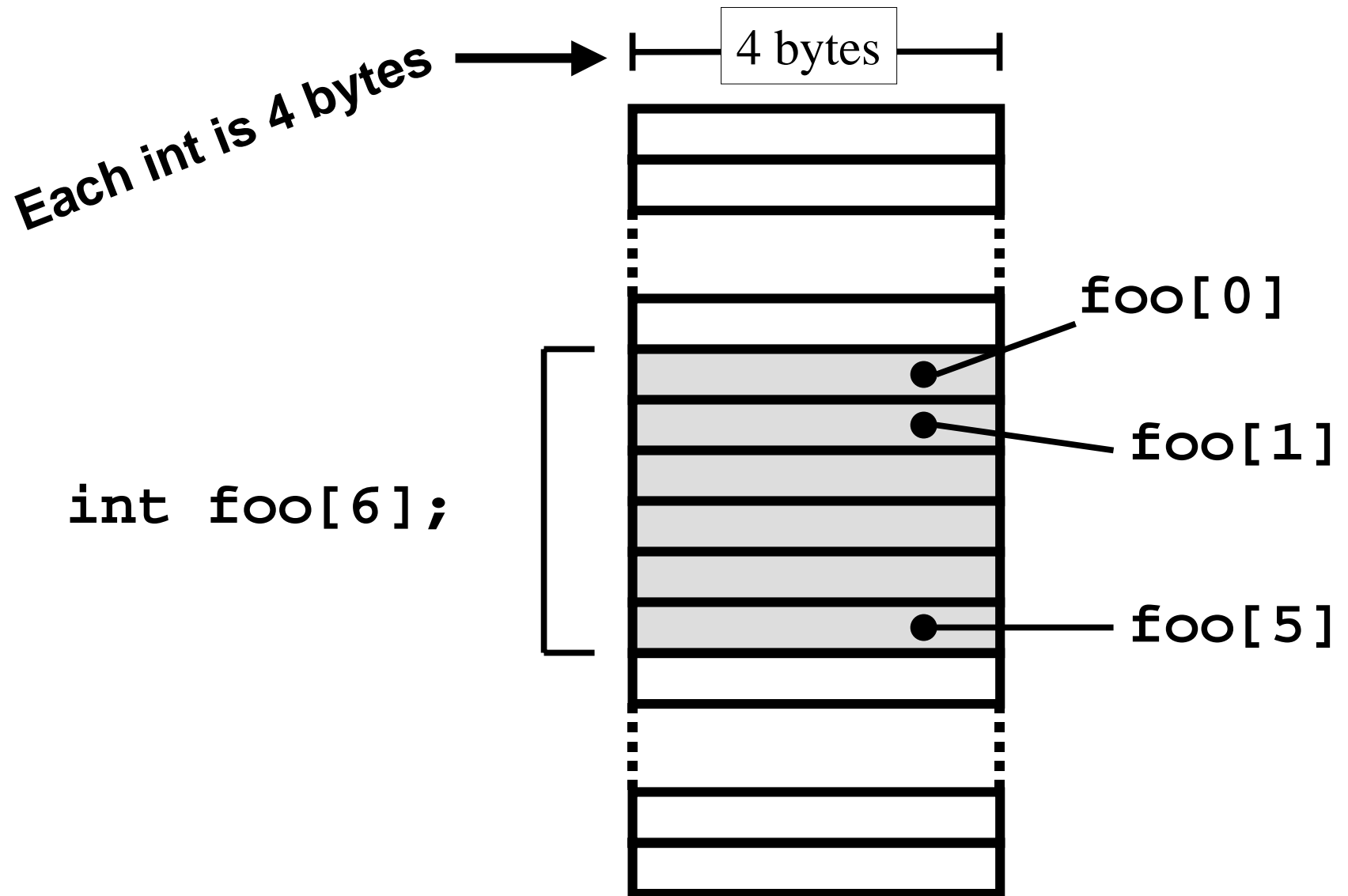


# C++ Arrays

# C++ Arrays

- An array is a consecutive group of memory locations.
- Each *group* is called an element of the array.
- The contents of each element are of the same *type*.
  - Could be an array of int, double, char, ...
- We can refer to individual *elements* by giving the position number (index) of the element in the array.

# Memory and Arrays



# C++ Arrays start at 0 !!!!!!!

- The first element is the 0<sup>th</sup> element!
- If you declare an array of  $n$  elements, the last one is number  $n-1$ .
- If you try to access element number  $n$  it is an error!

If only millenniums started at 0 ...

# Array Subscripts

- The element numbers are called subscripts.

*Array name* → **foo[i]** ← *subscript*

A subscript can be any integer expression:

These are all valid subscripts:

**foo[17]**    **foo[i+3]**    **foo[a+b+c]**

# Array Example

```
int main(void) {
    int facts[10];

    for (int i=0;i<10;i++)
        facts[i] = factorial(i);

    for (int i=0;i<10;i++)
        cout << "factorial(" << i << ") is " <<
            facts[i] << endl;
}
```

# Declaring An Array

```
element_type array_name[number_of_elements];
```

`element_type` can be any C++ variable type.

`array_name` can be any valid variable name.

`number_of_elements` can be an expression.

# Initialization

- You can initialize an array when you declare it (just like with variables):

```
int foo[5] = { 1, 8, 3, 6, 12};
```

```
double d[2] = { 0.707, 0.707};
```

```
char s[] = { 'R', 'P', 'I' };
```



**You don't need to specify a size when initializing, the compiler will count for you.**

# An array printing function

**Can pass an array as a parameter.  
You don't have to say how big it is!**



```
void print_array(int a[], int len) {  
    for (int i=0;i<len;i++)  
        cout << "[" << i << "] = "  
            << a[i] << endl;  
}
```

# What if we want to print doubles?

- The `print_array` function is declared to handle only `ints`.
- We can write another function that can be used to print doubles.
- We have to write another function (we can't use the same one).
  - Not really true – this is what templates can do for you!

## `print_array()` for doubles

```
void print_array(double a[], int len)
{
    for (int i=0;i<len;i++)
        cout << "[" << i << "] = "
            << a[i] << endl;
}
```

# Which is it?

- We now have two functions with the same name:

```
void print_array(double a[], int len);  
void print_array(int a[], int len);
```

This is fine – as long as the prototypes are different everything works.

This is called "overloading", using the same name for two (or more) different functions.

# Arrays of char are special

- C++ provides a special way to deal with arrays of characters:

```
char string1[] =  
    "RPI without PI is like meat without eat";
```

- char arrays can be initialized with string literals.

# Arrays of Arrays

- You can create an array of arrays:

```
int a[2][2];
```

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

```
    for (int j=0;j<2;j++)
```

```
        a[i][j] = i+j;
```

## 2-D Array: `int A[3][4]`

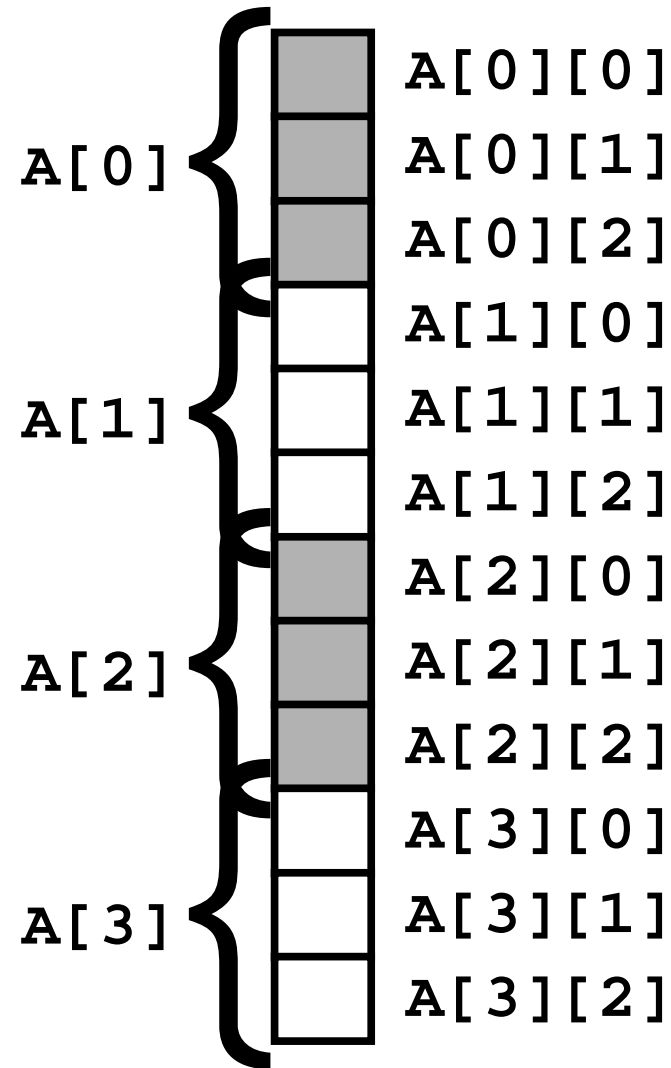
	Col 0	Col 1	Col 2	Col 3
Row 0	<code>A[0][0]</code>	<code>A[0][1]</code>	<code>A[0][2]</code>	<code>A[0][3]</code>
Row 1	<code>A[1][0]</code>	<code>A[1][1]</code>	<code>A[1][2]</code>	<code>A[1][3]</code>
Row 2	<code>A[2][0]</code>	<code>A[2][1]</code>	<code>A[2][2]</code>	<code>A[2][3]</code>

# 2-D Memory Organization

```
char A[4][3];
```

A is an array of size 4.

Each element of A is  
an array of 3 chars



# 2-D Array Example

```
const int NumStudents = 10;
const int NumHW = 3;
double grades[NumStudents][NumHW];

for (int i=0;i<NumStudents;i++) {
    for (int j=0;j<NumHW;j++) {
        cout << "Enter HW " << j <<
            " Grade for student number " <<
            i << endl;
        cin >> grades[i][j];
    }
}
```

## 2-D Array (cont.)

**You don't need to specify the size of the first dimension  
You must include all other sizes!**



```
double student_average( double g[][NumHW], int stu) {  
    double sum = 0.0;  
    for (int i=0;i<NumHW;i++)  
        sum += g[stu][i];  
  
    return(sum/NumHW);  
}
```

# Another way

```
double array_average( double a[], int len) {  
    double sum = 0.0;  
    for (int i=0;i<len;i++)  
        sum += a[i];  
  
    if (len==0)  
        return(0);  
    else  
        return(sum/len);  
}
```

**← Division by 0 is bad idea!**

# Two ways to do it

```
// Using student_average with grades
for (int i=0;i<NumStudents;i++)
    cout << "Student #" << i << " has average "
        << student_average( grades, i ) << endl;
```

-Or-

```
// Using array_average with grades
for (int i=0;i<NumStudents;i++) {
    cout << "Student #" << i << " has average "
        << array_average( grades[i], NumHW ) << endl;
```

# Arrays and pass-by-reference

- Arrays are always passed by reference
  - Inside a function any changes you make to array values are for keeps!
  - You can write functions that modify the contents of an array.
  - You need to make sure that a function knows how big the array is!!!

# A Bad Idea

```
int read_array( int a[] ) {
    int i=0;
    int val;
    do {
        cout << "Enter next value, 0 to end\n";
        cin >> val;
        if (val) a[i++] = val;
    } while (val);
    return(i); // returns the number or numbers
}
```

The problem is that the function might go beyond the size of the array.

Result: Segmentation Violation (or worse!).

# C++ does not have *bounds checking*

```
int a[6];  
int foo;
```

