Introduction to C++

→ Introduced in 1985.... many improvements since then!

→ Still in top 2 or 3 programming languages!

→ Engineers (IEEE rankings)

1. Python
2. C
3. C++
4. Javascript
5. SQL

C → C++

Add true object-oriented capability.

fix things that are cumbersome / too verbose in C

Improvement 1 : Input / Output (Formatting)

```
#include <iostream>
int main() {
    std::cout << "Hello World!" <<
                              endl;
}
```

Namespaces:

a way to (a) make syntax easier

(b) avoid library collisions (same function name in multiple libraries)

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!" << endl;
}
```

Output formatting:

```
#include <iomanip>
#include <iostream>
using namespace std;
int main() {
        double x;
        cin >> x;
        cout << fixed << setprecision(2);
        cout << x << endl;
}
```

Summary of New Things for Ch. 10/11

① use of cin / cout (iostream)

cout << ......

cin >> .....

(i) send things to cout

(ii) send things from cin

② format qualifiers (fixed, set precision, etc.)

\# include <iomanip>

③ math functions → # include <cmath>

④

```
int num1;
int num2;
double x;

x = static_cast<double>(num1) *
                         num2
```

Vectors in C++ (Ch. 12)

In C, we saw primitive types, arrays, pointers, and structs.

C++ introduces a new complex type, called a standard vector. Basically, it is a dynamic array. One can add /subtract to the length of the array, dynamically. Sort of like a Python list. But, vectors are of a single type.

Example:

```
#include <cstdlib>
#include <iostream>
#include <vector>

int main () {
    vector<int> userInts;
    userInts.push_back(3);
    userInts.push_back(5);
    userInts.push_back(7);

    return 0;
}
```

create an empty vector of ints!

add elements to the vector

Summary of Useful Vector Methods

Add to end of vector → • push_back(—)

• at (index)

get element at index ↗

• size ( ) ← returns # of elements, currently.

Longer Example: (Zylabs 12.21)

1. get # of values from user
2. get values from user (doubles)
3. find the max. value.
4. normalize all values to max. value.
5. print out normalized values.

```
# include <iostream>
# include <iomanip>
# include <vector>

using namespace std;

int main() {
        vector<double> userValues;
        double numValues;
        double currValue;
        double maxValue;
        int i;
        unsigned int j;

        cin >> numValues;

      for (i=0; i<numValues; i++) {
              cin << currValue;
              userValues.push_back(currValue);

      }

      maxValue = userValues.at(0);
      for (j=0; j<userValues.size(); j++) {
            if (userValues.at(j) > maxValue) {
                  maxValue = userValues.at(j);
            }
      }
```

```
    cout << fixed << setprecision(2);

    for (j=0; j < userValues.size(); j++){
        cout << userValues.at(i)/maxValue
                              << " ";
    }
    cout << endl;

}
```

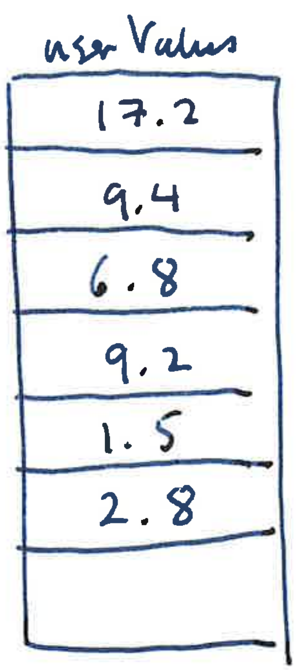
A better way to loop over a vector!

→ almost always, we tend to loop over all elements of a vector, in order. I.E. we iterate over the elements.

→ There is a special type of object in C++, called an iterator, that makes this super fast, super robust, and super easy!

Create → iterator object for a vector of doubles

```
vector<double>::iterator ptr;

for (ptr = userValues.begin();
                ptr < userValues.end();
                        ptr++) {

        .... do things ...
        *ptr   dereferences
               to get value!!
}
```



← userValues.begin()
(first element!)

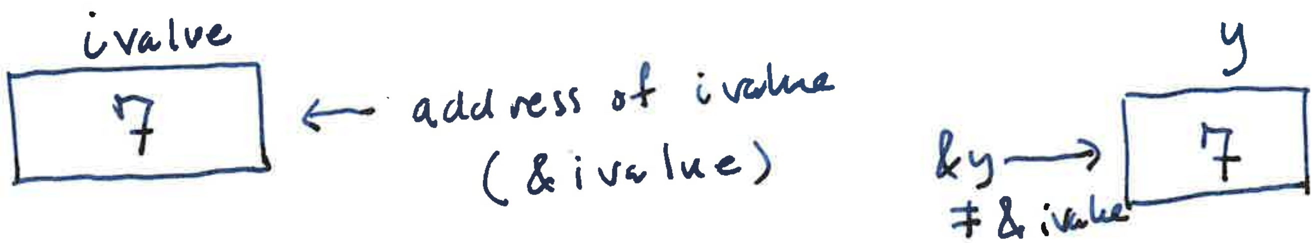
← userValues.end()
(last element + 1)
≡ !!!!

Passing by Reference (Ch. 13)

When we studied user functions in C, we spent time talking about passing values of variables vs. passing pointers to variables.

In C++, we are going to add another way, which is called passing by reference.

References are like aliases.




```
void passBy (int y) {
        y = y + 4;
        return
}

int main () {
        int ivalue = 7;
        passBy (ivalue);
}
```

y is a copy of ivalue.

7

cout << ivalue;

If we pass, instead, a <u>pointer</u>:

```
void passBy( int* py) {
        *py = *py + 4;
        return;
}

int main()  {
        int ivalue = 7;
        passBy( &ivalue);
        cout << ivalue;
}
```

(&ivalue $\boxed{\cancel{7}\ 11}$)

$\Rightarrow$ 11

(ivalue $\boxed{7}$ $\leftarrow$ &ivalue)

So, how do we pass by reference?

```
void passBy ( int& ry ) {
    ry = ry + 4;
    return
}

int main () {
    int ivalue = 7;
    passBy (ivalue)
    cout << ivalue;
}
```

ry

$\boxed{\cancel{7}\ 11}$ ← &ry

Same address.

ivalue

$\boxed{\cancel{7}\ 11}$ ← &ivalue

Conclusion: We can switch between pass by reference and pass by value by changing only the function definition ⟹ we don't have to change main!!

Either way, we call the function with just the variable name!

You will find detailed examples of passing arrays & vectors in the project called Pass By.

Summary

Function	Result
passByValue(int y)	- passes a copy - value in main unchanged
passByPtr(int* y)	- passes a pointer - value in main changes
passByRef(int& y)	- passes a reference - value in main changes
passArrayByValue(int a[], int length)	- passes a copy - values in main change - weird !!!!
passArrayByPtr(int* a, int length)	- passes a pointer - values in main change
passArrayByRef(int& a, int length)	⇐ NOT ALLOWED!
passVectorByValue(vector<int> q)	→ passes a copy → values in main unchanged
passVectorByRef(vector<int>& q)	→ passes a reference → values in main change
~~passVector~~ passConstVectorByRef(const vector<int>& q)	→ passes by reference, but can't change

Objects, and Object-Oriented Programming, in C++ (Ch. 14)

In C, we already saw some initial aspects of object-oriented concepts, with the use of structs. C++ extends this, in a much more complete way!!

Object-Oriented Concept 1 : Encapsulation.

→ hide the internal variables of the complex object from the user!!

GOOD → makes code easier to maintain.

"BAD" → makes code harder to write, and makes design a premium!!

In C++, ~~objects~~ the concept of objects are realized through the introduction of classes. A class is a template for how to make objects. It's like the DNA of C++!!

Example: We want to develop an app for a restaurant rating system (like YELP!)

(i) we will want to store data for many restaurants. Each restaurant will be represented by a different data object. In C, we might do something like:

```
typedef struct Restaurant_struct {
        char name[20];
        int rating;
        char price[5];
        char cuisine[30];
        int id
} Restaurant;

Restaurant moes;
Restaurant schooners;
Restaurant mickeydees;
```

This would create 3 restaurant objects. Of course, we would have to also write an initialization method, setter/getter methods, and other complex methods.

In C++, the same functionality is acheived, plus much, much, much more, by using <u>classes</u>.

<u>Restaurant.h</u>

```
class Restaurant {

    private:
        string name;
        int rating;
        string price;
        string cuisine;
        int id;
    public:
        Restaurant();
        void SetName(string myName);
        void SetRating(int myRating);
        void SetPrice(string myPrice);
        void SetCuisine(string myCuisine);
        void SetID(int id);
        string GetName() const;
        int GetRating() const;
        string GetPrice() const;
        string GetCuisine() const;
        int GetID() const;
        void Print() const;
}
```

Constructor

Notes: (i) there is now an explicit notation of private internal variables, and public methods!!

(ii) For the public methods, we can further specify that the method cannot modify internal variables, by adding const after the method prototype!!

Both of these things lead to better encapsulation of Restaurant objects.

(iii) As usual, we will have to provide the code for all of these public methods in a separate file, called Restaurant.cpp, BUT:

For getter methods, it is usual to write these inline; right in the Restaurant.h header file.

For example:

```
string GetName() const {return name};
```

(iv) Constructors

When using structs, we talked about providing a method to initialize structs. (InitCar())

In C++, this process is streamlined, expanded, and in fact is required, for every class template!!!

There must exist, at the minimum, a constructor for objects of the class.

```
Restaurant();
```

→ default constructor of the restaurant class.

In addition, if we desire, we can also provide additional initialization constructor methods.

Example: Restaurant.h →

```
class Restaurant {
    private:
        ⋮

    public:
        Restaurant();
        Restaurant(string userName,
            int userRating, string userPrice,
                string userCuisine);
        ⋮

}
```

Restaurant.cpp →

```
Restaurant::Restaurant() {
    name = "No Name";
    rating = -1;
    price = "No Price";
    cuisine = "No Cuisine";
    id = Φ
}
```

Restaurant.cpp →

```
Restaurant::Restaurant(string userName,
        int userRating, string userCuisine,
            string userPrice){
    name = userName;
    price = userPrice;
    rating = userRating;
    cuisine = userCuisine;
    id = 0;

}
```

This is an example of another super-cool C++ feature → <u>overloading</u>

We can write <u>two</u> functions, with the <u>same name</u>, but different #'s of arguments. The compiler/executable will choose the correct one based on how we call it!

calls default → Restaurant moes;

calls initialization → Restaurant schooners("schooner's", 5, "$$", "American");

<u>Note</u>: Because this is so common, the usual way is to <u>combine</u> the default and initialization constructors together;

> See BasicObjects project for an example of how this is done!

Relationships between objects of the same class.

You might imagine that it would be desirable to have the internal id # of our restaurant objects be:

(a) auto-generated #

(b) sequential.

e.g.

Restaurant moes; ← id = 1001

Restaurant shooners; ← id = 1002

Restaurant muchadees; ← id = 1003

⋮

There is a cool way to do this in C++ classes!! Add an extra private variable, nextID:

```
private:
    ⋮
    static int nextID
```

When we declare an internal variable as static, it means that it is a variable of the class, and not a particular object of the class. Think of it like a global variable of the class. All objects of the class have access to it, and it is a single value for all objects.

We also need to provide an initialization method for each static variable.

Restaurant.cpp →

```
int   Restaurant::nextID = 1001;
```

Then, all we have to do, in our constructors, is set:

```
id = nextID;
nextID++;
```

Now, each new object will get a unique, sequential ID number!!

Pointers in C++ (Ch. 15)

→ we have already seen pointers in C, and have used pointers in C++ up to this point, in much the same way.

→ what about pointers to objects??

Example:

```
Class Point {                     public member
                                  variables (not usual)
public:
        double X;
        double Y;                 constructor
        Point (double xValue = 0, double YValue=0);
        void Print();             simple print method.
}

Point::Point (double xValue, double yValue) {
        X = XValue;
        Y = yValue;
}
Point::Print()  {
        cout << "(" << X << "," << Y << ")" << endl;
}
```

```
int main () {

    Point* pp1 = new Point;
```

⇨ defines a pointer to an object of type point, and calls the <u>default</u> constructor.

```
    (*pp1).Print();
```

⇨ dereferences pp1 (which will be the actual object), and prints that object (expect (0, 0)

```
    Point* pp2 = new Point(8,9);
```

⇨ pointer to a new, different object of type point. Initialize with X=8, Y=9.

```
    (*pp2).Print;
```

⇨ expect (8, 9)

```
    pp1 -> Print();
```

⇨ we use the "->" symbol for pointers to objects!!

```
delete pp1;
delete pp2;
```

⇒ free up memory allocated to these objects.

```
pp1 -> print();
pp2 -> print();
```

Random behavior!!!

Eeek!!

```
}
```

---

what about ~~[illegible]~~ <u>vectors</u> of <u>pointers</u> to <u>objects</u>?

Memory Management in C++ : Static Memory, the Heap, and the Stack

Four different regions:

(1) Code memory: where the program instructions are stored.
(we can't access this, typically)

(2) Static Memory: global variables, and static local variables.
→ allocated once and stay there for the entire program execution.

dynamic memory

(3) The Stack: place to store local function variables. The OS takes care of allocation and deallocation for us.
→ automatic

(4) The Heap: new → allocates
delete → de-allocates

Let's look at a simple program to illustrate this:

```
#include <cstdlib>
#include <iostream>
using namespace std;
                                        ⇐ Static memory
int myGlobal = 33;

int main() {
                                        ⇐ Stack
    int myInt;
    int* myPtr = nullptr;               ⇐ Stack, for now!
    myInt = 555;
                                        ⇐ Heap!!
    myPtr = new int;
    *myPtr = 222;

    cout << *myPtr << myInt << endl;
    delete myPtr;                       ⇐ Removes from Heap!!

    myFunction();

    return 0;                           ⇐ removes myInt from stack
}
```

```
void myFunction ( ) {
        int myLocal;          <= Stack
        ~~[illegible]~~ myLocal = 999;
        return;               <= removes from
                                 stack.
}
```

See Project StackHeap on GitHub for code, and further Documentation.

Memory Allocation & DeAllocation of Objects in C++

→ we have seen how basic memory management works with objects in C++, using new, delete, and also how we can use pointers to objects in C++, and access member variables using the "→" operator.

→ another aspect of this topic is how to build memory management into the class itself.

→ We've seen how we can create new objects, using constructors.

→ How do we handle the deletion? We use Destructors.

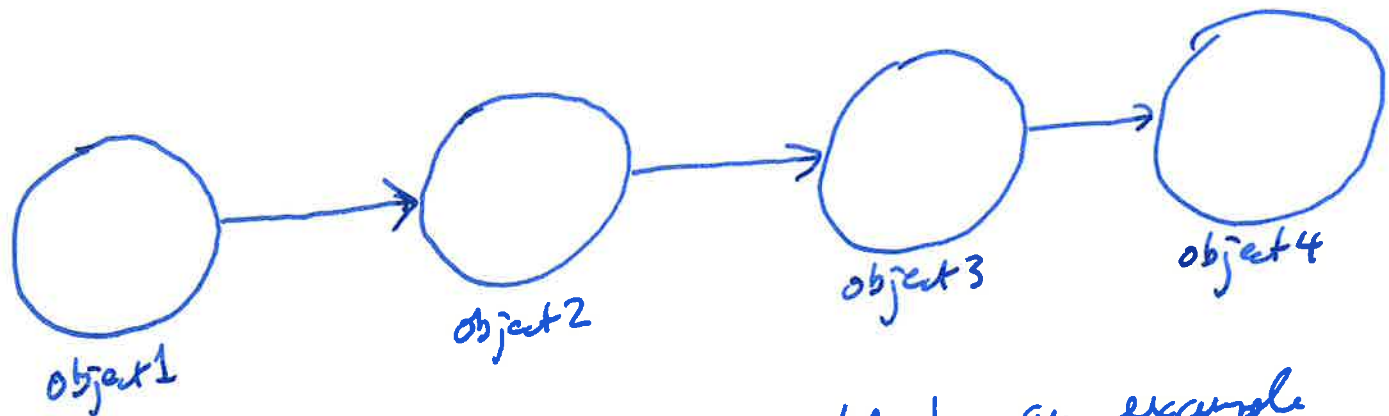
→ How do we set one object "equal" to another object? → Copy constructors.

→ To understand this, we are going to take a bit of a detour!

Linked Lists

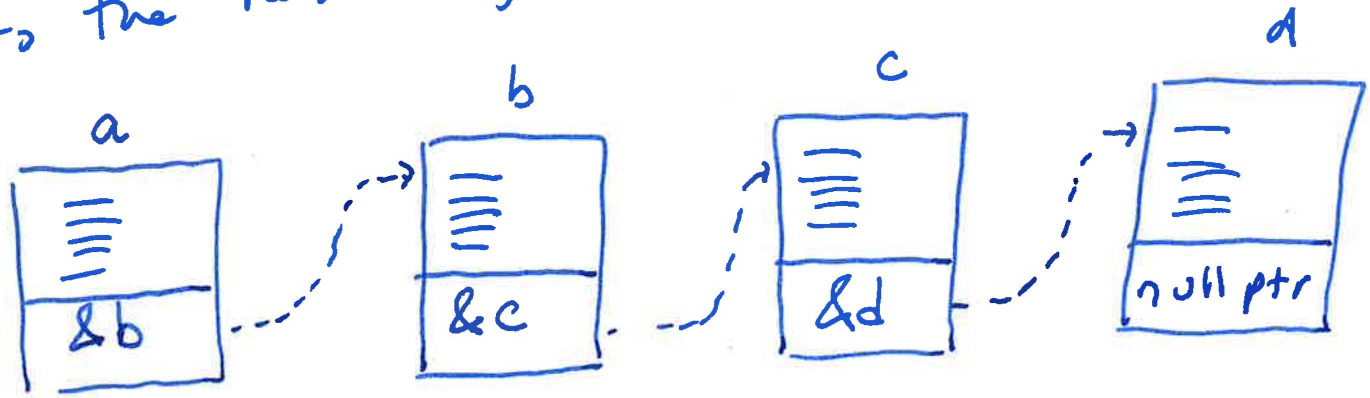
→ big topic in the study of data structures.

→ a linked list is a series of objects that are somehow "linked" to one another, sequentially.



A C++ vector of objects would be an example of a linked list.

What is the "link"? Typically, it is a pointer to the next object in the list!!



Each object in a linked list is called a node.
The first object in the list is the head node.
The last object in the list has a null pointer for the address of the next node.

↳ we know when we get to the end of a linked list, simply by looking for a null ptr!

↳ Creating a linked list involves creating the head node, only.

(IntNode.h)

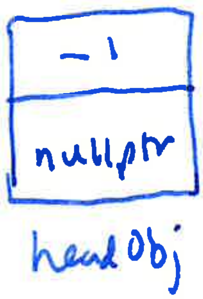
```
class IntNode {
  public:
    IntNode (int dataInit=0, IntNode*
                        nextLoc = nullptr);

    void InsertAfter (IntNode* nodeLoc);
    IntNode* GetNext( );
    void PrintNodeData();
  private:
    int dataVal;
    IntNode* nextNodePtr;
}
```

IntNode.cpp → need code for the initialization constructor, InsertAfter, GetNext, and PrintNodeData

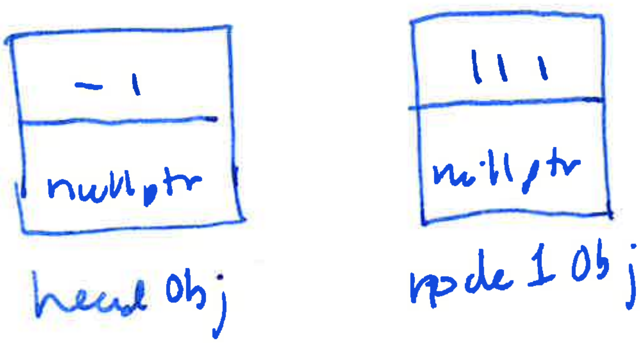
```
IntNode::IntNode(int dataInit, IntNode* nextLoc){
        this->dataVal = dataInit;
        this->nextNodePtr = nextLoc;

}

void IntNode::PrintNodeData() {
        cout << this->dataVal << endl;

}

IntNode*    IntNode::GetNext() {
        return this->nextNodePtr;

}

void IntNode::InsertAfter(IntNode* nodeLoc) {
        IntNode*   temp = nullptr;              1
        temp = this->nextNotePtr;               2
        this->nextNodePtr = nodeLoc;            3
        nodeLoc->nextNodePtr = temp;            4

}
```

main.c

```
IntNode* headObj = new IntNode(-1);
```

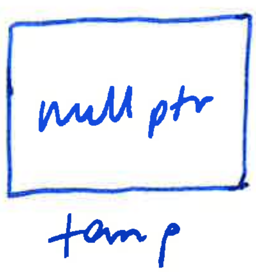



```
IntNode* node1Obj = new IntNode(111);
```

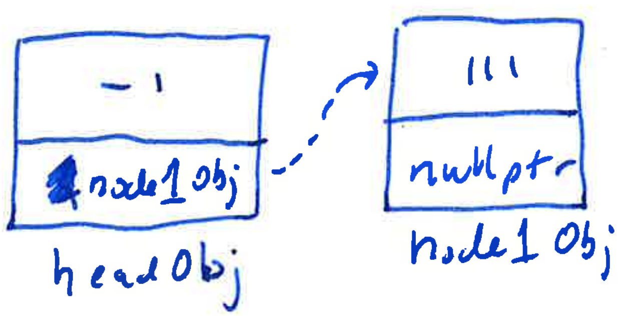


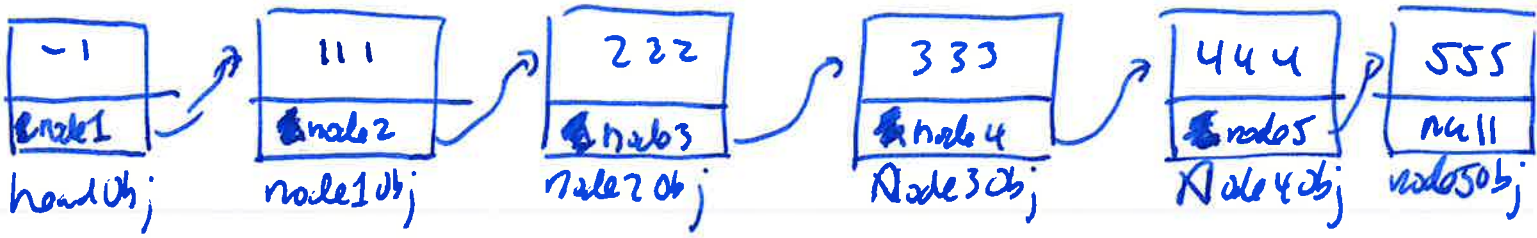

```
headObj → InsertAfter(Node1Obj);
```

Step 1/2:



Step 3/4

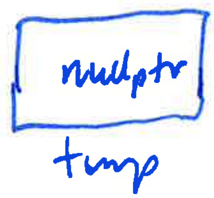




IntNode* node6obj = new IntNode(666);

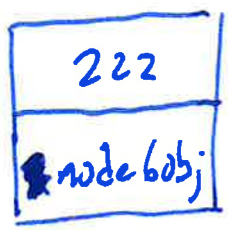
node2obj -> InsertAfter(node6obj);

Step 1

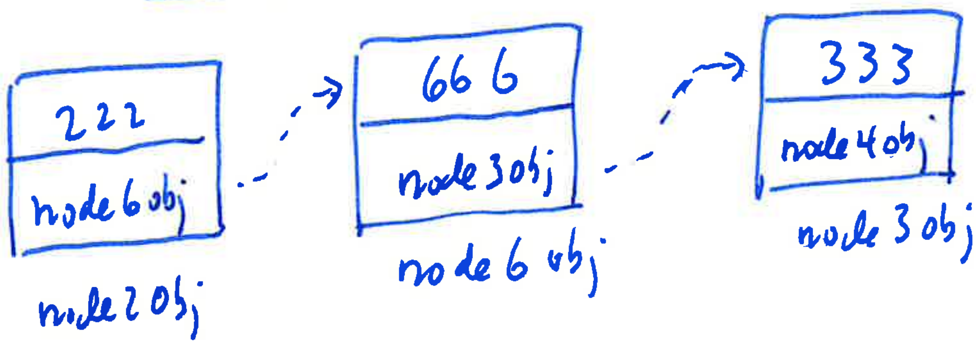


Step2 :

Step 3 :



Step4 :



BIG HUGE MASSIVE SUPER-IMPORTANT QUESTION:

What if we want to remove an element from a linked list????? This is not such an easy thing, and requires care and caution. It's probably one of the things the most often that is not handled properly in C++ programming!!

Destructors:

We add a destructor to the class

```
class IntNode{
    public:
        IntNode(int dataInit=0, IntNode* nextLoc
                                   = nullptr);

destructor ⇒ ~IntNode();
        ⋮
    private:
        ⋮
}
```

A More Useful Linked List, and a Better Example of Destructors.

Let's separate the tasks of creating/destroying nodes from the tasks of maintaining the actual linked list of nodes!!

Intnode.h } nodes
Intnod.cpp

LinkedList.h } the list
LinkedList.cpp

```
Class IntNode {
    private:
        int data;
        IntNode* next;
    public:
        IntNode (int dataValue);          <= wrt. a node.
        ~IntNode ( );                     <= destructor of a node
        void SetData (int dataValue);     } Setter
        int GetData ( ) const;            } Getter
        void SetNext (IntNode* nextPtr);
        IntNode * GetNext ( ) const;
```

```
class LinkedList {

    private:
        IntNode*  head;

    public:
        LinkedList( );
        ~LinkedList ( );
        IntNode*  GetHead( ) const;
        void  SetHead ( IntNode*  headPtr);

        void  Prepend (int dataValue);

}
```

← linked list defined by only the location of the head node!!

Add a node in front of the head node, and make this new node the head node!!

(LIFO → last in, first out buffer)

→ most of the methods of these two classes are totally straight forward, and very similar to what we have seen before.

→ There are two methods (in LinkedList.cpp) which we need to look at.

(1) LinkedList :: Prepend (int dataValue) {

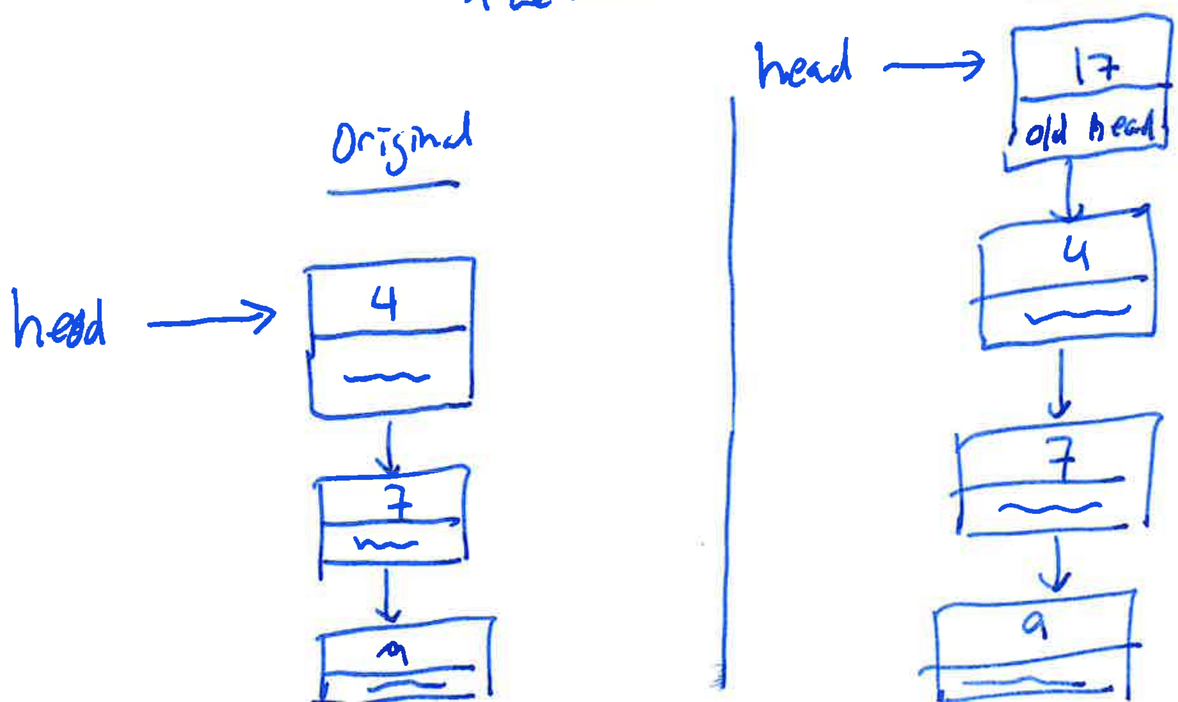
create new node. ⇒ IntNode* newNode = new IntNode(dataValue);

set its nextPtr to the current head node. ⇒ newNode → SetNext (head);

head = newNode;

}

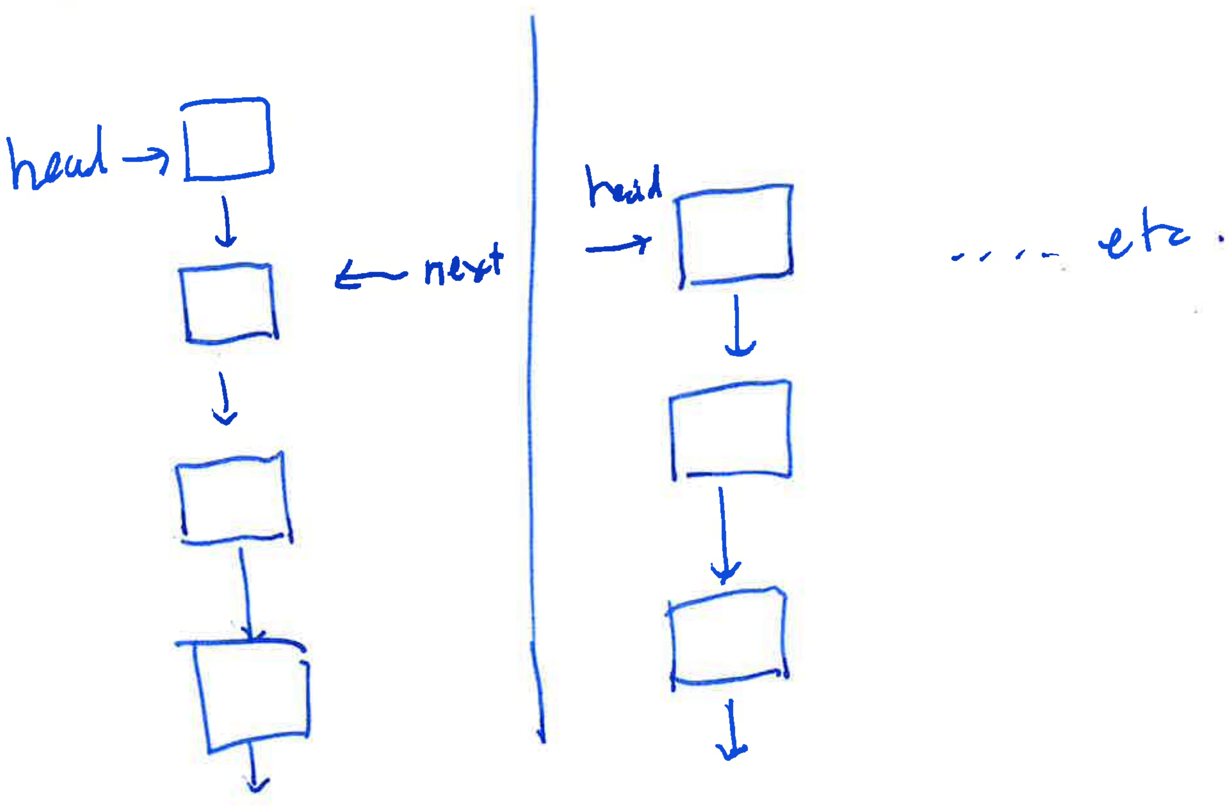
⇗ make the new node the head node.



② LinkedList :: ~LinkedList ( ) {

```
    while (head) {
        IntNode* next = head -> GetNext();
        delete head;
        head = next;
    }
}
```

What does this do? Start at the head node. Set next = "the next node". Delete the head node. Make next the new head node. Keep going until all nodes are deleted.



Copy Constructors

→ when we pass objects to functions, a copy of that object is made. Then the function acts on the copy.

→ BUT: if there are member variables that are pointers, and we manipulate those pointers in the function, or delete those pointers, we can get into trouble.

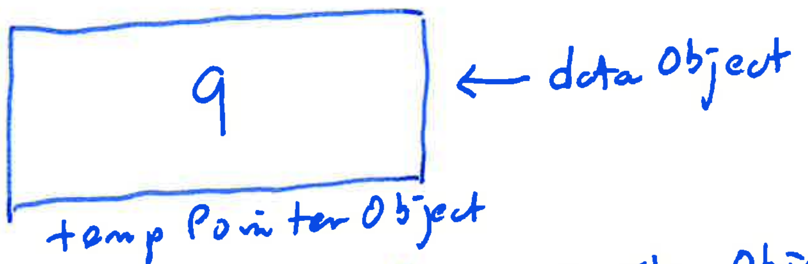
→ Solution: Provide a copy constructor
→ set of rules for how to make copies of your objects.

```
class MyClassInt {
   private:
    int dataObject                                   ⇐ single int member variable.
   public:
    MyClassInt() {                          ⇐ constructor
        dataObject=0;
    }
    void SetDataObject(int i) {dataObject=i;}       ⇐ Setter/Getter.
    int GetDataObject(){return dataObject;}
}
```

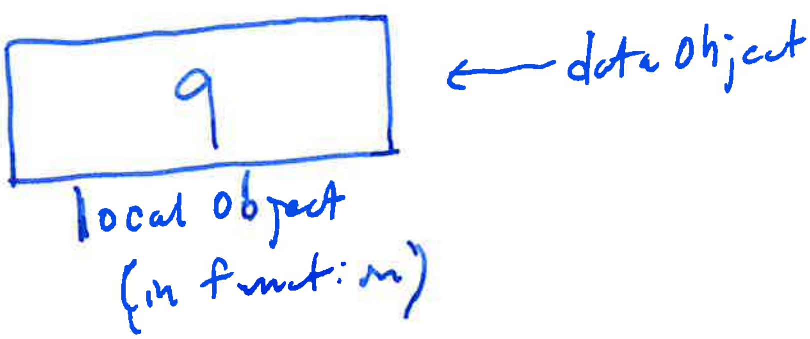
```
Class MyClassIntPointer {
    private:
        int* dataObject

    public:
        MyClassIntPointer () {
            dataObject = new int;
            *dataObject = 0;
        }                                   constructor

        ~MyClassIntPointer() {
            delete dataObject;
        }                                   destructor

        MyClassIntPointer (const MyClassIntPointer&
                                  origObject) {

            dataObject = new int;
            *dataObject = *(origObject.dataObject);

        }
copy
constructors
        MyClassIntPointer&   operator= (const MyClassIntPointer&
                                          objToCopy) {
            if ( this != &objToCopy) {
                    delete dataObject;
                    dataObject = new int;
                    *dataObject = *(objToCopy.dataObject);
            }
            return *this
        }
```

So, what happens when we pass an object of the MyClassIntPointer class to a function?

MyClassIntPointer tempPointerObject;
tempPointerObject.SetDataObject(9);



SomePointerFunction(tempPointerObject);



copy made automatically.

The problem is that when we come back from the function, local object is deleted, and this deletes dataObject pointer!! This is bad!!