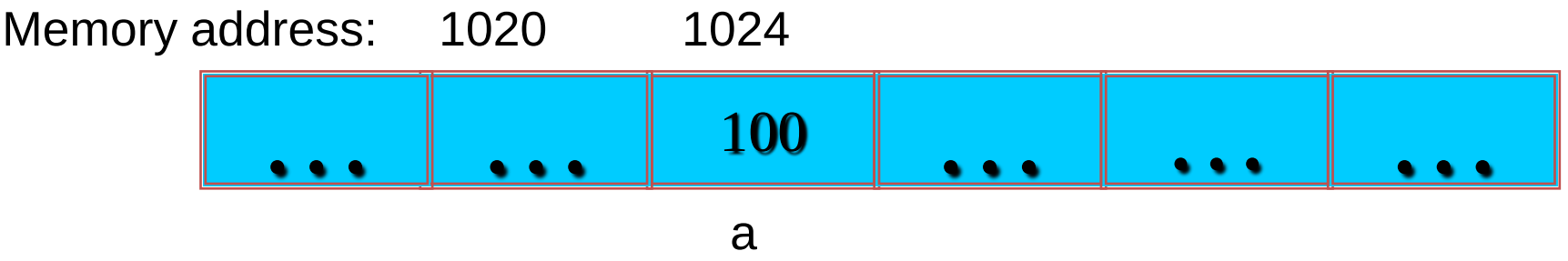
```
int *n;
DayOfYear *r;
int **p;    // pointer to pointer
```

C++ Data Types



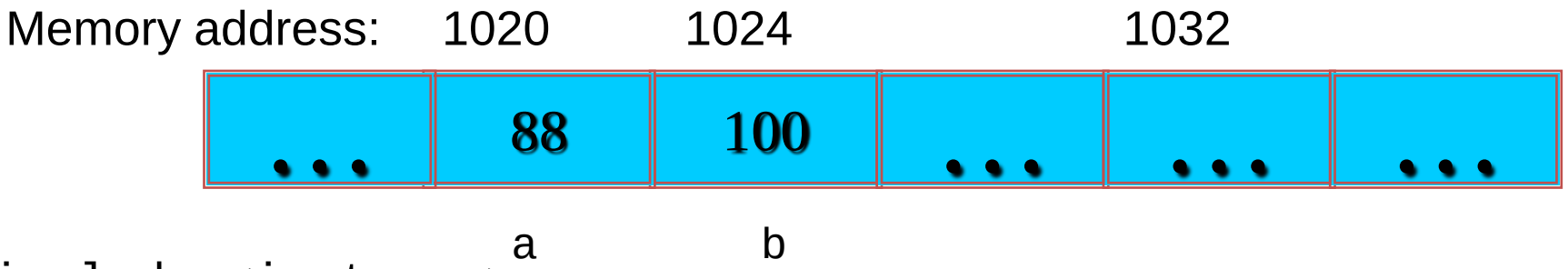
Address Operator &

- *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 &



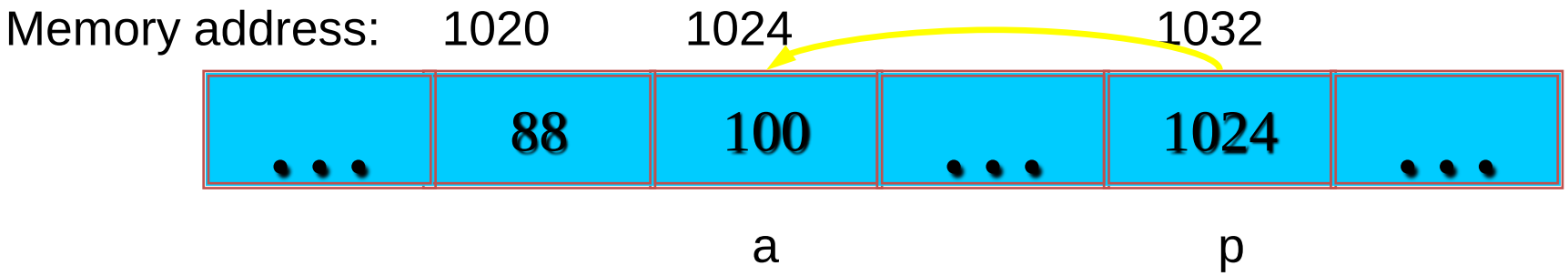

```
#include <iostream>
using namespace std;
void main(){
   int a, b;
   a = 88;
   b = 100;
   cout << "The address of a is: " << &a << endl;
   cout << "The address of b is: " << &b << endl;
}
```

Result is:

The address of a is: 1020

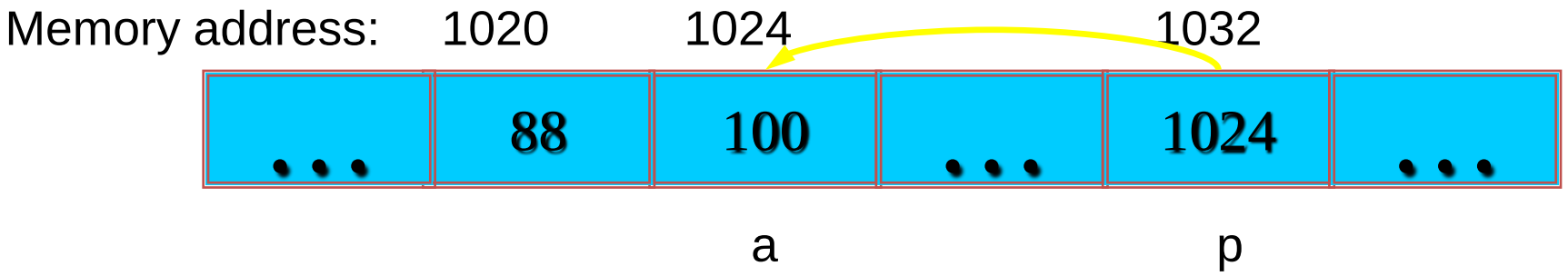
The address of b is: 1024

Pointer Variables




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

Pointer Variables




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

Result is:

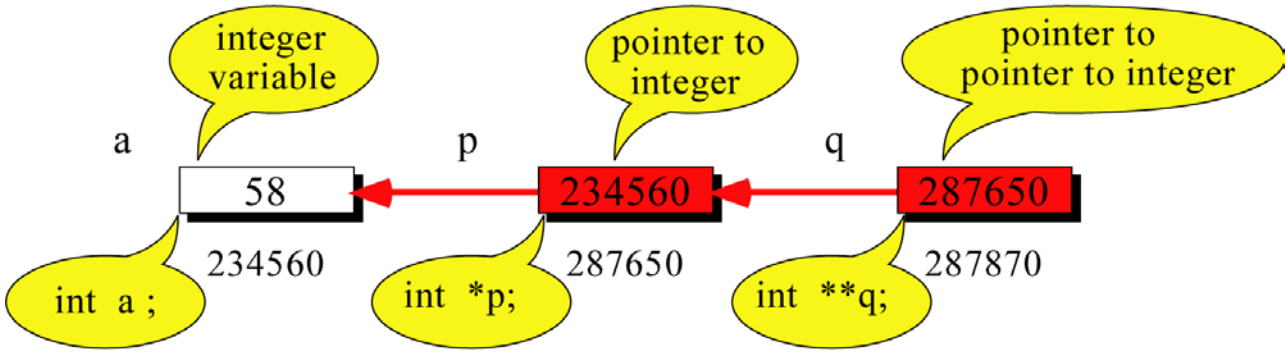
100 1024

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 ;
```



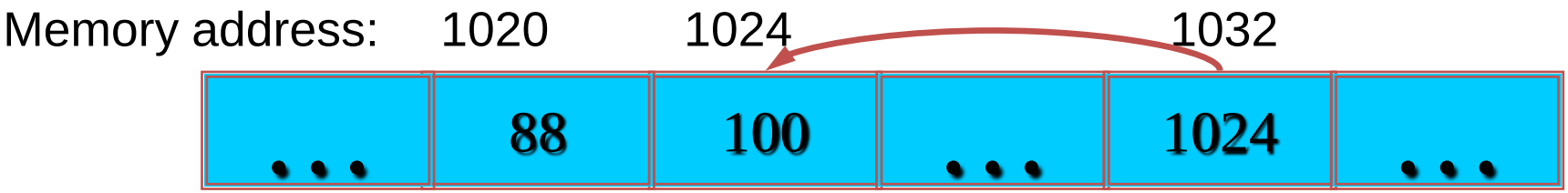
What is the output?

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 do it!

A pointer must have a value before you can *dereference* it (follow the pointer).

```
int *x;
*x=3;
```

ERROR!
Overrode the value located in some address!

```
int foo;
int *x;
x = &foo;
*x=3;
```

this is fine
x points to foo

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;
```

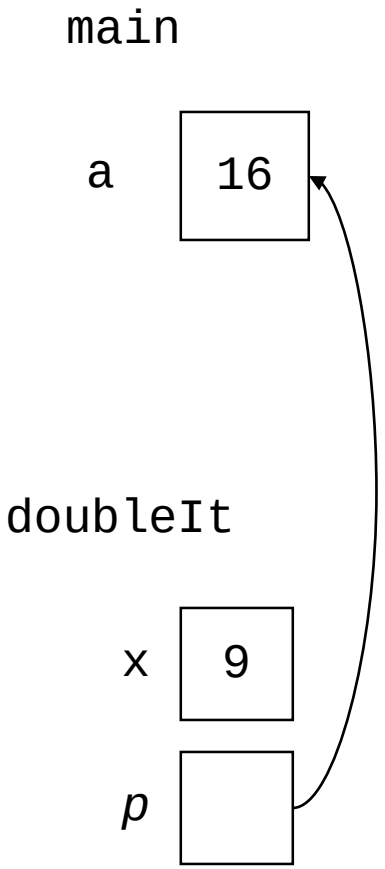
A Pointer Example

The code

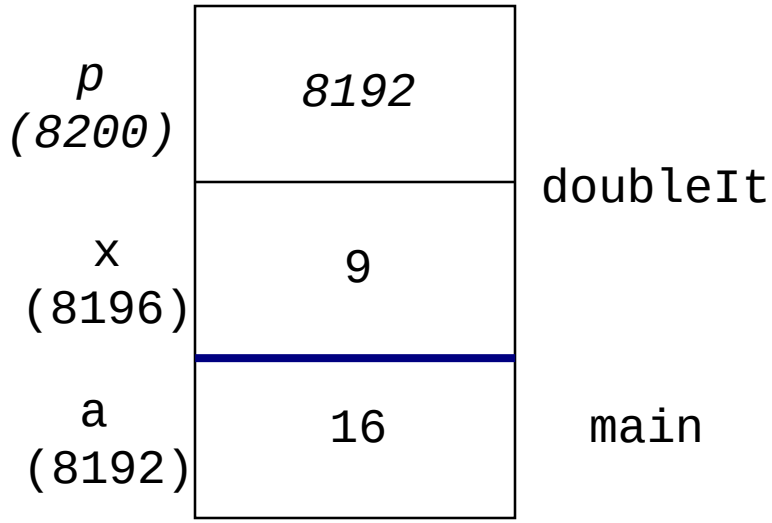
```
void doubleIt(int x,
              int* p)
{
   *p = 2 * x;
}
int main(int argc, const
   char* argv[])
{
   int a = 16;
   doubleIt(9, &a);
   return 0;
}
```

a gets 18

Box diagram



Memory Layout



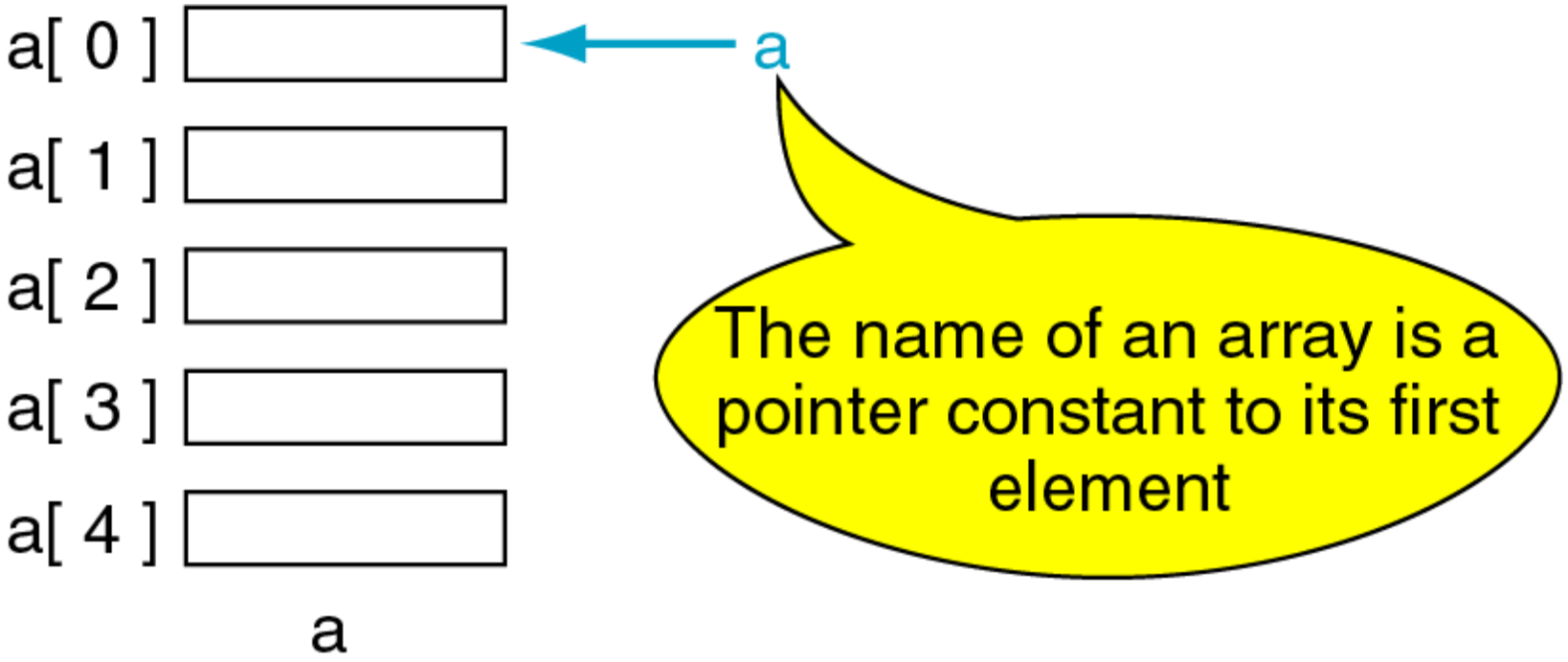
Another Pointer Example

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;
}
```

Pointers and Arrays

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



Array Name is a pointer constant

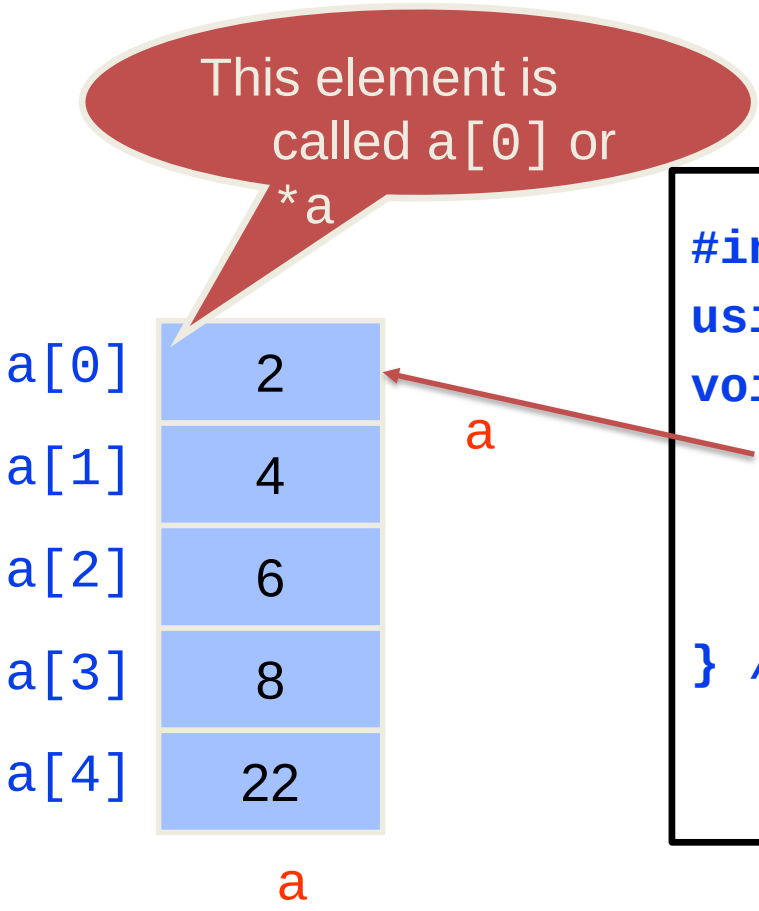
```
#include <iostream>
using namespace std;

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
```

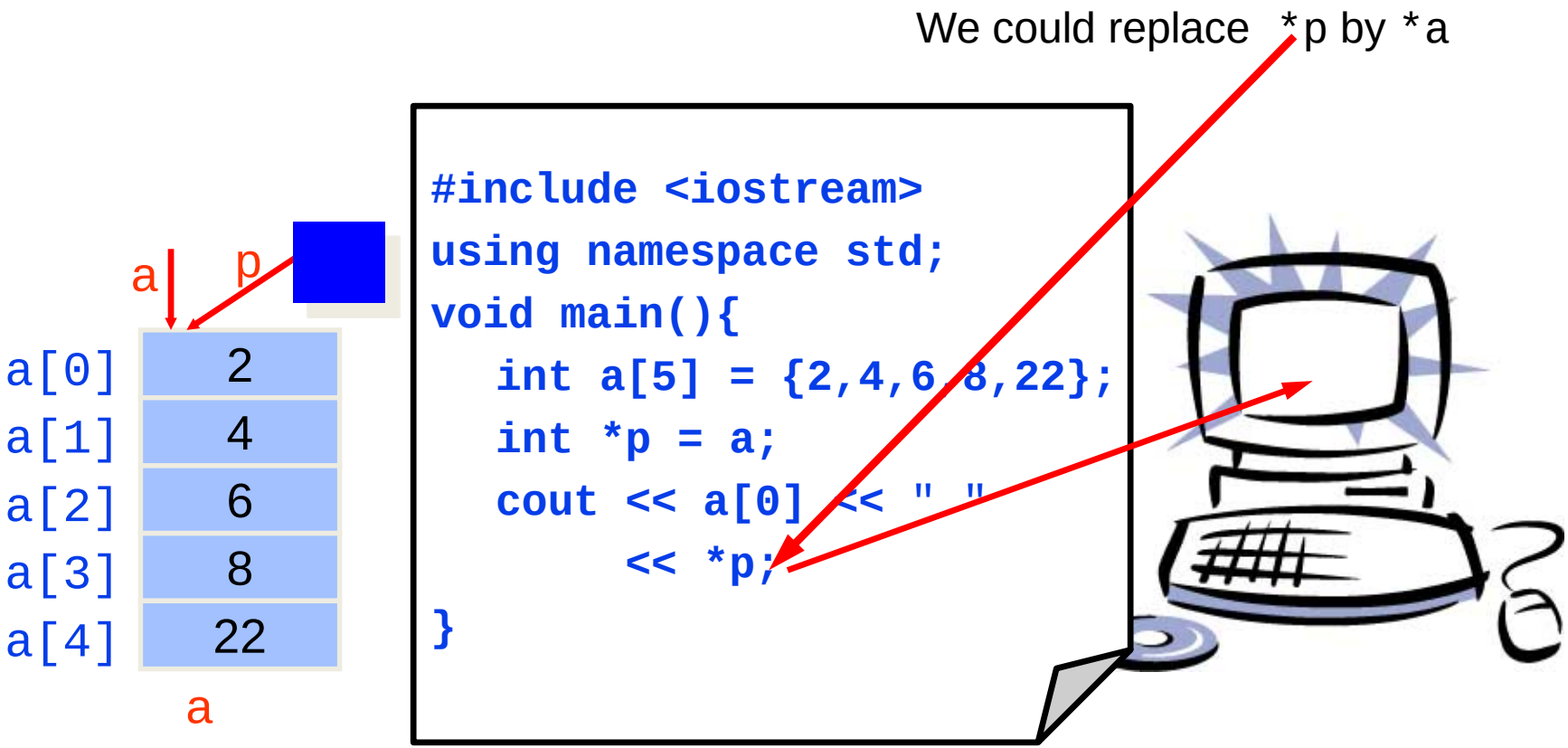
Dereferencing An Array Name




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

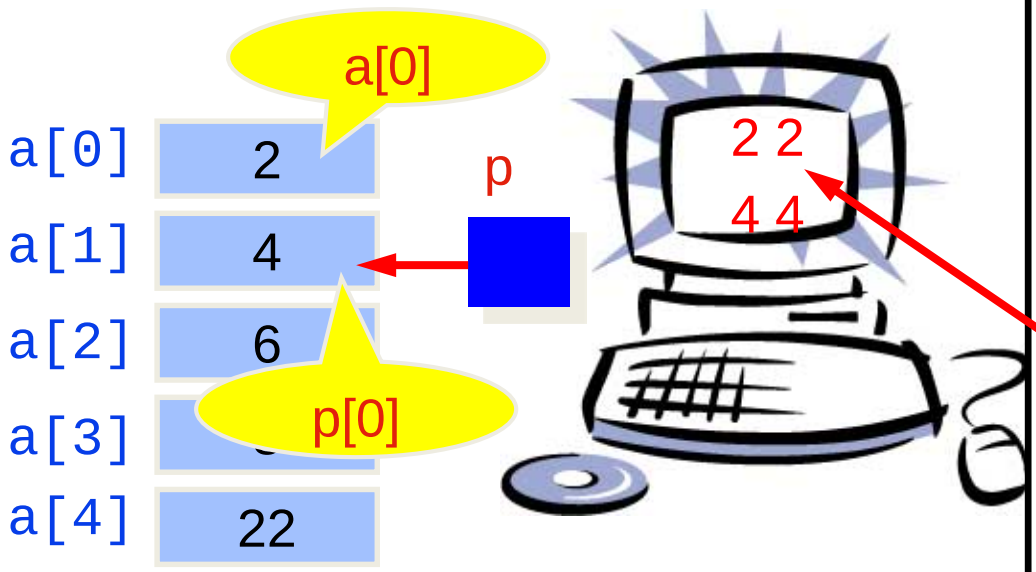
Array Names as Pointers

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



Multiple Array Pointers

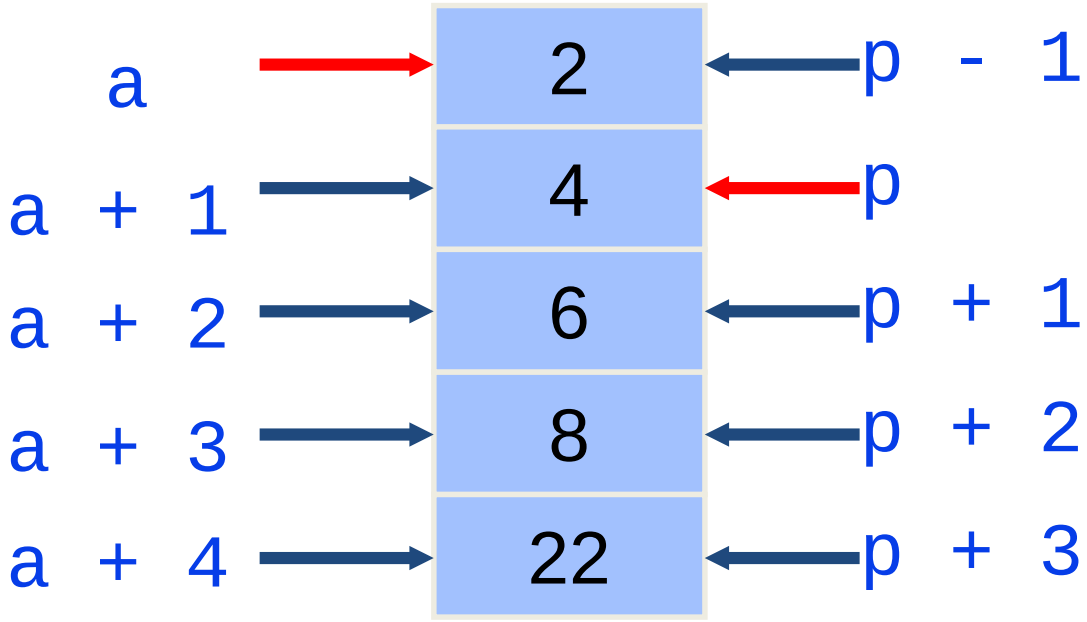
Both a and p are pointers to the same array.




```
#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];
}
```

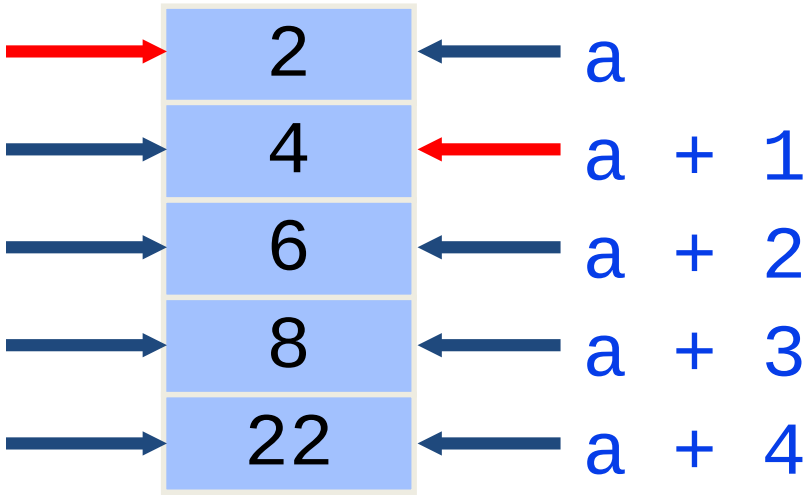
Pointer Arithmetic

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



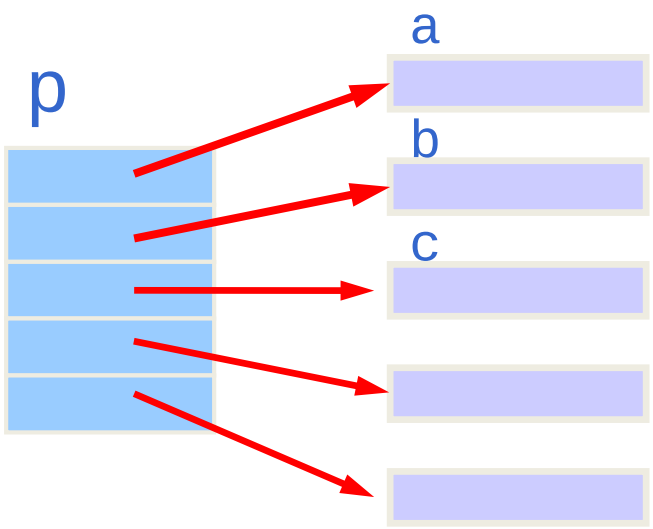
Dereferencing Array Pointers

```
a[0] or *(a + 0)
a[1] or *(a + 1)
a[2] or *(a + 2)
a[3] or *(a + 3)
a[4] or *(a + 4)
```



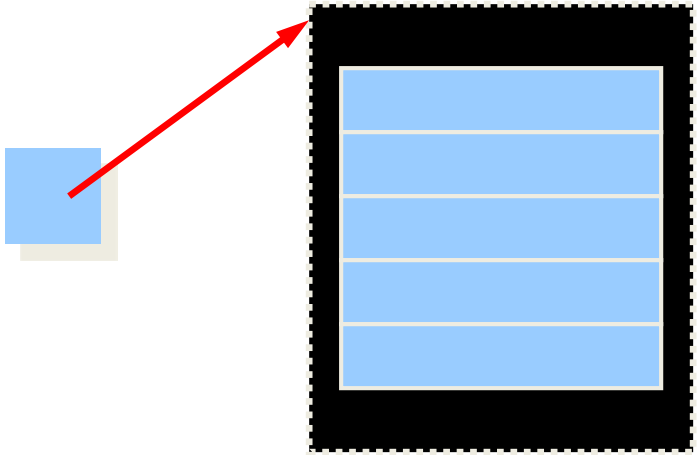
`*(a+n)` is identical to `a[n]`

Array of Pointer & Pointer of 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

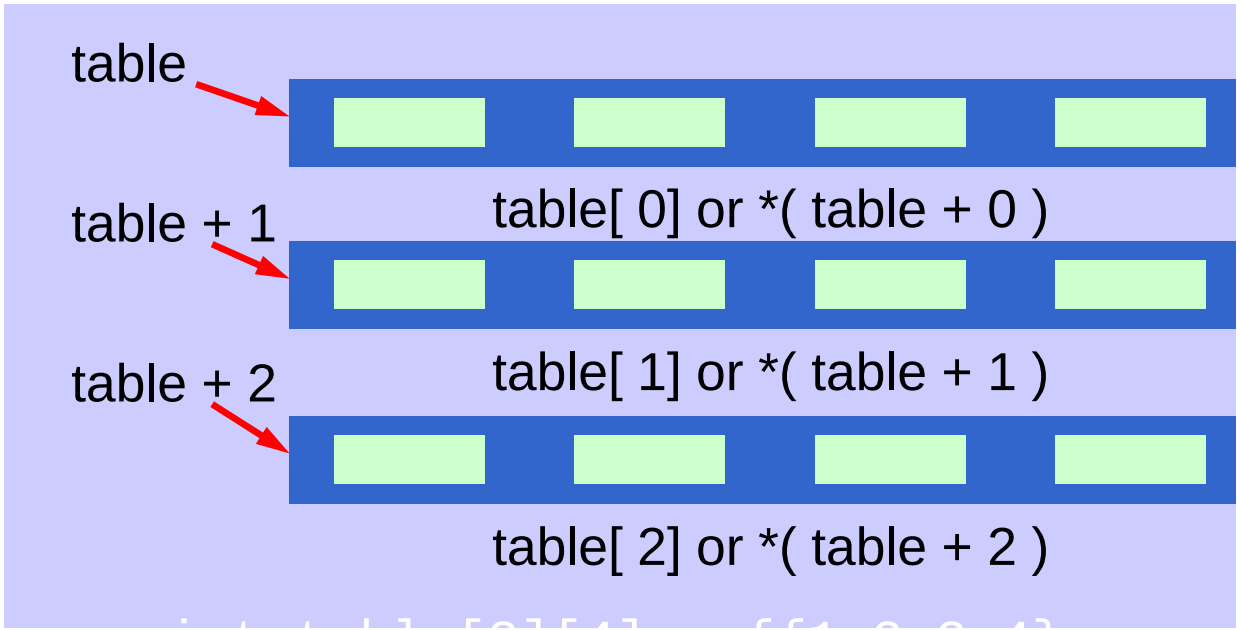
```
int list[5] = {9, 8, 7, 6, 5};
int *p;
P = list;//points to 1st entry
P = &list[0];//points to 1st entry
P = &list[1];//points to 2nd entry
P = list + 1; //points to 2nd entry
```

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

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

What is `**table` ?

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

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!
```

Java vs. C++

Java's References

- String s = new String("a string");
  - The value of the variable 's' is the location of the object on the heap, but **we don't have access to this value**!
- You can not have a variable that 'points' to a primitive type
- String s = **null**;
  - means "I don't have the address of any meaningful data"

C++ Pointers

- Have **access to actual memory locations**.
- Can point to anything!
- The **null** value in C++ is the number 0 (memory address 0).

Java vs. C++

Java's References

- In Java, you can't directly access the memory of such object! Only by reference.

C++

- A pointer is a variable that holds the address of some other variable.
- In C++, you can access them directly! Alter them any way you like, without any limitations!
  - ✓ Benefit: More control than Java.
  - ✓ Drawback: Memory leaks, Memory corruption.
- A 64-bit computer can address 264 bytes!
- Each memory cell is 1 byte. (byte-accessible machines – most of the machines today)
- Each pointer takes a static size of 4 bytes in a 32bit OS, and 8bytes in a 64bit OS.
- Pointers themselves are saved on the stack.