
CS11 – Introduction to C++

Fall 2009-2010

Lecture 4

Topics for Today

- Inline functions
 - Initializer-lists
 - Classes and structs
 - Lab 4
-

Inline Functions

- Calling functions incurs overhead
 - Creating a new stack-frame
 - Jumping to the function's code
 - Passing arguments and return-values
- Programs can have a lot of small functions

```
class Cfoo {  
    int x;  
    ...  
    int getX() { return x; } // Accessor for x  
};  
...  
cout << foo.getX();
```

- For small functions, the compiler can inline the code
 - Compiler can inline this code, producing, in effect:
cout << foo.x;
-

Inline Functions vs. Macros

- In C, macros can be used to inline code

```
#define max(x, y) ((x) > (y) ? (x) : (y))
```

- C/C++ macros are simple text substitution!
- Either **x** or **y** will be evaluated multiple times

```
    k = max(i++, j--);  
    ↪  
    k = ((i++) > (j--) ? (i++) : (j--));
```

- No type-safety
 - Extra parentheses to avoid precedence issues!
- Inline functions solve these problems
 - Inline function arguments are evaluated only once
 - Argument and return-value types are declared, and checked by compiler
-

Defining Inline Functions

- Function definitions in class declarations are *automatically* inline

```
class Point {  
    double x, y;  
    ...  
    double getX() const { return x; }    // Inline!  
    double getY() const { return y; }  
    double distanceTo(const Point &p) const;  
};
```

- Can also follow class declaration in header file

```
inline double Point::distanceTo(const Point &p) const {  
    ...  
}
```

- Keeps class declaration from becoming cluttered
 - Normal function definitions in `.cc` file are not inlined
-

Inlining Tips

- Inlining is a hint! Compiler may ignore you
 - Certain operations can prevent inlining:
 - e.g. an inline function that recursively calls itself
 - Compiler also may simply choose not to do it!
 - Inlining will probably make your binary larger
 - The function's code is replicated everywhere
 - Only inline small functions
 - Accessors are great candidates
 - Or, let the compiler decide what to inline
 - Modern compilers can figure out what to inline on their own
 - e.g. `gcc/g++`, MS Visual C++
-

Constructors and Member Variables

- So far, have written constructors like this:

```
Point::Point(double x, double y) {  
    x_coord = x; // Store x and y values.  
    y_coord = y;  
}
```

- How come this works?

- *How were `x_coord` and `y_coord` set up in the first place?!*

- Answer:

- An object's data-members are constructed *before* the class' constructor body is run.
-

Classes with Class-Members

- A graphics-engine class:

```
class GraphicsEngine {  
    Matrix viewportTransform;  
    Matrix modelViewTransform;  
    ...  
public:  
    GraphicsEngine();  
    GraphicsEngine(const RenderConfig &conf);  
    ...  
};
```

- **Matrix** constructors get called *before* **GraphicsEngine**'s constructor-body is run.
 - Specifically, the default constructors are called.
-

Constructing the Graphics Engine

- Graphics engine's constructor:

```
GraphicsEngine::GraphicsEngine() {  
    viewportTransform = Matrix(4, 4);  
    modelViewTransform = Matrix(4, 4);  
    ... // Rest of graphics-engine initialization  
}
```

- ...but the matrices were already constructed once!
 - This code does extra work:
 - 2 × default **Matrix** constructor
 - 2 × two-arg **Matrix** constructor (this is all we want!)
 - 2 × assignment-operator (and its cleanup/copy work)
 - 2 × destructor (clean up temporary objects)
-

Member Initializer Lists!

- Member initializer lists solve these problems
 - Can specify non-default construction of class data-members
- After constructor signature, before body

```
GraphicsEngine::GraphicsEngine() :  
    viewportTransform(4, 4), modelViewTransform(4, 4) {  
    ... // Rest of graphics-engine initialization  
}
```

- Colon goes before initializer list
 - Member initializations separated with commas
-

More Initializer List Details

- Best for data-members that are objects
 - Avoid extra work – default initialization, assignment, etc.
 - Gives biggest performance benefit
 - Also very useful for primitive data-members
 - Doesn't improve performance
 - Just simplifies initialization
 - Some data-members require an initializer!
 - Members whose type doesn't have a default constructor
 - **const** data-members
 - Reference data-members
 - (references aren't allowed to refer to nothing...)
 - These types require initialization details that the compiler can't guess
-

Classes and Structs

- In C++, structs are just like classes
 - They can have constructors, member functions, access modifiers, etc.
 - Only difference is that default access is public
 - `struct s { ... };`
 - `class s { public: ... };` (same thing)
 - Constructors for structs are particularly useful
 - Check for valid values, or initialize defaults
 - Can also write copy-constructors, destructors, etc.
-

Structs for Internal Data

- Structs typically used for objects that don't need all the features of classes
 - e.g. helper-objects inside an implementation
 - “a chunk of data”
 - Hiding internal structs is a good idea!
 - Part of encapsulation/abstraction
 - Good object-oriented design
-

Hiding Structs

- Can declare structs in your class-declaration

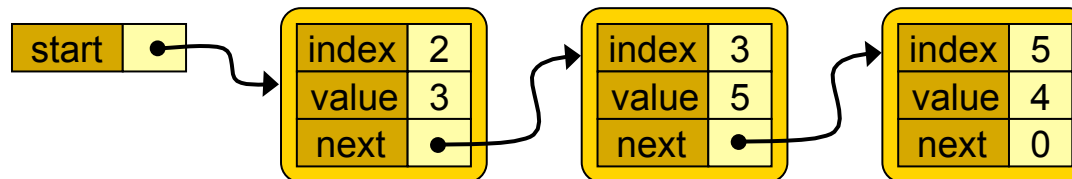
```
class Scheduler {
private:
    // A "scheduled task" struct, for internal use only
    struct task {
        int id;
        string desc;
        task *next;    // Can chain tasks together into a list
    };

    task *schedTasks;    // My list of scheduled tasks
    ...
};
```

- **task** type can be used in **Scheduler** functions
 - **task** is inaccessible outside of **Scheduler**
 - If it were public, its name would be **Scheduler::task**
-

Homework Tips!

- Lab 4 is notoriously difficult
 - Implement a sparse vector of integers
 - Only store nonzero values!
 - Singly linked list of nodes, ordered by index
- Example:
 - Vector (0, 0, 3, 5, 0, 4)
 - Stored as:



Lab 4 Approach

- Lab 4 is broken into two one-week parts
 - Lab 4a involves implementation of basic features
 - Lab 4b involves more sophisticated operations
 - Each part specifies a sequence of steps
 - Solve problems in an easy-to-hard approach
 - Reuse your efforts, to maximize your results
-

Linked-List Nodes

- A linked-list node type:

```
struct node {  
    int index;  
    int value;  
    node *next;    // Pointer to next node in list  
};
```

- This is a singly linked list
 - Each node only points to *next* node
 - Can only traverse list in one direction
 - Makes things trickier than a doubly-linked list
 - Put **node** type inside your **SparseVector** class
 - **node** is only used within **SparseVector**
-

Why a Singly Linked List?

- Sparse data structures try to minimize space usage
 - Each element takes more space than in dense version
 - ...ideally, number of nonzero elements will be “small enough” to see a benefit
 - Size of values: `int` = 4 bytes, pointer = 4 bytes
 - Dense representation: T total elements
 - Requires 4T bytes
 - Sparse representation: N nonzero elements
 - Singly linked: Requires $N \times (4_{\text{idx}} + 4_{\text{val}} + 4_{\text{nxt}}) = 12N$ bytes
 - Doubly linked: Requires $N \times (4_{\text{idx}} + 4_{\text{val}} + 4_{\text{prv}} + 4_{\text{nxt}}) = 16N$ bytes
 - In singly linked list, $N \leq T/3$ for a benefit
 - In doubly linked list, $N \leq T/4$ for a benefit
-

Linked List Construction

- Make a **node** constructor:

```
struct node {  
    int index;        // Index of element in vector  
    int value;       // Value of element in vector  
    node *next;     // Pointer to next element, or 0  
  
    node(int index, int value, int next = 0) :  
        index(index), value(value), next(next) { }  
};
```

Annotations:
- An arrow points from the text "default value" to the default argument `int next = 0`.
- An arrow points from the text "data-member" to the parameter `index` in the constructor body.
- An arrow points from the text "argument" to the parameter `next` in the constructor body.

- Empty constructor body, because setup is done with initializers

- C++ allows us to specify default values for args

```
node *n1 = new node(7, -4);    // n1->next set to 0  
node *n2 = new node(3, 2, n1); // n2->next set to n1
```

Linked List Construction (2)

- Arguments with default values must be at end of argument-list

- This is invalid:

```
void foo(int i, char *psz = 0, double x, int j = -1);
```

- Move all default args to end:

```
void foo(int i, double x, char *psz = 0, int j = -1);
```

- Can manually link nodes together, too:

```
node *n1 = new node(3, 5);    // Elem 3 = value 5  
node *n2 = new node(5, -2);  // Elem 5 = value -2  
n1->next = n2;
```



General Linked-List Tips

- Use meaningful variable names!
 - **start**, **curr**, **prev**, **next** are good candidates
 - **start** = first node of list (the “head”)
 - **curr** = current node
 - **prev** = previous node
 - **next** = next node
 - Really clear and obvious meanings
 - Makes things clear to you, and to others!
-

Linked-List Algorithms

- It takes practice to write clean algorithms with singly linked lists
 - Think about efficiency of your implementations
 - “Can I make this simpler?”
 - “Can I incorporate special cases more cleanly?”
 - Don’t be afraid to experiment and refine
 - The fewer special cases, the easier it is to find and fix bugs.
-

General Theme

- Most operations follow this form:

```
node *curr = start;

while (curr != 0) {
    ... // Do something with current node

    // Go to next node in list
    curr = curr->next;
}
```

- Simple list traversal
 - Copying, deleting, comparing lists – all variations on this theme
-

Deleting a Singly Linked List

- Deleting a list requires deleting all of its nodes
- First try:

```
node *curr = start;

while (curr != 0) {
    // Delete this node.
    delete curr;

    // Go to next node in list
    curr = curr->next;
}
```

- This won't work!
 - ❑ Can't delete a `node` object, then try to access it!
 - ❑ Need to fetch out what comes next, *before* deleting `curr`
-

Deleting a Singly Linked List (take 2)

- Add a **next** pointer to our code:

```
node *curr = start;

while (curr != 0) {
    // Get what is next, before deleting curr.
    node *next = curr->next;

    // Delete this node.
    delete curr;

    // Go to next node in list
    curr = next;
}
```

Copying a Singly Linked List

- Make a copy of another list
 - Need to traverse the other list in sequence
 - Also need the *previously created* node, to append another node to it
 - Each node only knows what node comes next...
- Start with these variables:

```
// Get a pointer to other list's first node
node *otherCurr = other.start;
```

```
// Use prev and curr to create the copy.
node *prev = 0;
node *curr;
```

Copying a Singly Linked List

```
// Get a pointer to other list's first node
node *otherCurr = other.start;

// Use prev and curr to create the copy
node *prev = 0;
node *curr;
while (otherCurr != 0) {
    // Copy other list's current node
    curr = new node(otherCurr->index, otherCurr->value);

    if (prev == 0)
        start = curr;        // curr is the first node in our copy!
    else
        prev->next = curr; // Make previous node point to current

    prev = curr;            // Done with current node!
    otherCurr = otherCurr->next; // Move to next node to copy
}
```

Retrieving Values

- **int getElem(int index)**
 - An index is specified
 - Find the node with the requested index, and return its value
 - If no node has the requested index, return 0
 - Iterate over all nodes in list.
 - If current node is 0, we hit end of list! Return 0.
 - If current node's index matches requested index, return the node's value.
 - Can stop early, if current node's index is larger than requested index.
 - Nodes are ordered by index...
-

Loop Guards

- In C++, logical Boolean operators are lazy

- Example:

```
while (curr != 0 && curr->index != index) {  
    ...  
}
```

- **Note:** Your code may need to be different. This is just an example...

- This works because:

- After the first false condition, no more conditions are evaluated
- `false && <anything> == false`

- Similarly, `||` is lazy

- After the first true condition, no more conditions are evaluated
 - `true || <anything> == true`
-

Setting Values

- `void setElem(int index, int value)`
 - This one is trickier:
 - If `index` is already in the list, but `value` is 0, must remove that element
 - If `index` isn't already in list, must add a new node
 - (This means we need a `prev` pointer.)
 - And, we must add it in the correct place.
 - Doing all this in one function is a mess:
 - The node exists, `value != 0`
 - The node exists, `value == 0`
 - The node doesn't exist, `value != 0`
 - The node doesn't exist, `value == 0`
-

Setting Values (2)

- Simplify the problem by breaking it down:
 - Write one function for when setting value to 0:
 - `removeElem(int index)`
 - Another for when `value` isn't 0:
 - `setNonzeroElem(int index, int value)`
 - Use an `assert` in this function to check `value`!
 - Both functions traverse the list to find the node, or the place where the node should go.
 - These are helper functions: *not* for the user!
 - Make them private.
-

Final Notes

- A test program will exercise this week's features
 - Comment your code well, follow good style!
 - This will help you at least as much as it helps your grader!
 - Ask for help if you get totally stuck! 😊
-