



CS 11 C++ track: lecture 4

- Today:
 - More on memory management
 - the stack and the heap
 - `inline` functions
 - `structs` vs. `classes`



The stack and the heap

- Two places where memory lives:
 - the stack (local variables)
 - the heap (long-lived data)
- Allocation is different for these



The stack (1)

- Local variable storage
- Objects allocated when enter block
- Objects deleted when leave block
- Destructor called automatically



The stack (2)

```
void foo() {  
    Matrix m(10, 20);  
    // ...  
} // Matrix destructor called here
```



The heap (1)

- Heap is the main computer memory
- Allocate explicitly with `new`
- Deallocate explicitly with `delete`
- If don't `delete`, memory never freed



The heap (2)

```
void foo2() {  
    Matrix *m = new Matrix(10, 20);  
  
    // ...  
  
    delete m; // NOT implicit!  
  
}
```



The heap (3)

- Not remembering to delete memory causes “memory leaks”
 - very common problem
 - invalid destructor can also leak memory
 - can cause memory exhaustion
 - difficult to track down!



The heap (4)

```
void foo3() {  
    Matrix *m1 = new Matrix(10, 20);  
    Matrix m2 = Matrix(10, 20);  
    delete m1; // NOT implicit!  
    // what about m2?  
}
```



`inline` functions (1)

- Function calls are not free
- Small functions can benefit from being expanded in-place
- This is called “inlining”



inline functions (2)

```
inline int foo(int x) {  
    return 2 * x + 5;  
}
```

```
void bar(int x) {  
    cout << foo(x) << endl;  
    // cout << 2 * x + 5 << endl;  
}
```



inline functions (3)

```
class foo {
private:
    int x, y;
public:
    foo(int nx, int ny); // constructor
    void reset() { // inline method
        x = 0; y = 0;
    }
    // other code
};
```



inline functions (4)

- Method def'n in class declaration is implicitly inline
- All method defs in header file should be inline
 - explicit if outside of class decl
 - otherwise can get linker errors
- Can also have inline fns in .cc files



inline functions (5)

```
class foo { // in foo.hh
public:
    foo() { ... }
    void do_stuff();
};
```

```
inline void foo::do_stuff() {
    //...
}
```



inline functions (6)

- Rules of thumb for inlining:
 - Inline **small** functions
 - Don't overdo it!
 - Compiler can ignore inline directives
 - Inlining speeds up but increases space usage
 - **Always** inline functions/methods in header file!
 - Rarely useful elsewhere



structs VS. classes

- **struct** is not the same as C **struct**
- **struct** is just a **class** where all members are public by default
- useful for very small objects that are used privately by a class
- can have methods, constructors, etc.