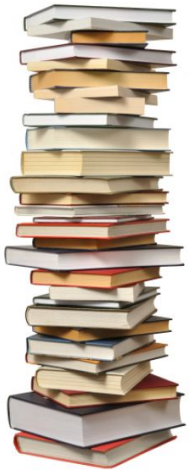
CS24 - Problem Solving with Computers II

More on memory + Linked Lists

Memory

Stack (LIFO) - think stack of books



Heap (can grab from anywhere) - think heap of clothes

```cpp
#include <iostream> // Input output stream
using namespace std;  // Standard namespace

int InnerFunction(int outerFunctionNumber){ // Used by OuterFunction()
    return outerFunctionNumber+1; // returns the number incremented by 1
}

int OuterFunction(int startingNumber){  // Decreases startingNumber then uses InnerFunction() to increment number
    return InnerFunction( outerFunctionNumber: startingNumber -1); // Calls InnerFunction on decremented number
}

int main() {
    int functionNumber = OuterFunction( startingNumber: 3);
    cout << functionNumber << endl; //Prints a 3
    return 0;
}
```

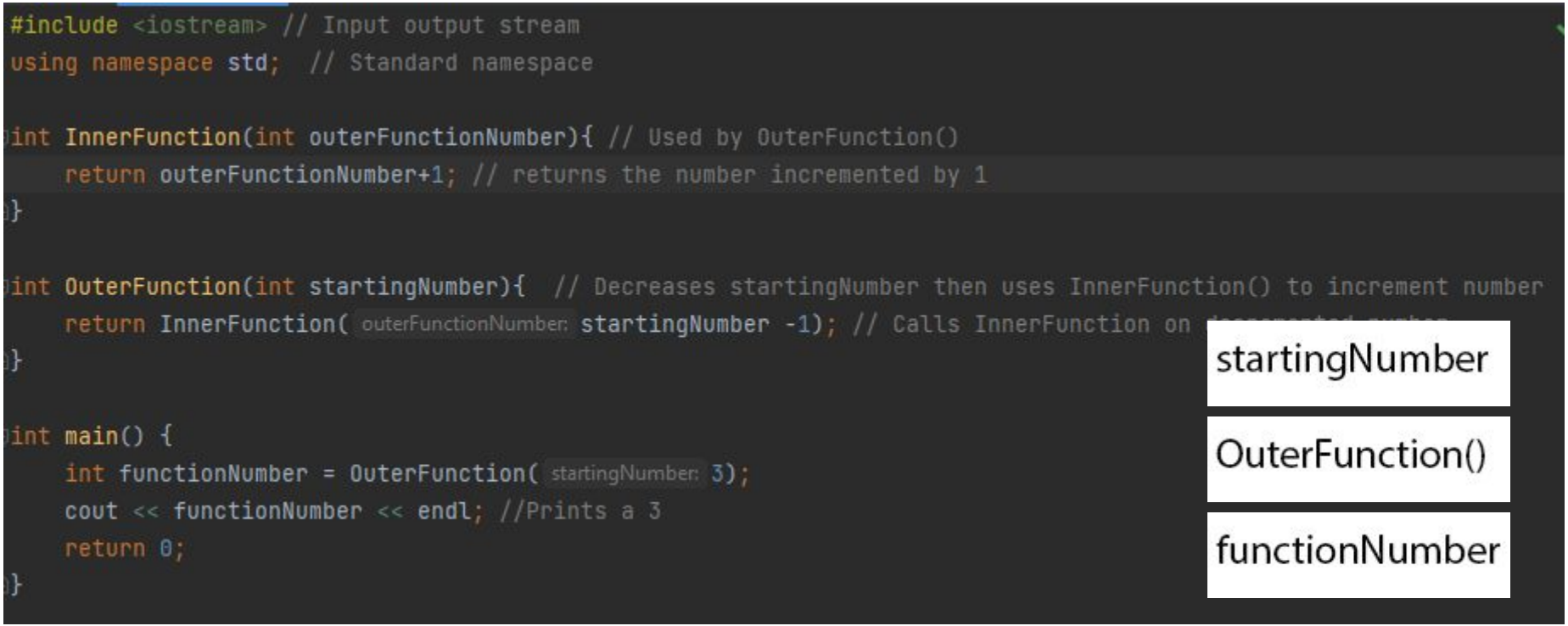
```cpp
#include <iostream> // Input output stream
using namespace std;  // Standard namespace

int InnerFunction(int outerFunctionNumber){ // Used by OuterFunction()
    return outerFunctionNumber+1; // returns the number incremented by 1
}

int OuterFunction(int startingNumber){  // Decreases startingNumber then uses InnerFunction() to increment number
    return InnerFunction( outerFunctionNumber: startingNumber -1); // Calls InnerFunction on decremented number
}

int main() {
    int functionNumber = OuterFunction( startingNumber: 3);
    cout << functionNumber << endl; //Prints a 3
    return 0;
}
```

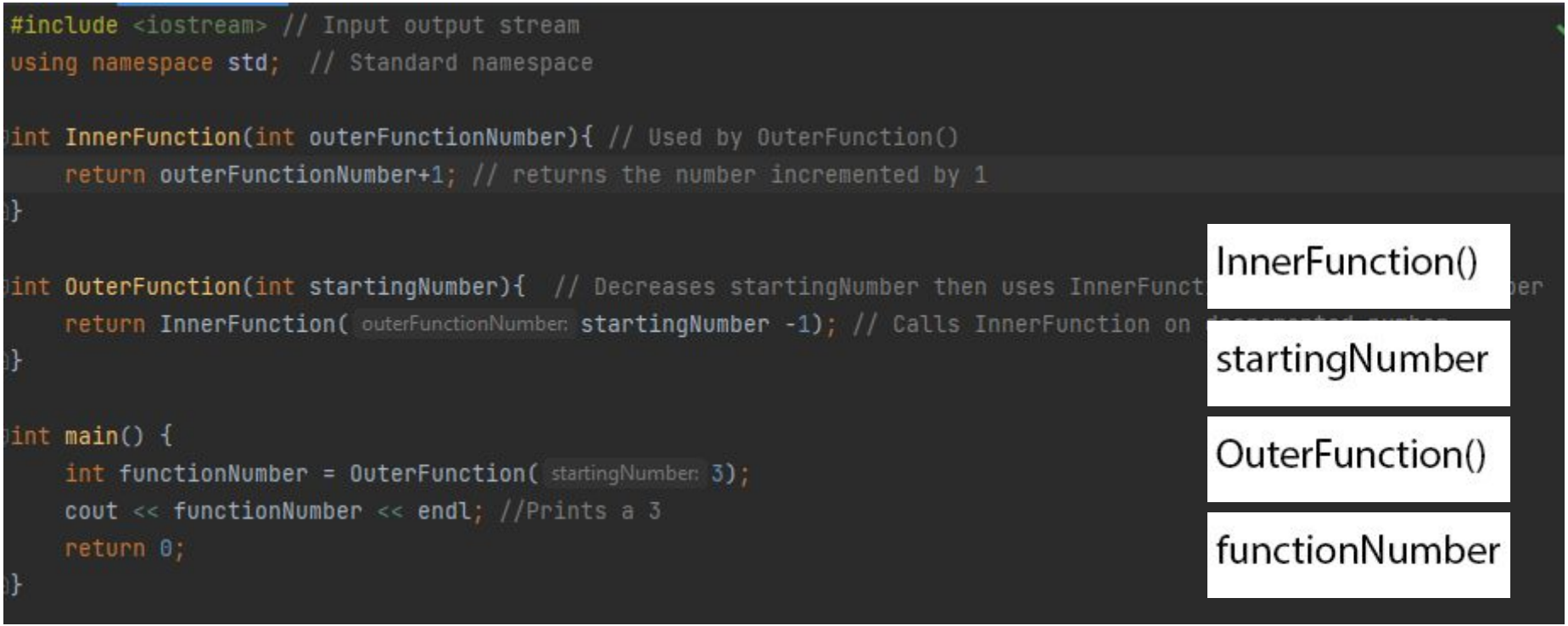
functionNumber

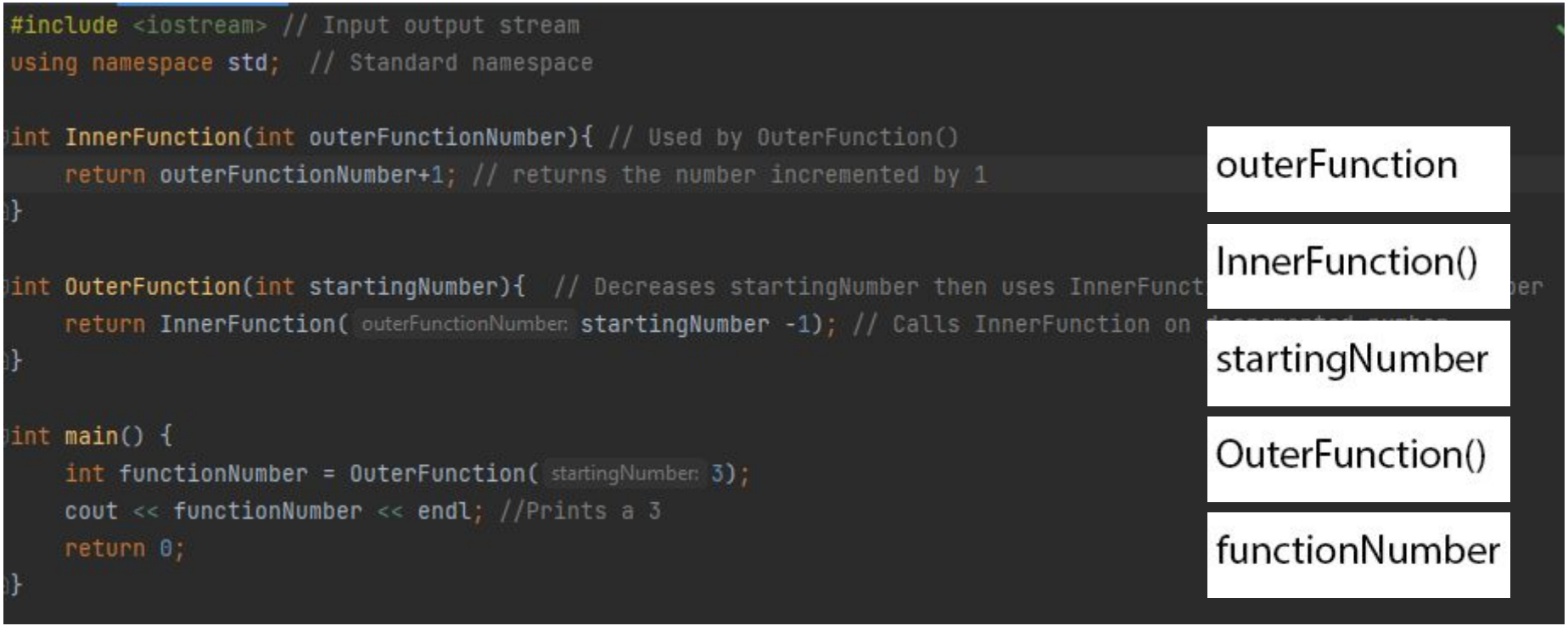
```cpp
#include <iostream> // Input output stream
using namespace std;  // Standard namespace

int InnerFunction(int outerFunctionNumber){ // Used by OuterFunction()
    return outerFunctionNumber+1; // returns the number incremented by 1
}

int OuterFunction(int startingNumber){  // Decreases startingNumber then uses InnerFunction() to increment number
    return InnerFunction( outerFunctionNumber: startingNumber -1); // Calls InnerFunction on decremented number
}

int main() {
    int functionNumber = OuterFunction( startingNumber: 3);
    cout << functionNumber << endl; //Prints a 3
    return 0;
}
```

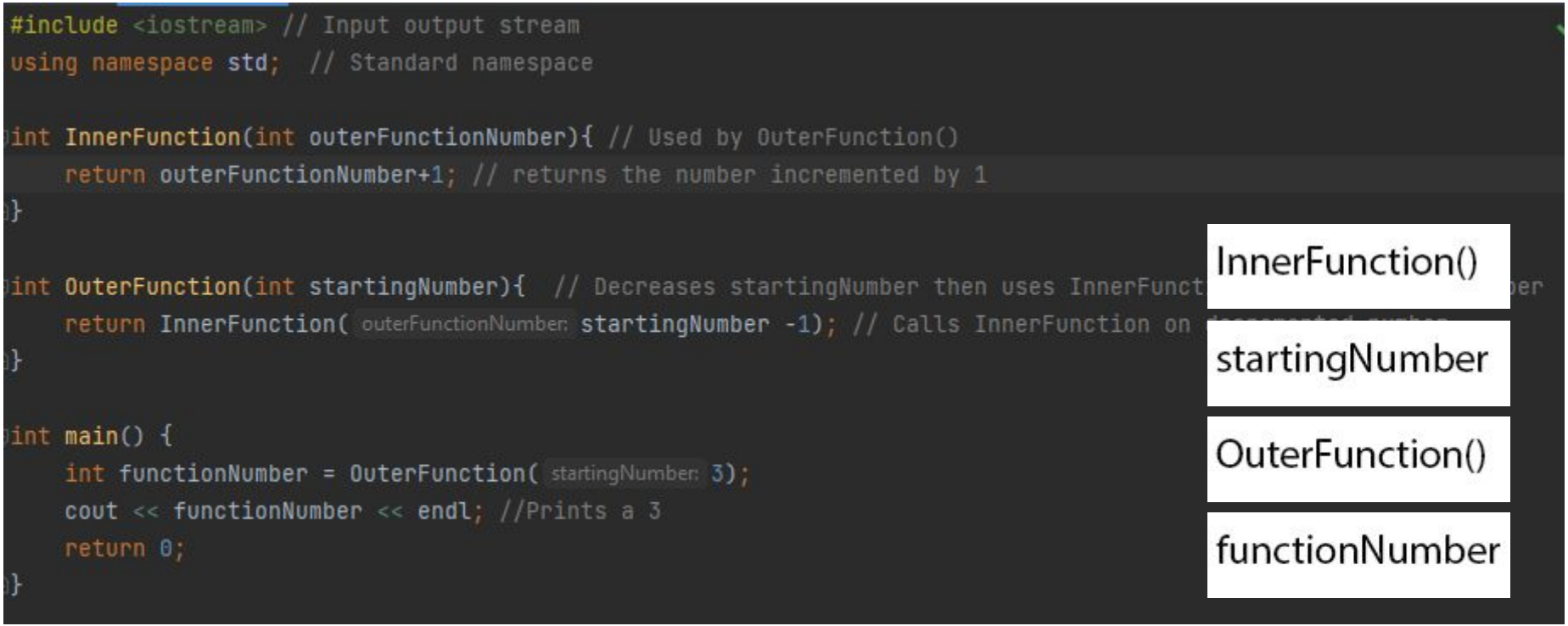
OuterFunction()

functionNumber


#include <iostream> // Input output stream
using namespace std; // Standard namespace
int InnerFunction(int outerFunctionNumber){ // Used by OuterFunction()
return outerFunctionNumber+1; // returns the number incremented by 1
}
int OuterFunction(int startingNumber){ // Decreases startingNumber then uses InnerFunction() to increment number
return InnerFunction(outerFunctionNumber: startingNumber -1); // Calls InnerFunction on
}
int main() {
int functionNumber = OuterFunction(startingNumber: 3);
cout << functionNumber << endl; //Prints a 3
return 0;
}
startingNumber
OuterFunction()
functionNumber


#include <iostream> // Input output stream
using namespace std; // Standard namespace
int InnerFunction(int outerFunctionNumber){ // Used by OuterFunction()
return outerFunctionNumber+1; // returns the number incremented by 1
}
int OuterFunction(int startingNumber){ // Decreases startingNumber then uses InnerFunct
return InnerFunction(outerFunctionNumber: startingNumber -1); // Calls InnerFunction on
}
int main() {
int functionNumber = OuterFunction(startingNumber: 3);
cout << functionNumber << endl; //Prints a 3
return 0;
}
InnerFunction()
startingNumber
OuterFunction()
functionNumber


#include <iostream> // Input output stream
using namespace std; // Standard namespace
int InnerFunction(int outerFunctionNumber){ // Used by OuterFunction()
return outerFunctionNumber+1; // returns the number incremented by 1
}
int OuterFunction(int startingNumber){ // Decreases startingNumber then uses InnerFunct
return InnerFunction(outerFunctionNumber: startingNumber -1); // Calls InnerFunction on
}
int main() {
int functionNumber = OuterFunction(startingNumber: 3);
cout << functionNumber << endl; //Prints a 3
return 0;
}
outerFunction
InnerFunction()
startingNumber
OuterFunction()
functionNumber


#include <iostream> // Input output stream
using namespace std; // Standard namespace
int InnerFunction(int outerFunctionNumber){ // Used by OuterFunction()
return outerFunctionNumber+1; // returns the number incremented by 1
}
int OuterFunction(int startingNumber){ // Decreases startingNumber then uses InnerFunct
return InnerFunction(outerFunctionNumber: startingNumber -1); // Calls InnerFunction on
}
int main() {
int functionNumber = OuterFunction(startingNumber: 3);
cout << functionNumber << endl; //Prints a 3
return 0;
}
InnerFunction()
startingNumber
OuterFunction()
functionNumber

```cpp
#include <iostream> // Input output stream
using namespace std;  // Standard namespace

int InnerFunction(int outerFunctionNumber){ // Used by OuterFunction()
    return outerFunctionNumber+1; // returns the number incremented by 1
}

int OuterFunction(int startingNumber){  // Decreases startingNumber then uses InnerFunction() to increment number
    return InnerFunction( outerFunctionNumber: startingNumber -1); // Calls InnerFunction on decremented number
}

int main() {
    int functionNumber = OuterFunction( startingNumber: 3);
    cout << functionNumber << endl; //Prints a 3
    return 0;
}
```

OuterFunction()

functionNumber

```cpp
#include <iostream> // Input output stream
using namespace std;  // Standard namespace

int InnerFunction(int outerFunctionNumber){ // Used by OuterFunction()
    return outerFunctionNumber+1; // returns the number incremented by 1
}

int OuterFunction(int startingNumber){  // Decreases startingNumber then uses InnerFunction() to increment number
    return InnerFunction( outerFunctionNumber: startingNumber -1); // Calls InnerFunction on decremented number
}

int main() {
    int functionNumber = OuterFunction( startingNumber: 3);
    cout << functionNumber << endl; //Prints a 3
    return 0;
}
```

functionNumber

```cpp
#include <iostream> // Input output stream
using namespace std;  // Standard namespace

int InnerFunction(int outerFunctionNumber){ // Used by OuterFunction()
    return outerFunctionNumber+1; // returns the number incremented by 1
}

int OuterFunction(int startingNumber){  // Decreases startingNumber then uses InnerFunction() to increment number
    return InnerFunction( outerFunctionNumber: startingNumber -1); // Calls InnerFunction on decremented number
}

int main() {
    int functionNumber = OuterFunction( startingNumber: 3);
    cout << functionNumber << endl; //Prints a 3
    return 0;
}
```

Memory

Heap: since we threw stuff into it, we'll need to take it back out to clean it up

Allocate to the heap with "new" or "malloc"

Delete from the heap with "delete" (or "delete[]" for an array) [Advanced: free() will clear memory allocation without calling the destructor -- for this reason, delete must be used with new while free() can be used with malloc]

Memory

Memory leak - failing to clean up memory after dynamic allocation, can become an issue if a program is supposed to run for long periods of time. Eventually the computer will run out of memory

Segmentation fault - trying to access a memory location that the program doesn't have access to, accessing memory used by other programs could corrupt the operating system

Dangling pointers - improperly deleting a pointer

Memory

Assigning and deleting pointers:

```
int main() {
    int newInt = new int;
    int* pointerToInt = newInt;
```

Memory

Assigning and deleting pointers:

```
int main() {
    int newInt = new int;
    int* pointerToInt = newInt;
```

```
auto newInt = new int;
cout << typeid(newInt).name() << endl;
```

Memory

Assigning and deleting pointers:

```
int main() {
    int newInt = new int;
    int* pointerToInt = newInt;
```

```
auto newInt = new int;
cout << typeid(newInt).name() << endl;
```

`Pi` ?

Memory

Assigning and deleting pointers:

```
int main() {
    int newInt = new int;
    int* pointerToInt = newInt;
```

```
auto newInt = new int;
cout << typeid(newInt).name() << endl;
```

Pi ?

```
auto newInt = new char;
cout << typeid(newInt).name() << endl;
```

Memory

Assigning and deleting pointers:

```
int main() {
    int newInt = new int;
    int* pointerToInt = newInt;
```

```
auto newInt = new int;
cout << typeid(newInt).name() << endl;
```

Pi ?

```
auto newInt = new char;
cout << typeid(newInt).name() << endl;
```

Pc

Memory

Assigning and deleting pointers:

```
int main() {
    int newInt = new int;
    int* pointerToInt = newInt;
```

```
auto newInt = new int;
cout << typeid(newInt).name() << endl;
```

Pi ?

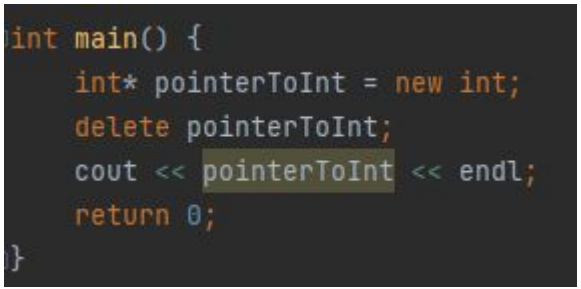
```
auto newInt = new char;
cout << typeid(newInt).name() << endl;
```

Pointers!

Pc

Memory

Assigning and deleting pointers:



Memory

Assigning and deleting pointers:

```
int main() {
    int* pointerToInt = new int;
    delete pointerToInt;
    cout << pointerToInt << endl;
    return 0;
}
```

Returns the address of the new int, pointer is stored in stack memory

Memory

Assigning and deleting pointers:

```
int main() {
    int* pointerToInt = new int;
    delete pointerToInt;
    cout << pointerToInt << endl;
    return 0;
}
```

Returns the address of the new int, pointer is stored in stack memory

Note: we could have used new int(xx) to initialize the int (ie. new int(37))

Memory

```
int main() {
    int* pointerToInt = new int; // Creates a new int and points to it
    delete pointerToInt; // Deletes the new int
    cout << pointerToInt << endl; // Prints the location being pointed to by the pointer
    return 0;
}
```

Memory

Assigning and deleting pointers:

```
int main() {
    int* pointerToInt = new int;
    delete pointerToInt;
    cout << pointerToInt << endl;
    return 0;
}
```

```
C:\Users\sbjsm\CLionProjects\lecture03\cmake-build-debug\lecture03.exe
0x1001770

Process finished with exit code 0
```

Memory

```
int main() {
    int* pointerToInt = new int; // Creates a new int and points to it
    delete pointerToInt; // Deletes the new int
    cout << pointerToInt << endl; // Prints the location being pointed to by the pointer
    *pointerToInt = 4;
    cout << "Made it to the end!" << endl;
    return 0;
}
```

Memory

```
int main() {
    int* pointerToInt = new int; // Creates a new int and points to it
    delete pointerToInt; // Deletes the new int
    cout << pointerToInt << endl; // Prints the location being pointed to by the pointer
    *pointerToInt = 4;
    cout << "Made it to the end!" << endl;
    return 0;
}
```

```
C:\Users\sbjsm\CLionProjects\lecture03\cmake-build-debug\lecture03.exe
0xf41770
Made it to the end!

Process finished with exit code -1073740940 (0xC0000374)
```

Memory

What if we had stored something in that memory location with another program?

Proper way to delete a pointer:

```
int main() {
    int* pointerToInt = new int; // Creates a new int and points to it
    delete pointerToInt; // Deletes the new int
    cout << pointerToInt << endl; // Prints the location being pointed to by the pointer
    pointerToInt = nullptr; // Now we point to nowhere
    cout << "Made it to the end!" << endl;
    return 0;
}
```

```
C:\Users\sbjsm\CLionProjects\lecture03\cmake-build-debug\lecture03.e
0xa36510
Made it to the end!

Process finished with exit code 0
```

Memory

Order is important!

If you point to nullptr before calling delete, there's nothing at the NULL location and it won't delete anything. The int value created in memory will still be there (memory leak!)

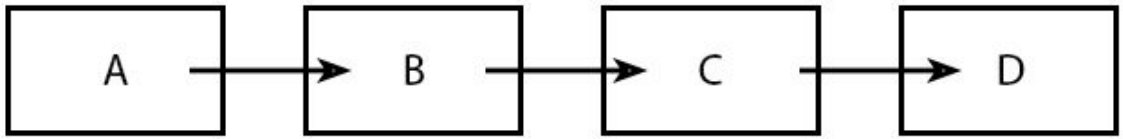
Why does stuff like this matter? Why do we bother with pointers? Data structures!

Linked Lists

What if we want to create a list of objects but we don’t know the size?

Linked Lists

What if we want to create a list of objects but we don't know the size?



Linked Lists

```
struct Node{
    char nodeChar;
    Node* nodePtr;

    Node(char character, Node* nodePointer):nodeChar(character), nodePtr(nodePointer){}
    bool HasNextNode(){return this -> nodePtr; } // Special syntax, if the pointer exists (not nullptr) it returns true

};
```

Linked Lists

```
class LinkedList{
private:
    Node firstNode = Node('a', nullptr);
public:
    LinkedList(Node start): firstNode(start){}
    void PrintList();
};

void LinkedList::PrintList(){
    Node currentNode = firstNode; // is directly equal to (same location in memory for pointers)
    cout << currentNode.nodeChar << endl;
    do{
        currentNode = currentNode.nodePtr;
        cout << currentNode.nodeChar << endl;

    }while(currentNode.HasNextNode());
}
```

Linked Lists

```
int main() {
    Node linkedList = Node( character: 'a',  nodePointer: new Node( character: 'b',  nodePointer: new Node( character: 'c',
                                                                    nodePointer: new Node( character: 'd', nodePointer: nullptr))));
    LinkedList startAtFirstNode = LinkedList(linkedList);
    startAtFirstNode.PrintList();
}
```

```
a
b
c
d
```

Linked Lists

All these “new” Nodes need deleted

```
int main() {
    Node linkedList = Node( character: 'a',  nodePointer: new Node( character: 'b',  nodePointer: new Node( character: 'c',
                                                                    nodePointer: new Node( character: 'd', nodePointer: nullptr)));
    LinkedList startAtFirstNode = LinkedList(linkedList);
    startAtFirstNode.PrintList();
}
```

```
a
b
c
d
```

Linked Lists

All these “new” Nodes need deleted

```
int main() {
    Node linkedList = Node( character: 'a',  nodePointer: new Node( character: 'b',  nodePointer: new Node( character: 'c',
                                                                    nodePointer: new Node( character: 'd', nodePointer: nullptr)));
    LinkedList startAtFirstNode = LinkedList(linkedList);
    startAtFirstNode.PrintList();
}
```

Not the best declaration method for a linked list. A real linked list will have “add”, “remove”, “find”, etc. You’ll be working on this next week!

```
a
b
c
d
```

Linked Lists

Default destructor won't traverse to delete everything

Must overload the deconstructor to handle this…

If overloading any of these three, you MUST overload all three (rule of three):

Destructor

Copy constructor

Copy assignment

Linked Lists

```
class LinkedList{
private:
    Node firstNode = Node('a', nullptr);
public:
    LinkedList(Node start): firstNode(start){}
    void PrintList();
    ~LinkedList(){ /* Code that deletes all Nodes */}   // Destructor
    LinkedList(LinkedList& copyFrom){ /* Code to copy each Node */} // Copy Constructor
    LinkedList& operator=(const LinkedList& copyFrom){ /* Code to copy each Node */} // Copy Assignment
};

void LinkedList::PrintList(){
    Node currentNode = firstNode; // is directly equal to (same location in memory for pointers)
    cout << currentNode.nodeChar << endl;
    do{
        currentNode = currentNode.nodePtr;
        cout << currentNode.nodeChar << endl;

    }while(currentNode.HasNextNode());
}
```

Overloading

Overloading is the process of replacing a previously defined function/operation with another version with the same function name or operator symbol.

With functions, overloading is done by providing different parameter types (ie. print(char thisChar) vs. print(int thisInt))

Operator overloading is similar, but operators have a default definition that is trivial in most cases. For example, the '=' operator by default will perform 'memmove' on any object type. This re-casts the object to a character array and copies it to the destination. Not a valid copy assignment for almost all custom classes.

Similarly, the default destructor operator (~) will simply call the destructor for each field of the class. Pointers do not have a default destructor which is why we must manually overload the destructor for Linked Lists.

Overloading

Example:

```
struct Node{
    char nodeChar;
    Node* nodePtr;

    Node(char character, Node* nodePointer):nodeChar(character), nodePtr(nodePointer){}
    bool HasNextNode(){return this -> nodePtr; } // Special syntax, if the pointer exists (not nullptr) it returns true
    ~Node()   // Destructor
    {
        delete nodePtr;
        nodePtr = nullPtr;
    }
    Node(Node& copyFrom) // Copy Constructor
    {
        nodeChar = copyFrom.nodeChar;
        nodePtr = new Node(copyFrom.nodeChar,  nodePointer: nullptr); // Does not copy the node after the next node!!
    }
    Node& operator=(const Node& copyFrom) // Copy Assignment
    {
        nodeChar = copyFrom.nodeChar;
        nodePtr = new Node(copyFrom.nodeChar,  nodePointer: nullptr); // Does not copy the node after the next node!!
    }
};
```

Announcements

Lab 00 is due tomorrow! I'll be releasing Lab01 (and all subsequent labs) earlier in the day on Wednesdays to allow for 2 full days of working on it before section.

Quiz 1 is this Friday, will be open for a 12 hour time window: 9am-9pm PST (BUT MUST BE COMPLETED WITHIN 30 MINUTES OF STARTING)

This weeks lab allows for partners. I recommend trying to find a compatible partner through Piazza before section. There's a "looking for partners" section where you can post an introduction of yourself

Office Hours are Wednesdays BY APPOINTMENT, please schedule them through the Gauchospace link

Up next...

Binary search trees