BIOPHOTONICS & BIOIMAGING LABORATORY

# Week 5 Pointers

陳立達 @ 2017 Summer

NTUBIME Lab405

# Preface

- Pointers are simply variables used to store memory addresses.

- To review the mechanism of pointers, and have some examples showing where these stored addresses can be used.

# Outline

- Addresses and Pointers
- Pointer Arithmetic
- Access Array using Pointers
- Passing Addresses

# Addresses and Pointers

# Addresses and Pointers

- Address operator &

- Putting the address operator " & " in front of a variable's name refers to the address of the variable.

- EX:

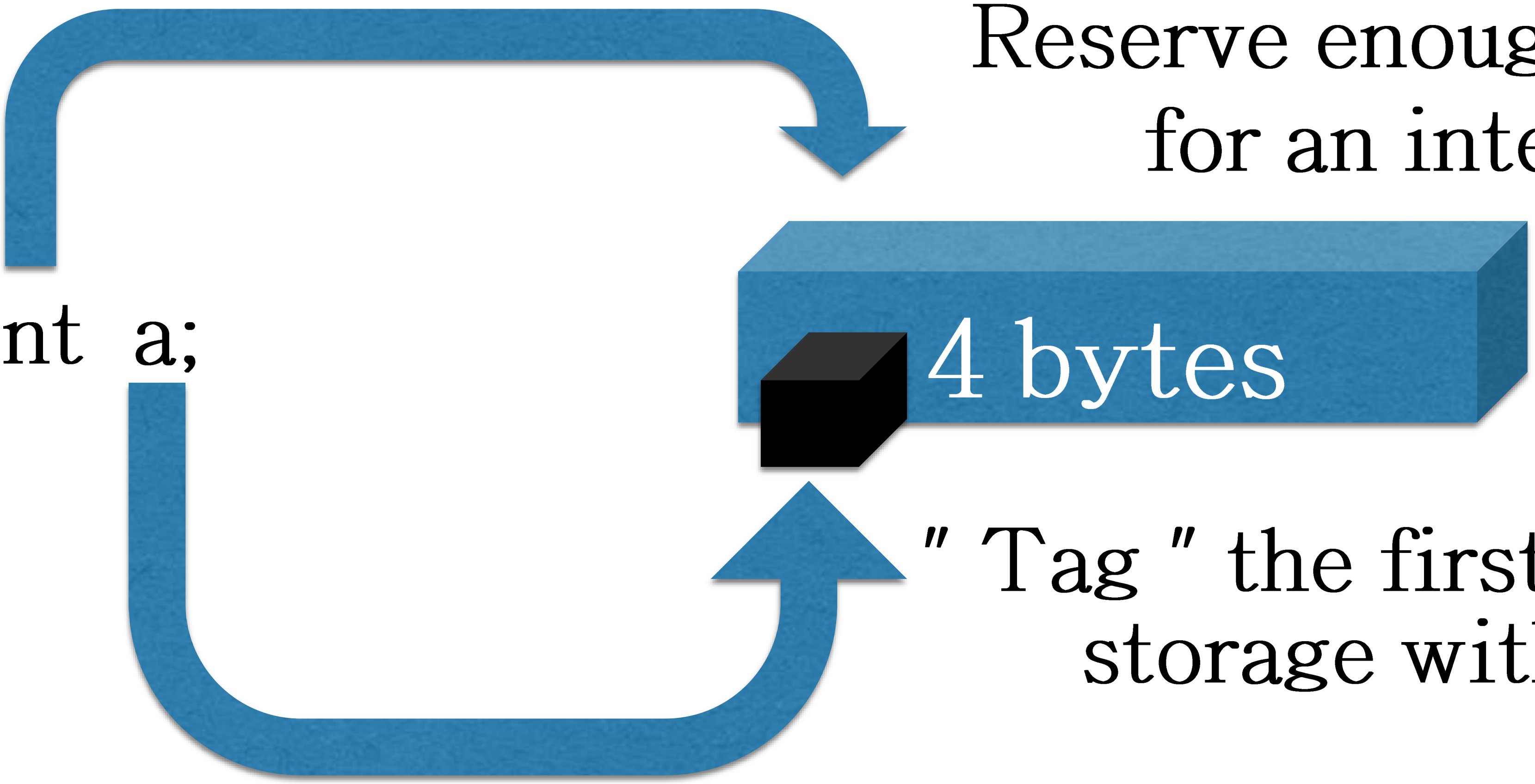
```
int a = 0;
cout << &a << endl;
```

Output: 012FFC90

Address of the first byte used by **a**

# Addresses and Pointers

- Memory Allocation



Tells the computer to

Reserve enough storage space
for an integer number

int  a;

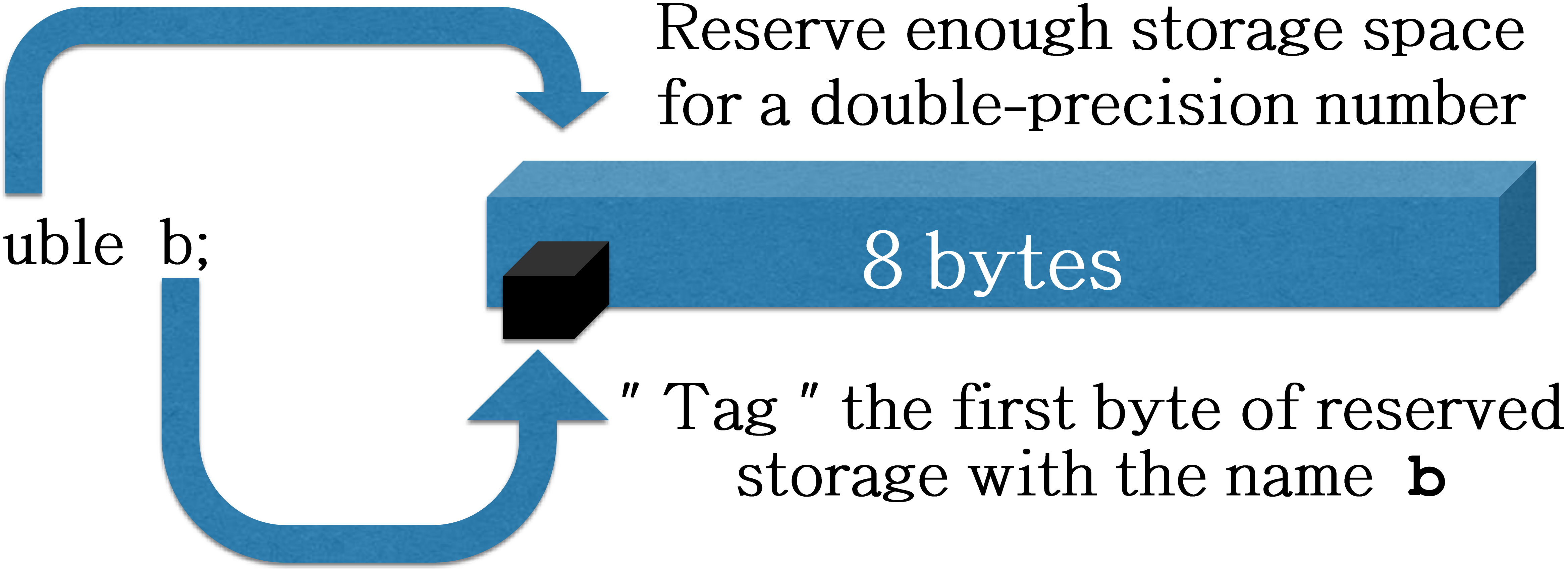
4 bytes

" Tag " the first byte of reserved
storage with the name **a**

Tells the computer to

# Addresses and Pointers

- Memory Allocation



Tells the computer to

Reserve enough storage space
for a double-precision number

double b;

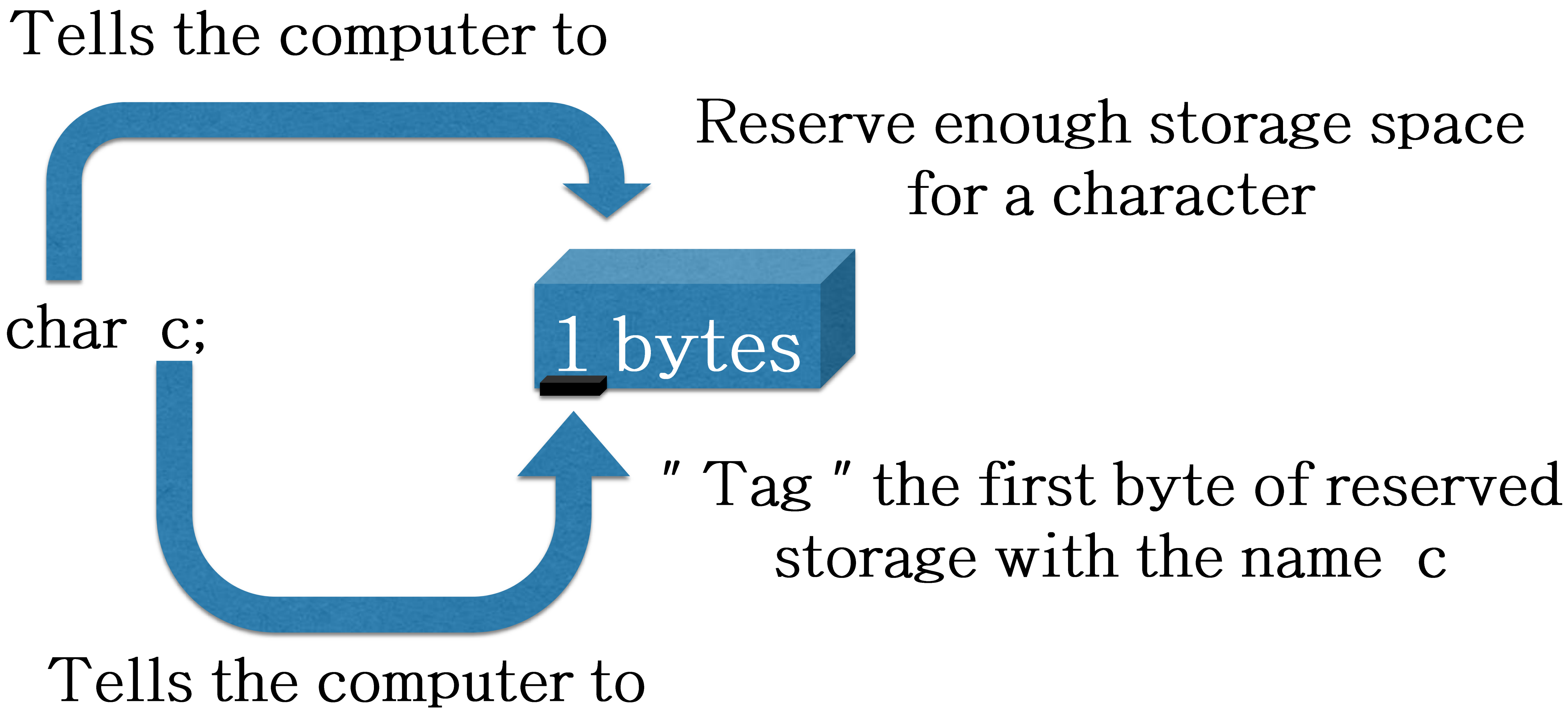
8 bytes

" Tag " the first byte of reserved
storage with the name **b**

Tells the computer to

# Addresses and Pointers

- Memory Allocation



Tells the computer to

Reserve enough storage space for a character

char c;

1 bytes

" Tag " the first byte of reserved storage with the name c

Tells the computer to

# Addresses and Pointers

- Memory Allocation

- The compiler uses the variable name to locate the first byte of storage previously allocated to the variable.

- Knowing the variable’s data type then allows the compiler to store or retrieve the correct number of bytes.

# Addresses and Pointers

* Storing Addresses

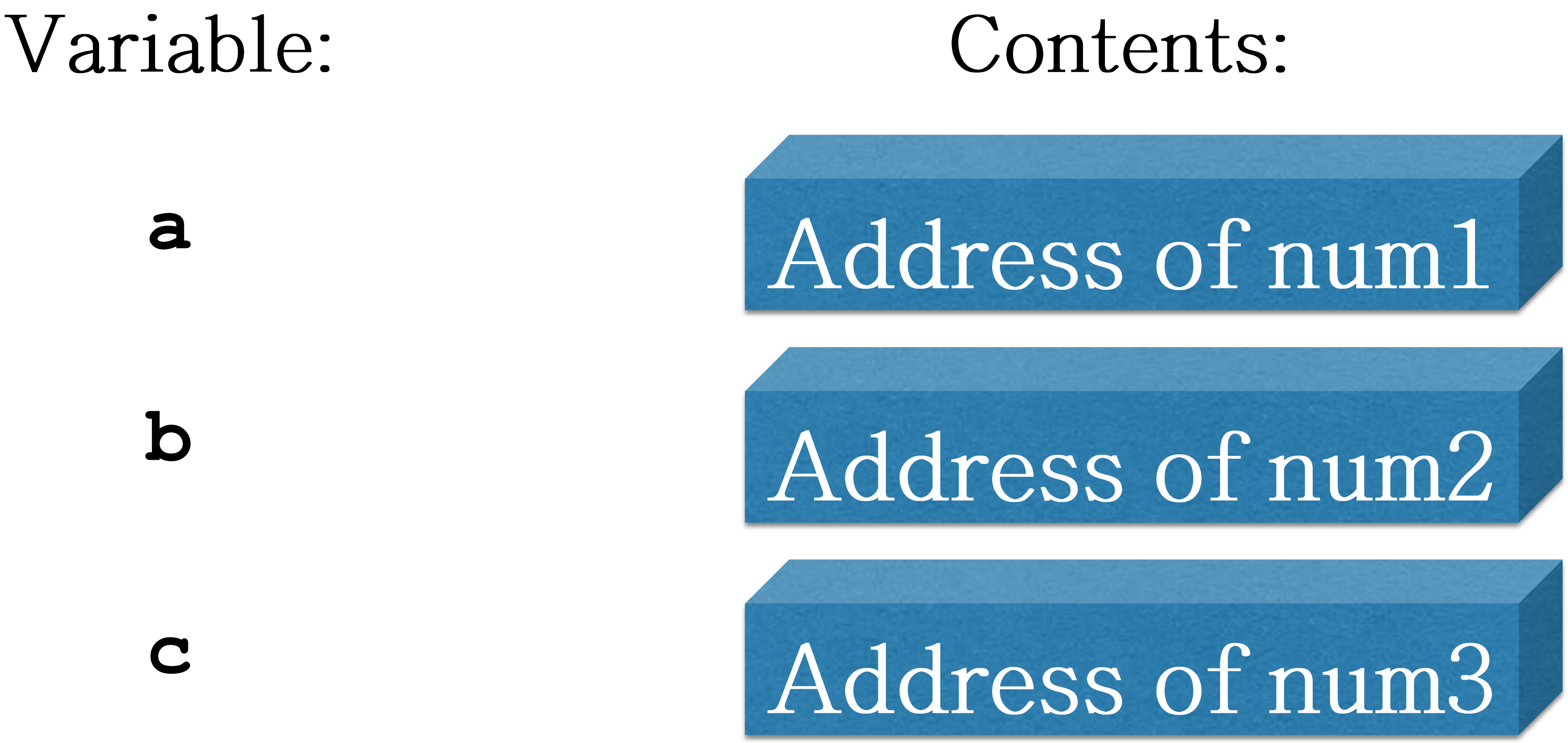
- The following statement stores the address to the corresponding variable.

- EX:

```
a = &num1;
b = &num2;
c = &num3;
```

# Addresses and Pointers

- Storing Addresses



| Variable: | Contents: |
|---|---|
| **`a`** | Address of num1 |
| **`b`** | Address of num2 |
| **`c`** | Address of num3 |

Because variables **`a`**, **`b`**, **`c`** are used to store address, so they are called **pointer variables** or **pointers.**

# Addresses and Pointers

- Declaring Pointers

*dataType* pointerName;*

```
EX:  int *a;
     double *b;
     char *c;
```

# Addresses and Pointers

- Declaring Pointers

Pointers could also be declared in this form:

```
int* a;

double* b;

char* c;
```

With a space between the indirection operator * and the pointer name

However, this form might not act as you want when multiple pointers are declared in the same statement.

# Addresses and Pointers

- Declaring Pointers

EX:

```
int* num1, num2;
```

This statement declares num1 as a pointer, and num2 as an integer variable.

# Addresses and Pointers

- Declaring Pointers

EX:

```
int *num1, *num2;
```

This statement declares both num1 and num2 as pointer.

# Addresses and Pointers

- Pointer Initialization

EX:

```
int *a = &num1;
```

This statement is valid if and only if

1. num1 is also an integer variable

2. num1 has been declared before pointer a

# Addresses and Pointers

- Indirection operator *

- When used in a non-declaration statement, the indirection operator * is used to access the value of the variable whose address is stored in the corresponding pointer.

Example 1

# Addresses and Pointers

- Indirection operator *

In brief, if **`numAddr`** is a pointer, ***`numAddr`** means

“the variable whose address is stored in **`numAddr`** ”

# Addresses and Pointers

- References Variables

- As a variable has been declared, it can be given additional names by using a reference declaration which has this form:

*datatype& newName = existingName ;*

# Addresses and Pointers

- References Variables

- EX:

```
double& sum = total;
```

- **sum** is an alternative name for **total** now

Example 2

# Addresses and Pointers

- References Variables

- Reference Variables could also be declared in this form:

*datatype &newName = existingName ;*

With a space between the ampersand and the data type, however, this form isn’t used much

# Addresses and Pointers

- References Variables

- Some restrictions of Reference Variables

1. The reference should be of the same data type as the variables it refers to

EX:

```
int num = 5;
double& numref = num;  // INVALID
```

# Addresses and Pointers

- References Variables

- Some restrictions of Reference Variables

2. Another compiler error is produced when an attempt is made to equate a reference to a constant

EX:

```
int& num = 5;  // INVALID
```

# Addresses and Pointers

- References Variables

- Some restrictions of Reference Variables

3. The reference variable can't be altered to refer to any variable except the one it's initialized to

EX:

```
double& numref = num1;
        numref = num2; // INVALID
```

# Addresses and Pointers

- Difference between References and Pointers

For Reference

```
int b;            //  b is an integer variable
int& a = b;   // a is a reference variable that stores b’s address
a = 10;          // this changes b’s value to 10
```

For Pointers

```
int b;               //  b is an integer variable
int *a = &b;   // a is a pointer - store b’s address in a
*a = 10;           // this changes b’s value to 10  by explicit dereference
                        of the address in a
```

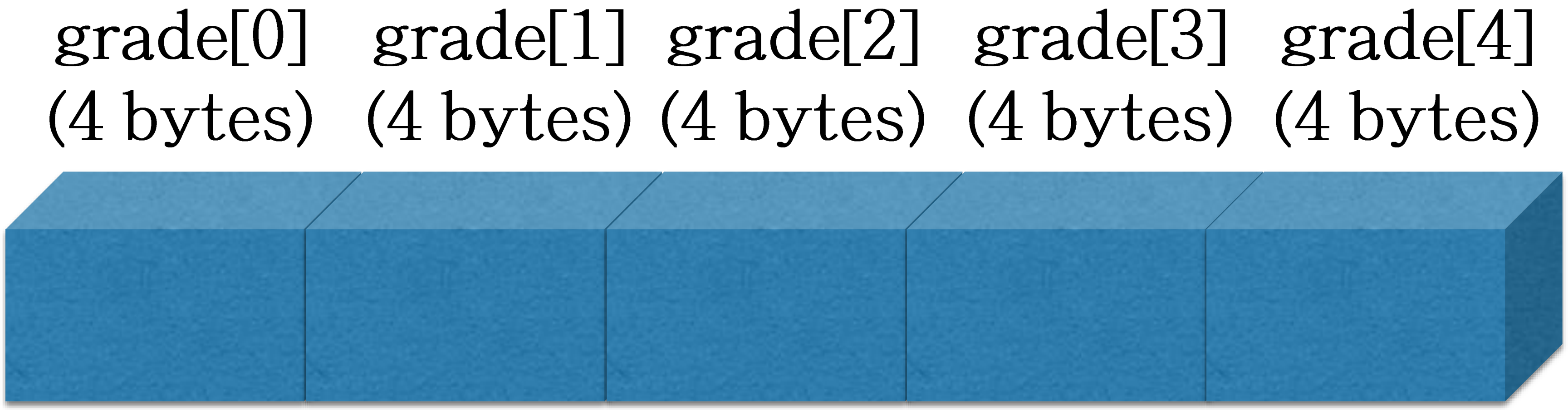
# Pointer Arithmetic

# Pointer Arithmetic

- Array name and Pointers

EX:

```
int grade[5] = {0};
```



grade[0] (4 bytes)
grade[1] (4 bytes)
grade[2] (4 bytes)
grade[3] (4 bytes)
grade[4] (4 bytes)

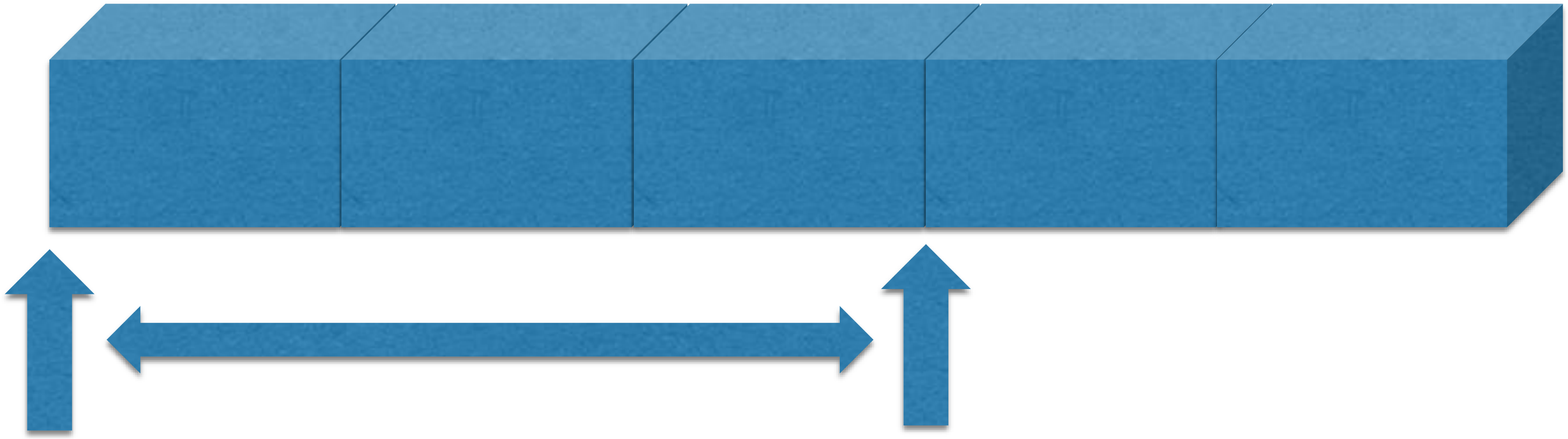
# Pointer Arithmetic

- Address computation

EX:

&grade[3] = &grade[0] + ( 3 * sizeof(int) )

grade[0] (4 bytes) grade[1] (4 bytes) grade[2] (4 bytes) grade[3] (4 bytes) grade[4] (4 bytes)



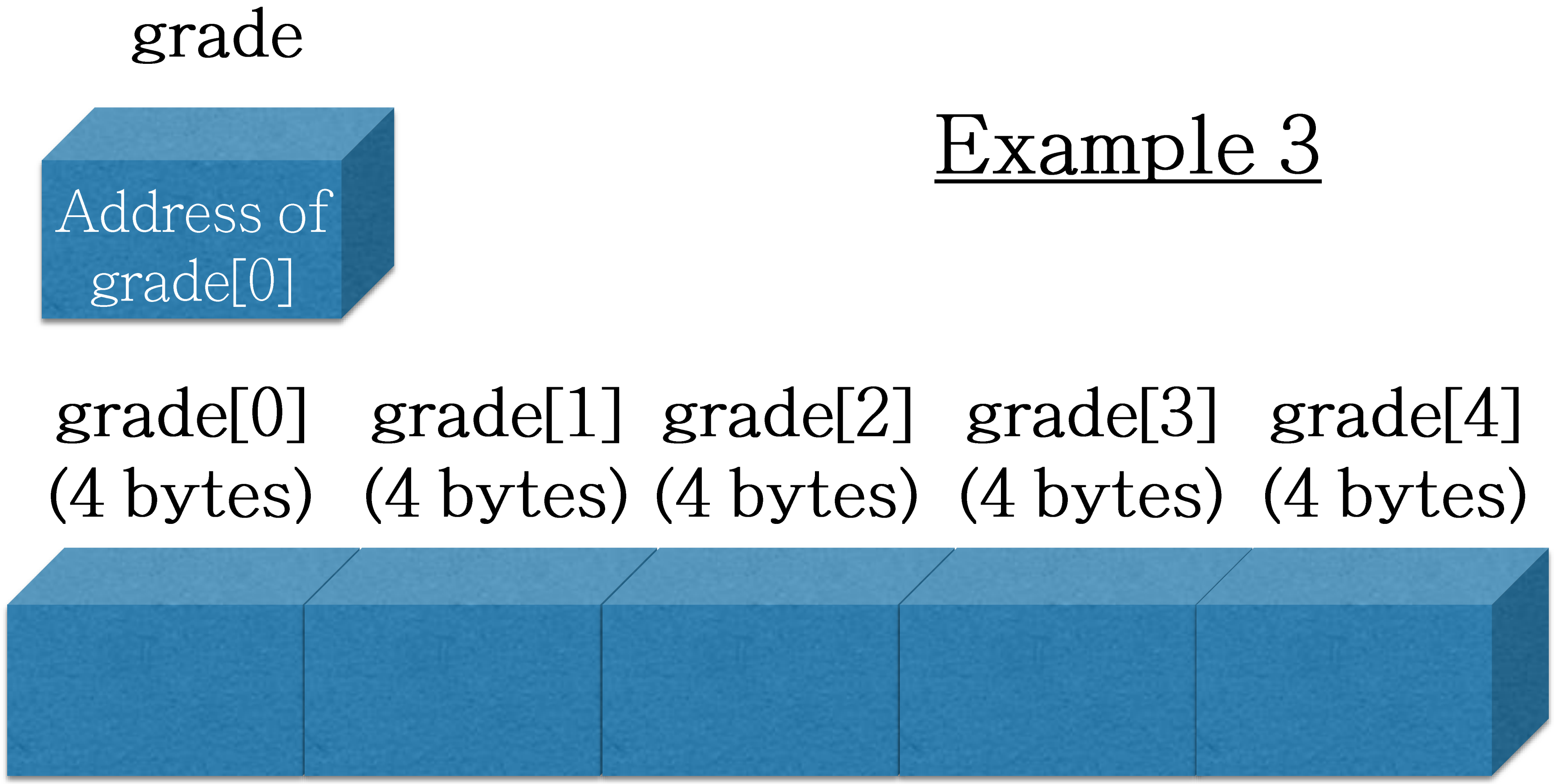
3 jump

# Pointer Arithmetic

- Array name and Pointers

Array name is actually a pointer constant

Example 3



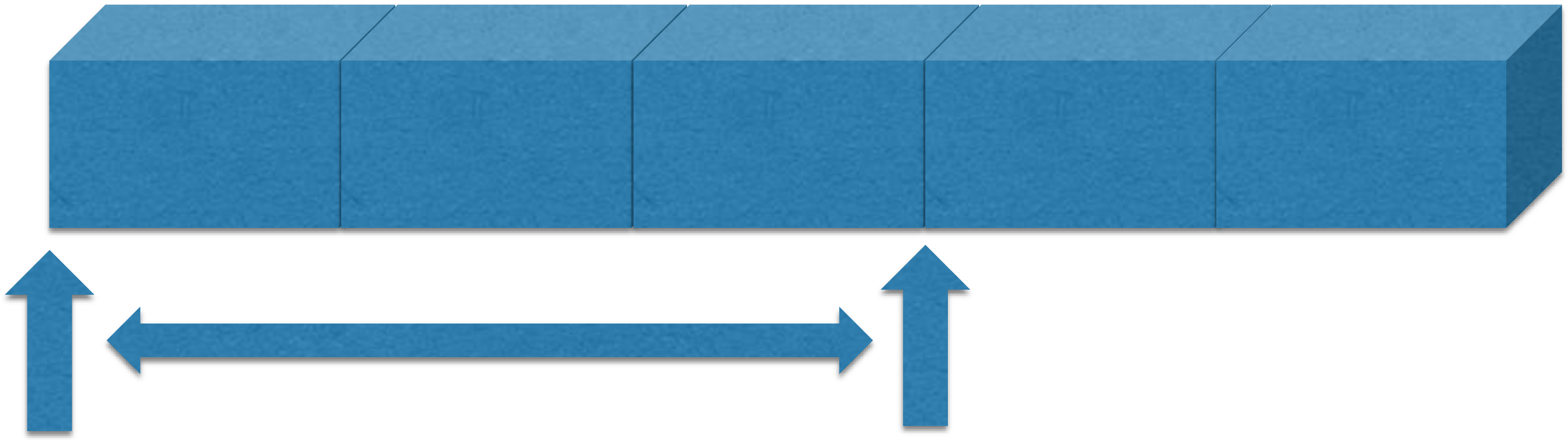
# Pointer Arithmetic

- Address computation could be simplified

EX:

&grade[3] = grade + 3

grade[0] (4 bytes) grade[1] (4 bytes) grade[2] (4 bytes) grade[3] (4 bytes) grade[4] (4 bytes)



3 jump

# Access Array using Pointers

# Access Array using Pointers

- Access Array by standard subscript notation

grade[i]

- Access Array by Pointer notation

*(grade + i)

Example 5

# Access Array using Pointers

- Dynamic Array Allocation

To allocate the amount of storage at run time rather than compile time

Require #include<new>

| Operator Name | Description |
|---|---|
| new | Reserves the number of bytes requested by the declaration. Returns the address of the first reserved location or NULL if not enough memory is available. |
| delete | Releases a block of bytes reserved previously. The address of the first reserved location must be passed as an argument to the operator. |

# Access Array using Pointers

- Dynamic Allocate Variable

EX:

```
int *num = new int;
```

or

```
int *num;
num = new int;
```

# Access Array using Pointers

- Dynamic Allocate Arrays

EX:

```
int *grades = new int[200];
```

Example 6

# Access Array using Pointers

- Dynamic Allocate 2D-Arrays

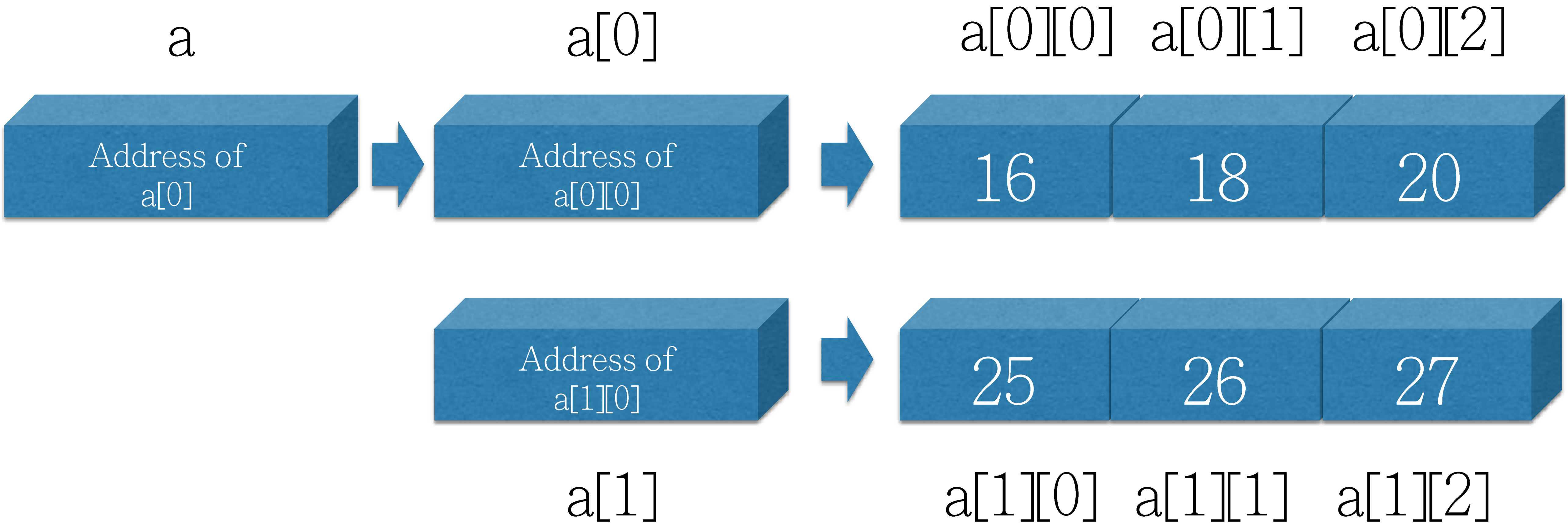
Pre-knowledge:

```
int  a[2][3] = {  {16, 18, 20}
                  {25, 26, 27}  };
```

# Access Array using Pointers

- Dynamic Allocate 2D-Arrays

Pre-knowledge:



# Access Array using Pointers

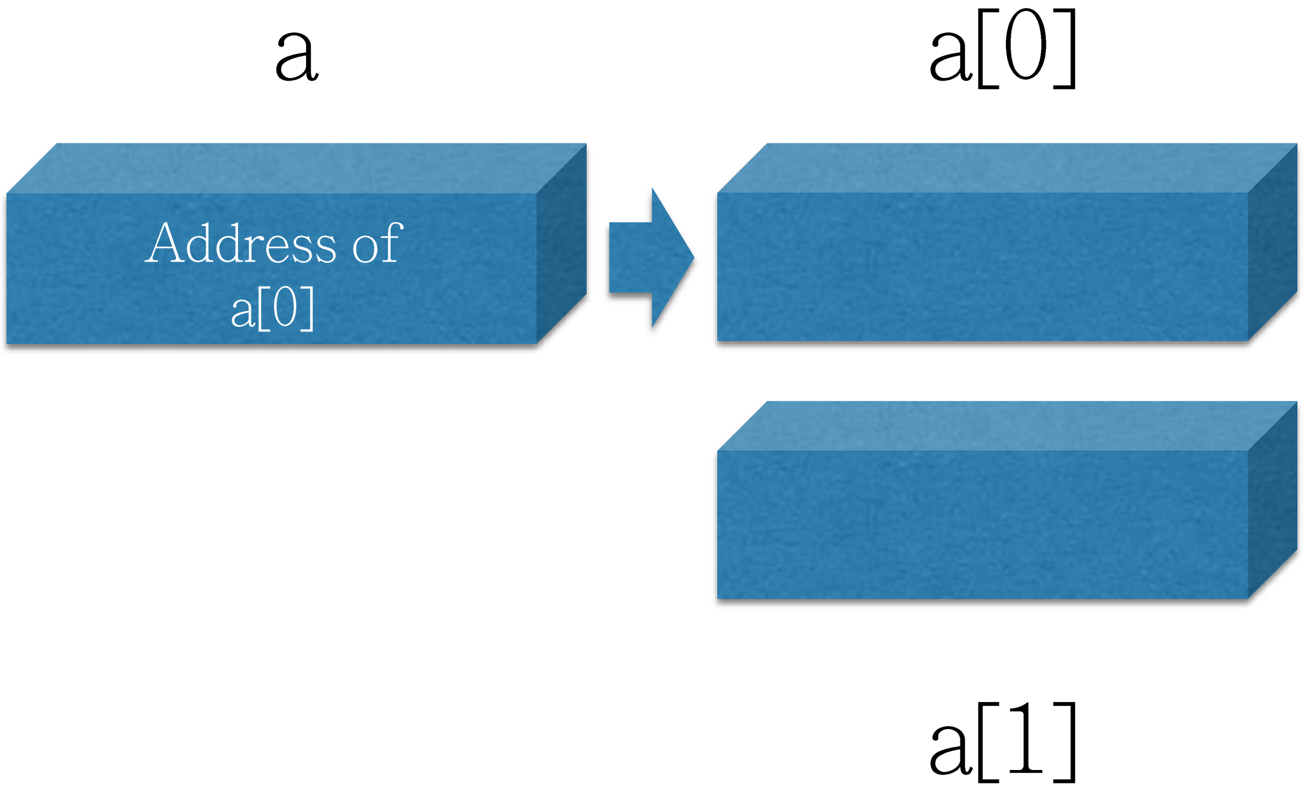
- Dynamic Allocate 2D-Arrays

<u>Example 7</u>

# Access Array using Pointers

- Dynamic Allocate 2D-Arrays

```
int** a = new int*[2];
```



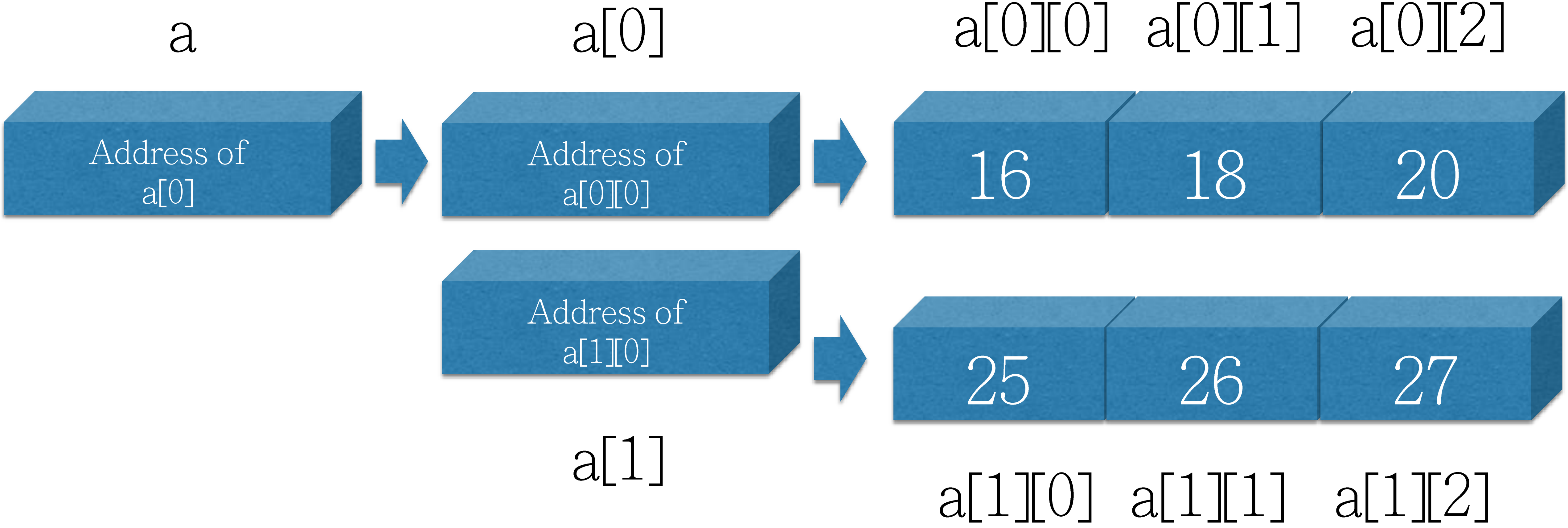
# Access Array using Pointers

- Dynamic Allocate 2D-Arrays

```
int** a = new int*[2];

for (int i = 0; i < 2; ++i)

a[i] = new int[3];
```



a

Address of a[0]

a[0]

Address of a[0][0]

a[1]

Address of a[1][0]

a[0][0] a[0][1] a[0][2]

16 18 20

25 26 27

a[1][0] a[1][1] a[1][2]

# Passing Addresses

# Passing Addresses

- Pass by reference

- Explicity passing addresses with the address operator &

- EX:

```
swap(&firstnum, &secnum);
```

Example 8

# Passing Addresses

- Passing Arrays

- When an array is passed to a function, it’s address is the only item actually passed

Example 9

Thank you ~~

BIOPHOTONICS & BIOIMAGING LABORATORY