# Chapter 18: Vector and Arrays

# Plan for today

- We will talk about:
  - Essential operations
  - Implicit conversions and explicit constructors
  - Access to vector elements via [ ]
    - Overloading on `const`
  - arrays
    - Pointers to array elements
    - Pointers and array
    - Array initialization
  - palindromes

# Essential operations

- Constructors from one or more arguments
- Default constructor

- Copy constructor (copy object of same type)
- Copy assignment (copy object of same type)
- Move constructor (move object of same type)
- Move assignment (move object of same type)
- Destructor

# Essential operations: when to define

- Constructors from one or more arguments
- Default constructor (meaningful and obvious default value)

- Copy constructor (copy object of same type)
- Copy assignment (copy object of same type)
- Move constructor (move object of same type)
- Move assignment (move object of same type)
- Destructor

# Essential operations: when to define

- Constructors from one or more arguments
- Default constructor (meaningful and obvious default value)

- Copy constructor (copy object of same type)
- Copy assignment (copy object of same type)
- Move constructor (move object of same type)
- Move assignment (move object of same type)
- Destructor

If an object has acquired a resource (and has a pointer member pointing to it), the default value of copy is almost certainly wrong

# Essential operations: when to define

- Constructors from one or more arguments
- Default constructor (meaningful and obvious default value)

- Copy constructor (copy object of same type)
- Copy assignment (copy object of same type)
- Move constructor (move object of same type)
- Move assignment (move object of same type)
- Destructor

A class needs a destructor if it acquires resources

# Initialization: implicit conversions and explicit constructors

- A problem:
  - A constructor taking a single argument defines a conversion from the argument type to the constructor's type
  - Our vector had `vector::vector(int)`, so

```
vector v1 = 7; // v1 has 7 elements, each with the value 0 (odd)
v1 = 20; // v1 is now a new vector with 20 elements (even more odd)
```

  - This is very error-prone, unless of course this is what we want

# Initialization: implicit conversions and explicit constructors

- A solution:
  - Declare constructors taking a single argument **`explicit`**

```
class vector {
  // …
  explicit vector(int);
  // …
};
 vector v1 = 7;  // error: no implicit conversion from int
 v1 = 20; // error: no implicit conversion from int
```

# Access to vector elements

- So far we used **`get()`** and **`set()`** member functions to access elements.
- Such access is verbose and ugly

# Access to vector elements

- So far we used **`get()`** and **`set()`** member functions to access elements.
- Such access is verbose and ugly, how about `[]` access?
  - `a[4]=12.5; // create a vector of 10 elements`

# Access to vector elements

- So far we used **`get()`** and **`set()`** member functions to access elements.
- Such access is verbose and ugly, how about `[]` access?
  - `a[4]=12.5; // create a vector of 10 elements`
- Define/overload the `operator[ ]`

```
double& operator[ ](int n)
{   return elem[n]; }
```

# operator[ ]

- We defined the `operator[]` as

```
double& operator[](int n)
{   return elem[n]; }
```

- Recall that we passed the vector by reference, as a constant:

```
double average(const vector& a)
```

  - It won't work with our implementation of the `operator[]`

# operator[ ]

- We defined the `operator[]` as

```
double& operator[](int n)
{   return elem[n]; }
```

- Recall that we passed the vector by reference, as a constant:

```
double average(const vector& a)
```

  - It won't work with our implementation of the `operator[]`

- To make it work, we will add one more definition:

```
double operator[](int n) const
{   return elem[n]; }
```

  - This version of `operator[]` will be working with const vectors

# Arrays

- For a while we used *array* to refer to a sequence of objects allocated on the free store

# Arrays

- For a while we used *array* to refer to a sequence of objects allocated on the free store
- They don't have to be on the free store. They are common as:
  - global variables (often a bad idea)  char name[20];
  - local variables (have serious limitations)

```
const int m = 10;                    // get value of n
int a[m];                            int a[n]; // error
```

  - function arguments (doesn't know its size)
  - class members (maybe hard to initialize)

# Arrays

- You might have a not-so-subtle bias in favor of vector(s) over arrays
- However, arrays existed long before vectors an are roughly equivalent to what is offered in other languages (like C), so we must know arrays (and know them well to be able to cope with older code and with code written by people who don’t appreciate the advantages of vector)

# Arrays

- An *array* is a homogeneous sequence of objects allocated in contiguous memory; the elements are numbered from 0 upward
- In a declaration, an array is indicated by the [ ]

```
const int max = 100;
int myArray[max];
```

# Arrays

- An *array* is a homogeneous sequence of objects allocated in contiguous memory; the elements are numbered from 0 upward
- In a declaration, an array is indicated by the [ ]

```
const int max = 100;
int myArray[max];
```

Local array “lives” until the end of its scope.

Limitation: the size of `myArray` must be known before the compile time, so we have to plan ahead, we assumed that it will have at most 100 elements.

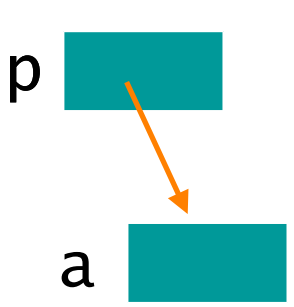
see the file **arrayWork.cpp**

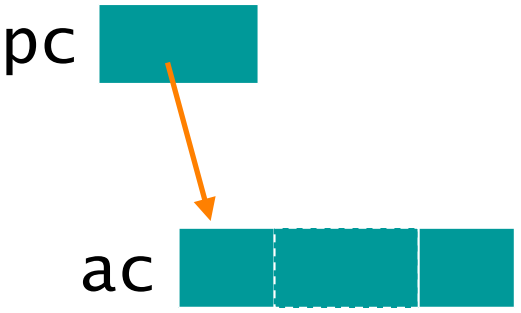
# Address of: &

- You can get a pointer to any object, not just to objects on the free store

```
int a;
char ac[20];

void f(int n) {
    int b;
    int* p = &b; // pointer to individual variable
    p = &a;          // now point to a different variable
    char* pc = ac; // the name of an array names a pointer to its first element
    pc = &ac[0];   //equivalent to pc = ac
    pc = &ac[n];  // pointer to ac's nth element (starting at 0th)
    // warning: range is not checked
}
```



# Arrays (often) convert to pointers

```
void f(int pi[ ]) // equivalent to void f(int* pi)
 {
   int a[ ] = { 1, 2, 3, 4 };
   int b[ ] = a;   // error: copy isn't defined for arrays
   b = pi; // error: copy isn't defined for arrays.
   pi = a; // ok: but it doesn't copy: pi now points to a's first element
           // Is this a memory leak? (maybe)
   int* p = a;  // p points to the first element of a
   int* q = pi; // q points to the first element of a
}
```

# Pointers to array elements

```
double a[10];
double* p = &a[5]; // p points to a[5]
*p = 7.3;
p[2] = 5.1;
p[-3] = 9.2;
p += 2; // moves p 2 elements to the right
p -= 5; // moves p 5 elements to the left
for( p = &a[0]; p < &a[10]; ++p)
{ cout << *p << " "; } // prints all the elements of a
for( p = &a[9]; p >= &a[0]; --p)
{ cout << *p << " "; } // prints all the elements of a
                       // backwards
```

# Be careful with arrays and pointers

```
char ch[20];
char* p = &ch[90];
   // …
*p = 'a'; // we don't know what this will overwrite
char* q;  // forgot to initialize
*q = 'b'; // we don't know what this will overwrite
```

# Pointers and arrays: more examples

Consider the function that counts the number of characters in a zero-terminated array of characters:

```
int strlen(const char* p) // int strlen(const char a[])
{ int count = 0;
  While (*p) { ++ count; ++p;}
  return count;
}
// similar to standard library strlen()
char name[] = “Natalia”;
strlen(name); //ok
strlen(&name[0]); // ok
```

# Why bother with arrays?

- It’s all that C has
  - In particular, C does not have vector
  - There is a lot of C code “out there”  ( “a lot” means N*1B lines)
  - There is a lot of C++ code in C style “out there” ( “a lot” means N*100M lines)
  - You’ll eventually encounter code full of arrays and pointers
- They represent primitive memory in C++ programs
  - We need them (mostly on free store allocated by new) to implement better container types
- Avoid arrays whenever you can
  - They are the largest single source of bugs in C and (unnecessarily) in C++ programs
  - They are among the largest sources of security violations, usually (avoidable) buffer overflows

# Palindromes

- [simple definition] A palindrome is a word that is spelled the same from both ends
  - Examples: anna, madam, racecar, etc.
- [definition] A palindrome is a word, number, phrase, or other sequence of symbols that reads the same backwards as forwards, ignoring punctuation symbols and lower/upper case
  - Examples: race car;  Madam, I'm Adam!

# Palindromes using string

Idea: start reading the string from the front and the back, compare the letters, move into the middle;

```
bool is_palindrome(const string& s) {
    int first = 0;
    int last = s.length() - 1;
    while ( first < last) {
        if ( s[first] != s[last] ) return false;
        ++ first;
        --last;
    }
    return true;
}
```

# Palindromes using array

Idea: start reading the string from the front and the back, compare the letters, move into the middle

```
bool is_palindrome(const char s[], int n) {
    int first = 0;
    int last = n - 1;
    while ( first < last) {
        if ( s[first] != s[last] ) return false;
        ++ first;
        --last;
    }
    return true;
}
```

# Palindromes using pointers

Idea: start reading the string from the front and the back, compare the letters, move into the middle

```
bool is_palindrome(const char* first, const char* last) {
    while ( first < last) {
        if ( *first != *last ) return false;
        ++ first;
        --last;
    }
    return true;
}
```

See the file **palindromes.cpp** for their use

# Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html

- Class textbook