Today’s Topics:
• More on constructor/destructor functions
• More on STL <list>
Recall: Constructor and destructor functions

- What are they?
  - Remember example:

```cpp
class Bert {
    public:
    Bert(char *);
    ~Bert();
    void add_bert();
    string getString();
    int getLength();
    protected:
    string str;
}

Bert::Bert(char *s)
{ printf("Creating new Bert (%s)\n", s);
  str = s;  }

Bert::~Bert()
{ printf("Bert destructor called (%s)\n", str.c_str()); }
```
Important details to remember about a constructor function

1. Automatically invoked when an object is created
2. Have same name as class name
3. May be overloaded (i.e., there may be multiple constructors with different signatures)
4. Must not specify a return type or explicitly return a value
5. Example:

```cpp
class Stack {
    protected:
        int * data;
        int top;
    public:
        Stack( int maxSize );
        Stack();
        ...
};

Stack::Stack(int maxSize) {
    top = 0;
    data = new int[maxSize];
}

Stack::Stack( ) {
    top = 0;
    data = new int[20];
}
```

6. If you don’t specify constructor function, C++ will create an implicit one for you that takes no arguments and does nothing.
Important details to remember about a destructor function

1. Automatically invoked when an object is destroyed
2. Has same name as the class, preceded by ~
3. Takes no arguments → can’t be overloaded (i.e., only one destructor per class)
4. Must not specify a return type or explicitly return a value
5. Example: `Stack::~Stack() { delete [] data; }`
6. Does not deallocate memory for its own object. But, it should delete any memory that constructor explicitly allocated for member objects using `new` operator
7. If you don’t specify destructor function, C++ will create an implicit one for you that takes no arguments and does nothing.
Interesting question:
Can you call constructor and destructor independently?

You can call a destructor directly:

```cpp
#include <stdio.h>
#include <string>
using namespace std;

class Bert {
public:
    Bert(int val);
    ~Bert();
    int getValue();
    int value;
};

Bert::Bert(int val)
{ value = val;
    printf("Creating Bert, %d\n", val);
}

Bert::~Bert()
{ printf("Bert destructor called, %d\n", value);
    value = 10;
}

main()
{ Bert *s1;
    s1 = new Bert(5);
    s1->~Bert();
    delete s1;
}
```

UNIX> test
Creating Bert, 5
Bert Destructor called, 5

But memory isn’t freed by calling destructor directly – it just does what the method instructs.

```cpp
... main()
{ Bert *s1;
    s1 = new Bert(5);
    s1->~Bert();
    delete s1;
}
```

UNIX> test2
Creating Bert, 5
Bert Destructor called, 5
Bert Destructor called, 10

UNIX> test
Creating Bert, 5
Bert Destructor called, 5
Important details to remember about creating and destroying objects

1. You can statically allocate an object by just declaring it:

```cpp
Stack a(5);
Stack b;
Stack c(a); // copy constructor
```

2. A statically allocated object can’t be explicitly destroyed. It will be destroyed automatically when the block or procedure to which the object belongs goes out of scope.

3. An object can be dynamically allocated using `new`

```cpp
Stack *x = new Stack(4);
Stack *y;
Stack *z = new Stack[5]; // allocate an array of 5 stacks
y = new Stack(*x);
```

4. A dynamically allocated object can be freed using `delete`

```cpp
delete x;
delete [] z;
```

5. If an array is allocated using `new`, it must be deleted by placing `[]` after delete command. Otherwise, destructor will be called only on the first object.
Two special types of constructors

- **Copy constructor**
  - Called when constructing a new object initialized to a copy of the same type of object

- **Assignment operator (operator=)**
  - Called when = is applied to two objects after they have been previously constructed
  - “lhs = rhs” copies state of rhs into lhs
Copy constructor

- Copy constructor copies contents of one object to another object. The two objects must be members of the same class.

- Copy constructor called when:
  - One class object is initialized with another class object. For example:
    ```
    // Uses copy constructor
    string a;
    string c(a);  // or string c = a;
    ```

  - A class object is passed by value as an argument to a function
    ```
    // NOTE: Uses assignment operator
    string a;
    string c;
    c = a;  // ✅ not copy constructor
    ```

  - A class object is returned as the return value from a function
Detour: C++ Parameter Passing (3 ways)

1. Call by constant reference
   - Value of argument cannot be changed
   - Use for class type or large objects that shouldn’t be altered

2. Call by value
   - Value of argument cannot be changed
   - Use for small objects (primitive types)

3. Call by reference
   - Value of argument can be changed

```cpp
double avg(const vector<int> & arr, int n, bool & errorFlag);
```
Detour: C++ Parameter Passing Example

```cpp
string::string(const string &objToBeCopied);
```

- This method is a copy constructor

- Note declaration of `objToBeCopied`:
  - declared to be constant reference of type string
    - contents can’t be changed by the method
  - A reference type is defined by following type specifier with an “address-of” (&) operator
  - Similar to pointer. Except, when you refer to the contents of the object pointed to by the reference type, you use dot (.) operator instead of arrow (->) operator.

- Why does C++ confuse everything by introducing this second type of pointer? Subtle reasons, not important here.
Detour: C++ Return Passing

1. Return by value (use for primitive return types)
2. Return by reference [avoid]
3. Return by constant reference (use for complex return types; avoids overhead of a copy)

- Be careful: expression in return statement has to have valid scoping beyond the return of the function

**Which example below is correct?**

**Correct**

```cpp
class Person {
public:
    Person(const string & name) : name(name) {} 
    const string & getName() const { return name; }
private:
    string name;
};
```

**Wrong:** maxValue does not have scope beyond this function

```cpp
class Person {
public:
    Person(const string & name) : name(name) {} 
    string getName() { return name; }
private:
    string name;
};
```
Important details to remember about copy constructors

Definition: Copy constructor copies contents of one object to another object. The two objects must be members of the same class

1. Copy constructor called when:
   - One class object is initialized with another class object.
   - A class object is passed by value as an argument to a function
   - A class object is returned as the return value from a function

2. Form of copy constructor definition (argument is object to be copied):

   `classname (const classname&)`
Important details to remember about **copy constructors** (con’t.)

3. Copy constructor copies values of each of the parameter object’s data members to each of the class’s data members

4. IMPORTANT: If a data member is a pointer, the copy constructor should not copy the pointer. Instead, it should:
   - Allocate a new block of memory large enough to hold contents of object being pointed to by the pointer
   - Copy the object to the newly allocated memory
   - Make the data member point to the new block of memory
Important details to remember about **copy constructors** (con’t.)

5. **IMPORTANT**: If your class contains pointers to dynamically allocated memory, you must define a copy constructor for that class that adheres to the convention of allocating memory for the object pointed to by a pointer and copying the object to that memory.

- If you fail to define a copy constructor, C++ will define a default copy constructor that will do a bit-wise copy of all the data members (including pointers)
Example of defining a copy constructor:

class IntCell {
    public:
        explicit IntCell (int initialValue = 0);
        IntCell (const IntCell & rhs);
        ~IntCell();
        const IntCell & operator= (const IntCell & rhs);
        int read() const;
        void write (int x);
    private:
        int *storedValue;
    }

IntCell::IntCell(int initialValue)
{  storedValue = new int (initialValue); }

IntCell::IntCell(const IntCell & rhs)
{  storedValue = new int (*rhs.storedValue);  }

// DEFAULT COPY CONSTRUCTOR WON'T WORK!!
IntCell::IntCell(const IntCell & rhs)
{  storedValue (rhs.storedValue);  } // copies pointer instead of pointee

Key reason why default copy constructor won't work:
Data members that are pointers aren't handled correctly
Diversion:
Copy constructors for robots?  😊

- Self-replicating robots (work at Cornell, reported last year in *Nature*)

http://mae2.wdg.us/ccsl/research/selfrep/morepictures.htm
Operator methods (e.g., operator=, operator+=, etc.)

- C++ allows most of its operators (e.g., =, ==, +, -, ++, ->) to be redefined by a class.

- This redefinition is called *overloading*, because the operator has different definitions depending on which object it is operating on.

- Operators should take care to release memory if they are replacing one piece of dynamically allocated memory with another.
Assignment Operator (operator=)

- Recall: assignment operator is called when "=" is applied to two objects, after they have both been previously constructed.

- By default, operator= is implemented by applying operator= to each data member in turn.
Example of operator=

class IntCell {
    public:
        explicit IntCell (int initialValue = 0);
        IntCell (const IntCell & rhs);
        ~IntCell();
        const IntCell & operator= (const IntCell & rhs);
            int read() const;
            void write (int x);
    private:
        int *storedValue;
};

const IntCell & intCell::operator= (const IntCell & rhs)
{ if (this != &rhs)
    *storedValue = *rhs.storedValue;
    return *this;
}

// DEFAULT operator= WON'T WORK!!
const IntCell & IntCell::operator= (const IntCell & rhs)
{ if (this != &rhs)
    storedValue = rhs.storedValue; // copies pointer instead of pointee
    return *this;
}
Moving on...

☐ More now on <list> ...
More on STL <list>

- STL list is implemented as a doubly-linked list

- Recall example of using <list>:

```cpp
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<char> coll;  // list container for character elements

    // append elements from 'a' to 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }

    /* print all elements; while still elements, print and remove first element
    while (! coll.empty()) {
        cout << coll.front() << ' ';
        coll.pop_front();
    }
    
    cout << endl;
}
```

Output?

```
abcdefghijkmnopqrstuvwxyz
```
Basic `<list>` operations

Creating, copying, and destroying operations:

- `list<Elem> c` ← creates an empty list of type “Elem” without elements
- `list<Elem> c1(c2)` ← creates a copy of another list of the same type
- `list<Elem> c(n)` ← creates a list with n elements created by default constructor
- `list<Elem> c(n,elem)` ← creates a list initialized with n copies of element elem
- `list<Elem> c(beg,end)` ← creates a list initialized with the elements of the range [beg,end]
- `c.~list<Elem>()` ← destroys all elements and frees the memory
Basic `<list>` operations (con’t.)

Non-modifying operations:

- `c.size()` ➔ returns the number of elements
- `c.empty()` ➔ returns whether the container is empty
- `c.max_size()` ➔ returns the maximum number of elements possible
- `c1 == c2` ➔ returns whether `c1` is equal to `c2`
- `c1 != c2` ➔ returns whether `c1` is not equal to `c2`
- `c1 < c2` ➔ returns whether `c1` is less than `c2`
- `c1 > c2` ➔ returns whether `c1` is greater than `c2`
- `c1 <= c2` ➔ returns whether `c1` is less than or equal to `c2`
- `c1 >= c2` ➔ returns whether `c1` is greater than or equal to `c2`

Assignment operations:

- `c1 = c2` ➔ assigns all elements of `c2` to `c1`
- `c.assign(n,elem)` ➔ assigns `n` copies of element `elem`
- `c.assign(beg,end)` ➔ assigns the elements of the range `[beg,end)`
- `c1.swap(c2)` ➔ swaps the data of `c1` and `c2`
- `swap(c1,c2)` ➔ *same as above*
Basic `<list>` operations (con’t.)

Element access:

c.front() ↦ returns 1st element (NO check whether 1st element exists)
c.back() ↦ returns last element (NO check whether last element exists)

Example:

```cpp
list<Elem> coll;

cout << coll.front();    //RUNTIME ERROR → undefined behavior
if (!coll.empty()) {
    cout << coll.back();    // THIS IS OK
}  
```
Basic `<list>` operations (con’t.)

Iterator functions:

- `c.begin()` ← returns a bidirectional iterator for 1st element
- `c.end()` ← returns a bidirectional iterator for pos. after last element
- `c.rbegin()` ← returns a reverse iterator for 1st element of a reverse iteration
- `c.rend()` ← returns a reverse iterator for pos. after last element of a reverse iteration
Basic `<list>` operations (con’t.)

Inserting and Removing elements:

- `c.insert(pos, elem)` inserts at iterator position `pos` a copy of `elem` and returns position of new element.
- `c.insert(pos, n, elem)` inserts at iterator position `pos` `n` copies of `elem` (and returns nothing).
- `c.insert(pos, beg, end)` inserts a copy of all elements of the range `[beg, end)` (and returns nothing).
- `c.push_back(elem)` appends a copy of `elem` at end.
- `c.pop_back()` removes last element (does not return it).
- `c.push_front(elem)` inserts a copy of `elem` at beginning.
- `c.pop_front()` removes the first element (does not return it).
- `c.remove(val)` removes all elements with value `val`.
- `c.remove_if(op)` removes all elements for which `op(elem)` is true.
- `c.erase(pos)` removes element at iterator position `pos` and returns pos of next element.
- `c.erase(beg, end)` removes all elements of range `[beg, end)`; returns pos of next element.
- `c.resize(num)` changes the number of elements to `num` (if size grows, new elements created by default constructor).
- `c.resize(num, elem)` changes the number of elements to `num` (if size grows, new elements are copies of `elem`).
- `c.clear()` removes all elements.
Basic <list> operations (con’t.)

Splice functions:

- `c.unique()` \(\rightarrow\) removes duplicates of consecutive elements with same value
- `c.unique(op)` \(\rightarrow\) removes duplicates of consecutive elements, for which \(\text{op}(\cdot)\) yields true
- `c1.splice(pos,c2)` \(\rightarrow\) moves all elements of \(c2\) to \(c1\) in front of iterator position \(pos\)
- `c1.splice(pos,c2,c2pos)` \(\rightarrow\) moves the element at \(c2pos\) in \(c2\) in front of \(pos\) of list \(c1\)
- `c1.splice(pos,c2,c2beg,c2end)` \(\rightarrow\) moves all elements of the range \([c2beg,c2end)\) in \(c2\) in front of \(pos\) of list \(c1\)
- `c.sort()` \(\rightarrow\) sorts all elements with operator <
- `c.sort(op)` \(\rightarrow\) sorts all elements with \(\text{op}(\cdot)\)
- `c1.merge(c2)` \(\rightarrow\) assuming both containers are sorted, moves all elements of \(c2\) into \(c1\) so that all elements are merged and still sorted
- `c1.merge(c2,op)` \(\rightarrow\) assuming both containers are sorted due to sorting criterion \(\text{op}(\cdot)\), moves all elements of \(c2\) into \(c1\) so that all elements are merged and still sorted
- `c.reverse` \(\rightarrow\) reverses the order of all elements
Exception Handling

- Lists have best support for “safety” in STL
- Almost all list operations will either succeed or have no effect
“find” algorithm

InputIterator
find(InputIterator beg, InputIterator end, const T& value)

This returns the position of the first element in the range [beg,end) that has a value equal to value.

Example:

list<int> coll;
list<int>::iterator pos1;

pos1 = find(coll.begin(), coll.end(), 4);
“copy” and Stream Iterators

- **“copy”:**
  ```cpp
  OutputIterator copy(InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg)
  ```
  This copies all elements of source range `[sourceBeg,sourceEnd)` into the destination range starting with `destBeg`

- **stream iterator:** used to read from and write to a stream
  ```cpp
  ostream_iterator<T>(ostream) \leftarrow creates an output stream iterator
  or
  ostream_iterator<T>(ostream, delim) \leftarrow creates an ostream iterator with string `delim` as delimiter between values
  ```

**Example:**

```cpp
list<int> list1, list2;

copy (list1.begin(), list1.end(), ostream_iterator<int>(cout, " "));
```
#include <iostream>
#include <list>
#include <iterator>
using namespace std;

void printLists (const list<int>& l1, const list<int>& l2)
{
    cout << "list1: ";
    copy (l1.begin(), l1.end(), ostream_iterator<int>(cout, " ") );
    cout << endl << "list2: ";
    copy (l2.begin(), l2.end(), ostream_iterator<int>(cout, " ") );
    cout << endl << endl;
}

int main()
{
    list<int> list1, list2;

    for (int i=0; i<6; ++i) {
        list1.push_back(i);
        list2.push_front(i);
    }
    printLists(list1, list2);

    // inserts all elements of list1 before
    // first element with value 3 of list2
    list2.splice(find(list2.begin(),list2.end(),
    3),
    list1);
    printLists(list1, list2);

    // move first element to the end
    list2.splice(list2.end(),
    list2,
    list2.begin());
    printLists(list1, list2);

    // sort second list, assign to list1
    // and remove duplicates
    list2.sort();
    list1 = list2;
    list2.unique();
    printLists(list1,list2);

    // merge both sorted lists into the
    // first list
    list1.merge(list2);
    printLists(list1, list2);
}

Output?
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0
list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0
list1:
list2: 4 0 1 2 3 4 5 3 2 1 0 5
list1:
list2: 4 0 1 2 3 4 5 3 2 1 0 5
list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0
list1:
list2: 0 0 1 1 1 2 2 3 3 3 4 4 4 5 5 5 5
list2: