Arrays

‘primitive’ arrays v. vectors (1)

allocate new, 10 elements

```
int a[10];
```

```
vector<int> v(10);
// or:
vector<int> v;
v.resize(10);
```

‘primitive’ arrays v. vectors (1)

allocate new, 10 elements

```
int a[10];              vector<int> v(10);
                        // or:
                        vector<int> v;
                        v.resize(10);
```

access (no bounds checking)

```
int foo = a[3];         int foo = v[3];
a[4] = 17;              v[4] = 17;
```

‘primitive’ arrays v. vectors (1)

allocate new, 10 elements

```
int a[10];
```

```
vector<int> v(10);
// or:
vector<int> v;
v.resize(10);
```

access (no bounds checking)

```
int foo = a[3];
a[4] = 17;
```

```
int foo = v[3];
v[4] = 17;
```

access (with bounds checking)

```
/* no equivalent */
```

```
int foo = v.at(3);
v.at(4) = 17;
```

‘primtive’ arrays v. vectors (2)

copy

```
int a[10]; int b[10];
...
// a = b does NOT work
for (int i = 0;
     i < 10; ++i)
    a[i] = b[i];
```

```
vector<int> a;
vector<int> b;
...
a = b;
```

‘primitive’ arrays v. vectors (3)

equality

```
int a[10]; int b[10];
...

// a == b does NOT work
  // instead: checks if
  // a, b are same array
  // (not same values)

bool isEqual = true;
for (int i = 0; i < 10; ++i)
    isEqual = (isEqual &&
        a[i] == b[j]);
```

```
vector<int> a;
vector<int> b;
...

bool isEqual = (a == b);
```

arrays, pointers, and memory (1)

```
int someInts[3] = {2, 4, 6};
int *pointer = someInts;
```

memory

address	value
...	...
0x10000	2
0x10004	4
0x10008	6
...	...
???	0x10000
...	...

arrays, pointers, and memory (1)

```
int someInts[3] = {2, 4, 6};
int *pointer = someInts;
```

memory

address	value	
…	…	
0x10000	2	someInts[0]
0x10004	4	someInts[1]
0x10008	6	someInts[2]
…	…	
???	0x10000	pointer
…	…	

arrays, pointers, and memory (2)

```
int someInts[3] = {2, 4, 6};

cout << "someInts␣is:␣" << someInts << endl;
cout << "&someInts[0]␣is:␣" << &someInts[0] << endl;
cout << "&someInts[1]␣is:␣" << &someInts[1] << endl;
cout << "someInts[1]␣is:␣" << someInts[1] << endl;
```

example output:

```
someInts is: 0x7ffda5455b44
&someInts[0] is: 0x7ffda5455b44
&someInts[1] is: 0x7ffda5455b48
someInts[1] is: 4
```

arrays, pointers, and memory (2)

```
int someInts[3] = {2, 4, 6};

cout << "someInts␣is:␣" << someInts << endl;
cout << "&someInts[0]␣is:␣" << &someInts[0] << endl;
cout << "&someInts[1]␣is:␣" << &someInts[1] << endl;
cout << "someInts[1]␣is:␣" << someInts[1] << endl;
```

example output:

array implicitly converted to pointer to first element

```
someInts is: 0x7ffda5455b44
&someInts[0] is: 0x7ffda5455b44
&someInts[1] is: 0x7ffda5455b48
someInts[1] is: 4
```

arrays, pointers, and memory (2)

```
int someInts[3] = {2, 4, 6};

cout << "someInts␣is:␣" << someInts << endl;
cout << "&someInts[0]␣is:␣" << &someInts[0] << endl;
cout << "&someInts[1]␣is:␣" << &someInts[1] << endl;
cout << "someInts[1]␣is:␣" << someInts[1] << endl;
```

example output:

arrays elements always at adjacent addresses
(4 bytes apart = ints are 4 bytes)

```
someInts is: 0x7ffda5455b44
&someInts[0] is: 0x7ffda5455b44
&someInts[1] is: 0x7ffda5455b48
someInts[1] is: 4
```

arrays, pointers, and memory (2)

```
int someInts[3] = {2, 4, 6};

cout << "someInts␣is:␣" << someInts << endl;
cout << "&someInts[0]␣is:␣" << &someInts[0] << endl;
cout << "&someInts[1]␣is:␣" << &someInts[1] << endl;
cout << "someInts[1]␣is:␣" << someInts[1] << endl;
```

exan

general rule: `&array[i] = array—addr + sizeof(array—elem) * i`

```
someInts is: 0x7ffda5455b44
&someInts[0] is: 0x7ffda5455b44
&someInts[1] is: 0x7ffda5455b48
someInts[1] is: 4
```

arrays as function parameters

```
void someFunc(int ptrToArray[], int size) { /* code */ }
int main() {
    int someInts[3];
    someFunc(someInts, 3);
    return 0;
}
```

is exactly equivalent to:

```
void someFunc(int *ptrToArray, int size) { /* code */ }
int main() {
    int someInts[3];
    someFunc(someInts, 3);
    return 0;
}
```

arrays as function parameters

```
void someFunc(int ptrToArray[], int size) { /* code */ }
int main() {
    int someInts[3];
    so
    re
}
```

ptrToArray is always an array

example: sizeof(ptrToArray) == sizeof(int*)

(even though sizeof(someInts) == 3 * sizeof(int))

is exactly equivalent to:

```
void someFunc(int *ptrToArray, int size) { /* code */ }
int main() {
    int someInts[3];
    someFunc(someInts, 3);
    return 0;
}
```

arrays of arrays

AKA multidimensional arrays

```
int m[2][3];
int n[2][3] = { {1,2,3}, {4,5,6} };
```

arrays of arrays

AKA multidimensional arrays

```
int m[2][3];
int n[2][3] = { {1,2,3}, {4,5,6} };

// "row" 1
n[0][0] == 1
n[0][1] == 2
n[0][2] == 3

// "row" 2
n[1][0] == 4
n[1][0] == 5
n[1][0] == 6
```

array of array storage

“row-major” order

address	value
…	…
`0x1000`	1
`0x1004`	2
`0x1008`	3
`0x100C`	4
`0x1010`	5
`0x1014`	6
…	…

array of array storage

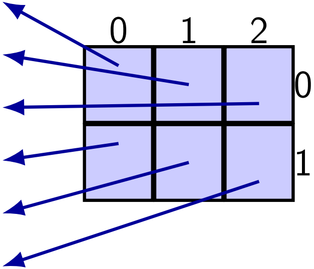
“row-major” order

address	value	
…	…	
`0x1000`	1	`n[0][0]`
`0x1004`	2	`n[0][1]`
`0x1008`	3	`n[0][2]`
`0x100C`	4	`n[1][0]`
`0x1010`	5	`n[1][1]`
`0x1014`	6	`n[1][2]`
…	…	

array of array storage

“row-major” order

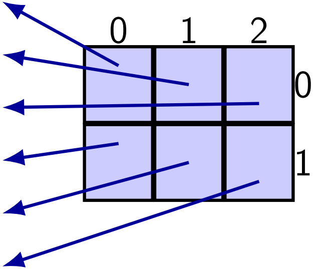
address	value	
…	…	
`0x1000`	1	`n[0][0]`
`0x1004`	2	`n[0][1]`
`0x1008`	3	`n[0][2]`
`0x100C`	4	`n[1][0]`
`0x1010`	5	`n[1][1]`
`0x1014`	6	`n[1][2]`
…	…	



array of array storage

“row-major” order

address	value		
…	…		
0x1000	1	n[0]	n[0][0]
0x1004	2		n[0][1]
0x1008	3		n[0][2]
0x100C	4	n[1]	n[1][0]
0x1010	5		n[1][1]
0x1014	6		n[1][2]
…	…		



pointers to array

(we will not test you on this)

```
int m[3][2] = { {1,2,3}, {4,5,6} }

// p1 is a "pointer to array of 3 ints"
    // yes, this syntax is really confusing
    // and generally not worth using
int (*p1)[3];
p1 = m;   // p1 contains address of m[0]
cout << p1 << endl;
    // OUTPUT: 0x... --- address of m[0][0]
cout << p1[1] << endl;
    // OUTPUT: 0x... --- address of m[1][0]

cout << p1[1][1] <<endl;
    // OUTPUT: 5
```

command line parameters

```
int main (int argc, char* argv[]) { ... }
// same as:
int main (int argc, char **argv) { ... }
```

`argc` — number of arguments + 1

`argv` — array of C-style strings

- `argv[0]` — program name
- `argv[1]`, `argv[2]`, ...— arguments

what about `int main() { ... }`?

- okay, but can't get arguments

C strings to strings

given a `char *c_style_string` (like `argv[i]`)

output:

```
cout << c_style_string
```

convert to C++-style string called `s`:

```
string s(c_style_string)
string s; s = c_style_string;
```

command line parameters

```
int main (int argc, char* argv[]) {
    // The 0th command line parameter is the program name.
    cout << "This␣program␣is␣called␣'" << argv[0]
         << "'" << endl;
    cout << "The␣following␣are␣the␣command␣"
         << "line␣parameters␣you␣specified:␣" << endl;
    // for loop starts at 1 to avoid printing
    // name of program (again)
    for ( int i = 1; i < argc; i++ ) {
        // we can convert the C-style strings into
        // C++-style strings, and then print them:
        string s(argv[i]);
        cout << "\t" << s << endl;
    }
    return 0;
}
```

command line parameters

```
int main (int argc, char* argv[]) {
    // The 0th command line parameter is the program name.
    cout << "This␣program␣is␣called␣'" << argv[0]
         << "'" << endl;
    cout << "The␣following␣are␣the␣command␣"
         << "line␣parameters␣you␣specified:␣" << endl;
    // for loop starts at 1 to avoid printing
    // name of program (again)
    for ( int i = 1; i < argc; i++ ) {
        // we can convert the C-style strings into
        // C++-style strings, and then print them:
        string s(argv[i]);
        cout << "\t" << s << endl;
    }
    return 0;
}
```

command line parameters

```
int main (int argc, char* argv[]) {
    // The 0th command line parameter is the program name.
    cout << "This␣program␣is␣called␣'" << argv[0]
         << "'" << endl;
    cout << "The␣following␣are␣the␣command␣"
         << "line␣parameters␣you␣specified:␣" << endl;
    // for loop starts at 1 to avoid printing
    // name of program (again)
    for ( int i = 1; i < argc; i++ ) {
        // we can convert the C-style strings into
        // C++-style strings, and then print them:
        string s(argv[i]);
        cout << "\t" << s << endl;
    }
    return 0;
}
```