Chapter 17: Vector and Free Store

Plan for today

- We will talk about:
  - Memory deallocation
  - Memory leaks
  - Simplified vector class

The sizeof operator

- So how much memory does an int really take up? A pointer?
  - The operator `sizeof` answers such questions

```
int a = 10;
int* p = &a;
cout << "An integer occupies "<< sizeof(a)<<" bytes\n";
cout << "A pointer to an integer occupies "
     << sizeof(p) << " bytes\n";
char* pc = &c;
cout << "A char occupies " << sizeof(c) << " bytes\n";
cout << "A pointer to a char occupies "
     << sizeof(pc) << " bytes\n";
```

See more in **sizeOfoperatorUse.cpp**

Pointers, arrays, and vector

Note:

- With pointers and arrays we are "touching" hardware directly with only the most minimal help from the language. Here is where serious programming errors can most easily be made, resulting in malfunctioning programs and obscure bugs
  - Be careful and operate at this level only when you really need to
  - If you get "segmentation fault", "bus error", or "core dumped", suspect an uninitialized or otherwise invalid pointer
- vector is one way of getting almost all of the flexibility and performance of arrays with greater support from the language (read: fewer bugs and less debug time).

free store deallocation

- The `new` operator allocates ("get") the memory from the free store
- Computer's memory is limited, hence
- It is a good idea to "return" memory to the free store once we finished using it
  - `delete` frees the memory for an individial object allocated by `new`
  - `delete []` frees the memory for an array of objects allocated by `new`
- If we do not deallocate the memory, we will have a memory leak.

free store deallocation: examples

```
int* p = new int{6};    // allocate one initialized to 6  int
int* q = new int[7];    // allocate seven uninitialized ints
...
delete p;
delete [] q;
```

free store deallocation: errors

Deleting an object twice is a mistake:

```
int* p = new int{6};
delete p;  // ok, p points to an object created by new
// … no use of p here …
delete p;  // error: p points to memory owned by the free-store manager
```

free store deallocation: errors

Deleting an object twice is a mistake:

```
int* p = new int{6};
delete p;  // ok, p points to an object created by new
// … no use of p here …
delete p;  // error: p points to memory owned by the free-store manager
```

Two problems with the second delete:

- We don’t own the object pointed to anymore so the free-store manager may have changed the internal data structure in such a way that it can’t correctly execute **delete p** again.
- The free-store manager may have “recycled” the memory pointed to by **p** so that **p** now points to another object: deleting that object (owned by some other part of the program) will cause errors in our program.

free store deallocation: nullptr

Deleting null pointer doesn't do anything, because the nullptr doesn't point to an object, so deleting it is harmless.

```
int* p = nullptr;
delete p;  // ok, no action is needed
// … no use of p here …
delete p;  // ok, still no action is needed
```

Vector (construction and primitive access)

- a very simplified vector of doubles:

```
class vector{
    int sz; // the size
    double* elem; // a pointer to the elements
public:
    vector(int s): sz(s), elem(new double[s]) // constructor
    {
        for(int i{0}; i<s; ++i) elem[i] = 0;
    }

};
```

Vector (construction and primitive access)

- a very simplified vector of doubles:

```
class vector{
    int sz; // the size
    double* elem;   // a pointer to the elements
public:
    vector(int s): sz(s), elem(new double[s]) // constructor
    {
        for(int i{0}; i<s; ++i) elem[i] = 0;
    }
    double get(int n) const {return elem[n]}; // access:read
    void set(int n, double v) { elem[n] = v;} // access:write
    int size() const {return sz;} // the current size
};
```

Vector (construction and primitive access)

- a very simplified vector of doubles:

```
class vector{
    int sz; // the size
    double* elem;   // a pointer to the elements
public:
    vector(int s): sz(s), elem(new double[s]) // constructor
    {
        for(int i{0}; i<s; ++i) elem[i] = 0;
    }
    double get(int n) const {return elem[n];}; // access:read
    void set(int n, double v) { elem[n] = v;} // access:write
    int size() const {return sz;} // the current size
};
```

If we do not deallocate the memory we will have a memory leak

Vector (construction and primitive access)

- a very simplified vector of doubles:

```
class vector{
    int sz; // the size
    double* elem;  // a pointer to the elements
public:
    …
    ~vector()  // destructor
    { delete [] elem; }
};
```

If we do not deallocate the memory we will have a memory leak

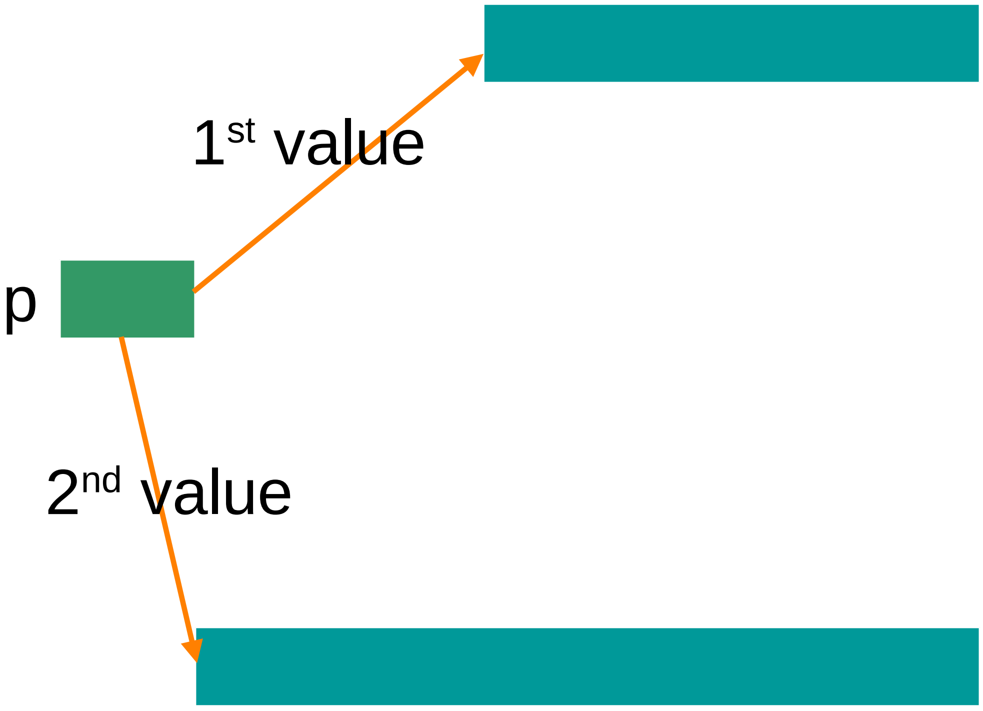
Memory leaks

- A program that needs to run “forever” can’t afford any memory leaks
  - An operating system is an example of a program that “runs forever”
- If a function leaks 8 bytes every time it is called, how many days can it run before it has leaked/lost a megabyte?
  - Trick question: not enough data to answer, but about 130,000 calls
- All memory is returned to the system at the end of the program
  - If you run using an operating system (Windows, Unix, whatever)
- Program that runs to completion with predictable memory usage may leak without causing problems
  - i.e., memory leaks aren’t “good/bad” but they can be a major problem in specific circumstances

Memory leaks

- Another way to get a memory leak

```
void f()
{
    double* p = new double[27];
    // …
    p = new double[42];
    // …
    delete[] p;
}
```



// 1st array (of 27 doubles) leaked

Memory leaks

- How do we systematically and simply avoid memory leaks?
  - don't mess directly with new and delete
    - Use vector, etc.
  - Or use a garbage collector
    - A garbage collector is a program the keeps track of all of your allocations and returns unused free-store allocated memory to the free store (not covered in this course; see http://www.stroustrup.com/C++.html)
    - Unfortunately, even a garbage collector doesn’t prevent all leaks
    - See also Chapter 25 (self-development)

Free store summary

- Allocate using new
  - New allocates an object on the free store, sometimes initializes it, and returns a pointer to it

```
int* pi = new int;      // default initialization (none for int)
char* pc = new char('a');   // explicit initialization
double* pd = new double[10];  // allocation of (uninitialized) array
```

  - New throws a **bad_alloc** exception if it can't allocate (out of memory)
- Deallocate using **delete** and **delete[ ]**
  - delete and delete[ ] return the memory of an object allocated by new to the free store so that the free store can use it for new allocations

```
delete pi;   // deallocate an individual object
delete pc;   // deallocate an individual object
delete[ ] pd;  // deallocate an array
```

  - Delete of a zero-valued pointer ("the null pointer") does nothing

```
char* p = 0; // C++11 would say char* p = nullptr;
delete p; // harmless
```

In-class practice

- Consider the following code fragment:

```
int *b{ nullptr }, *c{ nullptr }, x, y;
x = 3;
y = 5;
b = &x;
c = &y;
*b = 4;
*c = *b + *c;
c = b;
*c = 2;
```

- Let’s make a sketch of the memory for it:

In-class practice

- Let’s use the implementation of our simplified vector of doubles:
  - Use it to create a vector of 10 elements:

    {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
  - Display all the values of the vector
  - Define a member function that would display the values of the vector
  - Overload the cout operator<< to be used with objects of this class
  - Define a member function `resize (int newSz)` that will resize the vector to the new size, preserving all the existing elements

Resources used for these slides

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

- Class textbook