More on Inheritance

Outline

- How to prevent object copying
- When we use friend class
- How to reuse class definitions using templates

Problem of Object Copying

- Another subtle (thus hard-to-debug) problem can arise due to the use of call-by-value object parameters
- Let us assume that we want to define an ordinary function
  - `check_owner (railroad_car r, char *s)`
  - Which checks if the serial # of a railroad car equals to a character string

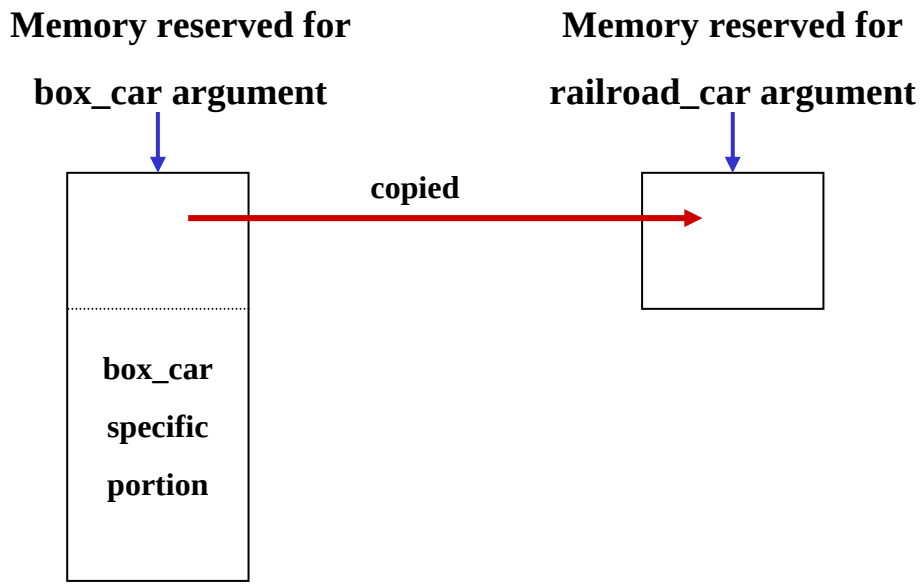
```
int nyc_count = 0;
for (n = 0; n < car_count; ++n)
  if( check_owner (*(train[n]), "NYC") )
    ++nyc_count;
cout << "There are " << nyc_count << "NYC cars." << endl;
```


```
int check_owner (railroad_car r, char* s) {
  if (strncmp (s, r.serial_number, 3))
    return 0;
  else
    return 1;
}
```


```
class railroad_car {
  public: char *serial_number;
          // Constructors:
          railroad_car ( ) { }
          railroad_car (char *input_buffer) {
            // Create new array just long enough:
            serial_number = new char[strlen(input_buffer) + 1];
            // Copy string into new array:
            strcpy (serial_number, input_buffer);
          }
          // Destructor:
          virtual ~railroad_car ( ) {
            cout << "Deleting a railroad serial number" << endl;
            delete [ ] serial_number;
          }
          // Other:
          virtual char* short_name ( ) {return "rrc";}
          virtual double capacity ( ) {return 0.0;}
};
```


Problem of Object Copying

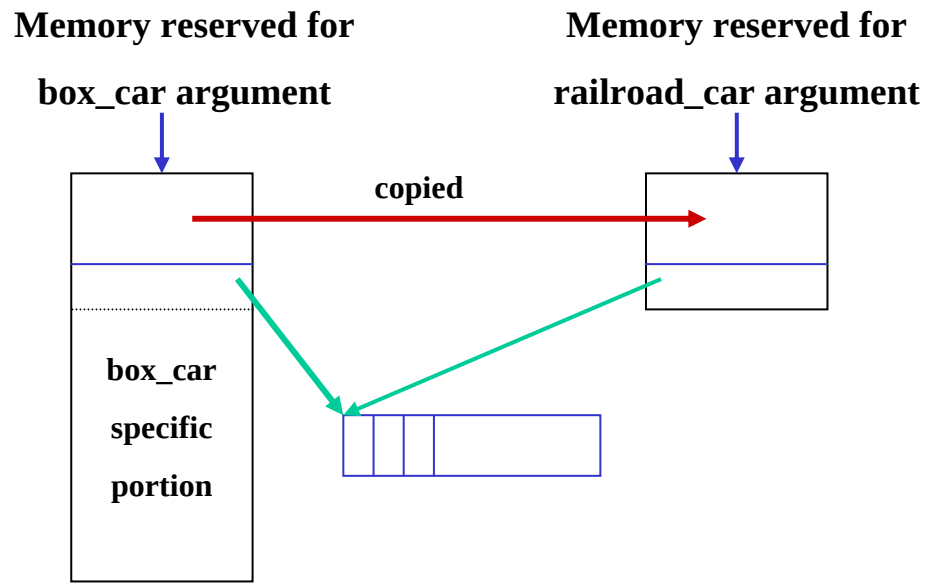
- Still have the same issue of copying only railroad portion
  - It is fine since in this particular situation, we need only railroad portion



- Then, what is the real problem for this situation?
  - Copying of the serial_number field

Problem of Object Copying

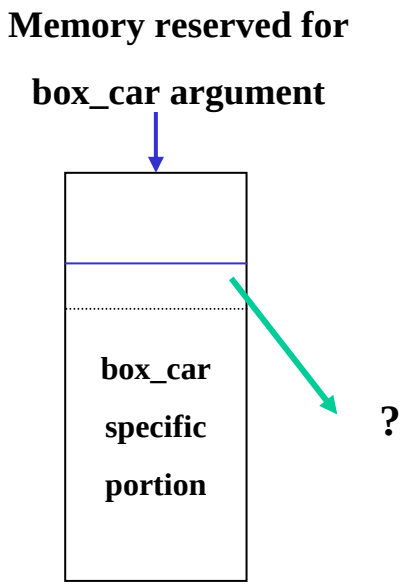
- Copying of the pointer to a serial_number object
  - When the copying is made, the pointer is also copied



- What can be the problem? When the function returns the destructor will be called for the copied object
  - Why? A parameter is also a local variable which should be deallocated when the function returns

Problem of Object Copying

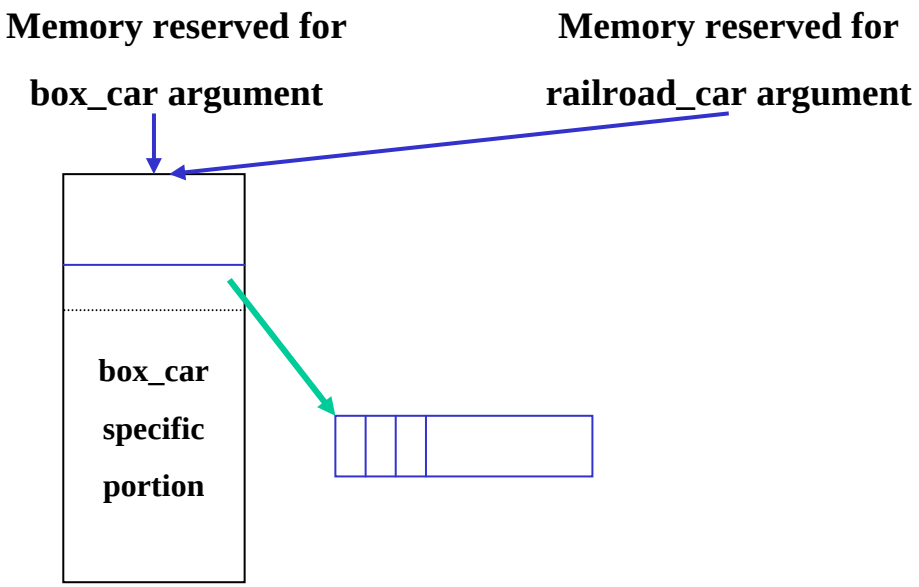
- When the destructor is called for the copied object
  - It will eliminate the (shared) serial_number object



- Now the pointer in the original object is dangling
  - The worst thing is that the problem is not immediately detected

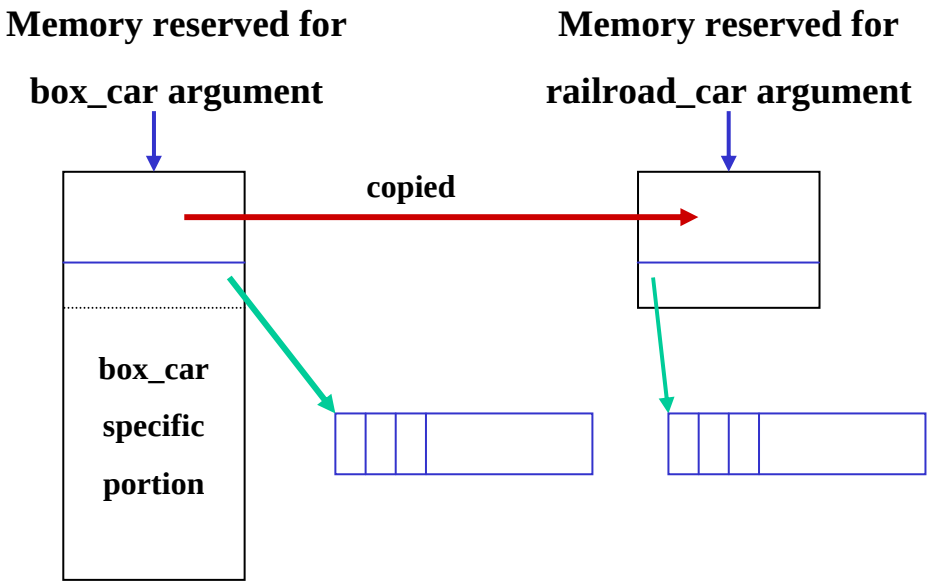
Solution of Object Copying

- Replace call-by-value by call-by-reference
  - No copy is made, no copy memory is reclaimed, no destructor is called



Another Solution of Object Copying

- Define your own copy constructor
  - C++ programs copy objects using a copy constructor
  - C++ compiler provides a default copy constructor if you do not supply
    - Which is simply member-wise copying as we already saw
  - You can provide one, which actually duplicates the serial_number object



Which Solution is Preferable

- Experienced programmers prefer call-by-reference. Why?
  - C++ objects generally represent real-world objects
  - Any object creation, copying, destruction should mimic corresponding actions in real world
  - Objects should not be copied merely because a function is called, and objects should not be destroyed merely because a function returns

- There are 3 reasons to avoid call-by-value object parameter
  - Subclass portion is not copied
  - Copying can lead to obscure reclamation bug
  - Copying/reclamation in function calls violate the principle of software objects mimicing real-world objects

How to Avoid Inadvertent Object Copying

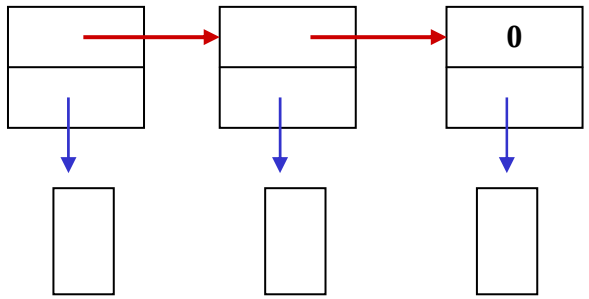
- Define your own copy constructor
  - If you define a destructor, define also a copy constructor
- Place it in the private part so that it cannot be called outside

```
class railroad_car {
  public: char *serial_number;
          // Constructors:
          railroad_car ( ) { }
          railroad_car (char *input_buffer) {
            // Create new array just long enough:
            serial_number = new char[strlen(input_buffer) + 1];
            // Copy string into new array:
            strcpy (serial_number, input_buffer);
          }
          ……..
 private:
          // Never-to-be-called copy-constructor protytype:
          railroad_car (railroad_car&);
};
```


How to Implement Lists

- Build a linked list for storing railroad cars instead of array
- One way is adding a link pointer field in railroad_car object
- This kind of internal pointers is not recommended
  - Adding a field to existing class definitions would be awkward
  - If we build a separate list for each car class, we need a point for each list

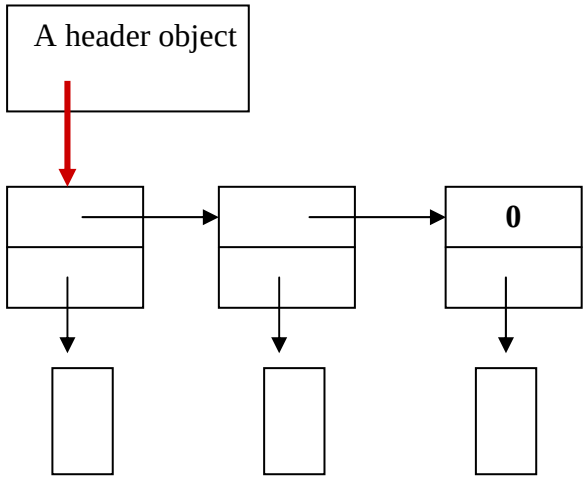
- Most experienced programmers use external pointers
  - Create a new class `link` which has two pointers
    - Pointer to the next link object
    - Pointer to a railroad_car object



Implementation of Linked Lists

- We add one more class, `header`
  - Each header object, one per list, contains a pointer to the 1st `link` object
- Member variables for header and link

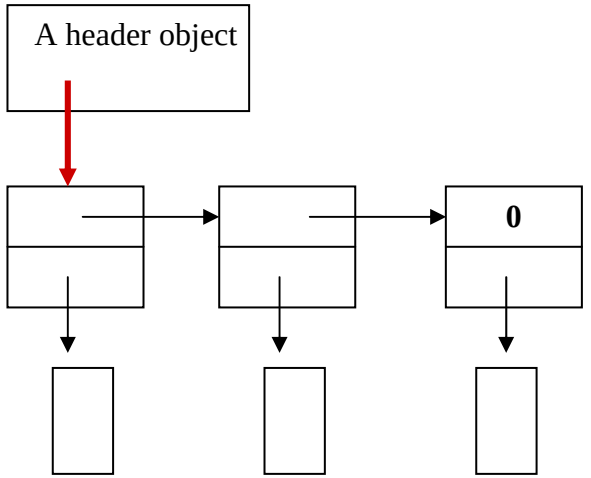
```
class link {
  public:  link *next_link_pointer;
           railroad_car *element_pointer;
           ...
};
class header {
  public:  link *first_link_pointer;
           header() {
             first_link_pointer = NULL;
           }
};
```



Implementation of Linked Lists

- We need a member function `add` for the `header`
  - `train.add(` pointer-to-`railroad_car` object `)`

```
class header {
  public:  link *first_link_pointer;
           header() {
             first_link_pointer = NULL;
           }
           void add (railroad_car *new) {
             first_link_pointer = new link (new, first_link_pointer);
           }
};
class link {
  public:  link *next_link_pointer;
           railroad_car *element_pointer;
           link (railroad_car *e, link *l) {
             element_pointer = e;
             next_link_pointer = l;
           }
};
```



Implementation of Linked Lists

- We now need to be able to access elements of the list
  - `current_link_pointer`:
  - `advance()`: advances `current_link_pointer`
  - `access()`: obtains a pointer to a `railroad_car` object from the `current_link_pointer`

```
class header {
  public:  link *first_link_pointer;
           link *current_link_pointer;
           header() {
             first_link_pointer = NULL;
             current_link_pointer = NULL;
           }
           void add (railroad_car *new) {
             first_link_pointer = new link (new, first_link_pointer);
             current_link_pointer = first_link_pointer
           }
};
```


Implementation of Linked Lists

- More functions
  - `endp()` for end predicate that checks if `current_link_pointer` is null
  - `Reset(): current_link_pointer to first_link_pointer`

```
    void advance ( ) {
      current_link_pointer = current_link_pointer -> next_link_pointer;
    }
    railroad_car* access ( ) {
      return current_link_pointer -> element_pointer;
    }
    int endp ( ) {
      return ! current_link_pointer;
    }
    void reset ( ) {
      current_link_pointer = first_link_pointer;
    }
};
```


Main() of Linked Lists

```
header train;

main () {
  // No initialization or increment expressions:
  for (; cin >> input_buffer;)
    switch (extract_car_code (input_buffer)) {
      case eng_code: train.add (new engine (input_buffer));   break;
      case box_code: train.add (new box_car (input_buffer));  break;
      case tnk_code: train.add (new tank_car (input_buffer)); break;
      case cab_code: train.add (new caboose (input_buffer));  break;
    }

  train.reset ();

  // No initialization; incremernt expression advances list:
  for (; !train.endp ( ); train.advance ())
    // Display number, short name, and capacity and terminate the line:
    cout << train.access () -> serial_number << "   "
         << train.access () -> short_name () << "   "
         << train.access () -> capacity () << endl;
}
```


Sample Data and Result

```
-------- Sample Data ---
TPW-E-783
PPU-B-422
NYC-B-988
NYC-T-988
------- Result -------
NYC-T-988    tnk 1539.38
NYC-B-988    box 3990
PPU-B-422    box 3990
TPW-E-783    eng 0
```


Hiding Implementation Details of Lists

- We want to move some members to private section

```
class link {
  private:
    link *next_link_pointer;
    railroad_car *element_pointer;
    link (railroad_car *e, link *l) {
      element_pointer = e;
      next_link_pointer = l;
    }
};
class header {
  public:
           header() {
             first_link_pointer = NULL;
             current_link_pointer = NULL;
           }
           ……..
  private: link *first_link_pointer;
           link *current_link_pointer;
};
```


Solution

- However, it does not work
  - why not? access of link members by header public functions
  - Solution: resorting to friend class

```
class link {
  friend class header;
  private:
    link *next_link_pointer;
    railroad_car *element_pointer;
    link (railroad_car *e, link *l) {
      element_pointer = e;
      next_link_pointer = l;
    }
};
```


How to Reuse Classes using Templates

- You want to make other lists once you have railroad car lists
- You can edit `link` and `header` class definitions by hand
  - But this is not a good idea. Why not?
    - Manual editing is error-prone
    - If you have future improvements, you need to propagate them to all
    - You need to give separate names for `header` and `link` for each list
- Solution
  - C++ provides a template mechanism, which enables you to define generic `header` and `link` template classes
  - What is template? 형판(型板)

Re-Interpretation of Link and Header Classes

```
class link {
  friend class header;
  private:
    link *next_link_pointer;
    railroad_car *element_pointer;
    link (railroad_car *e, link *l) {
      …
    }
};
class header {
  public:   …
            void add (railroad_car *new) {
            first_link_pointer = new link (new, first_link_pointer);
            …
            }
            railroad_car* access ( ) {
                return current_link_pointer -> element_pointer;
            }
            ….
  private:  link *first_link_pointer;
            link *current_link_pointer;
};
```


Converting into Template Class Definitions

- Convert into template class definition by adding prefixes

```
template <class link_parameter>
class link {
  ...
};

template <class header_parameter>
class header {
  ...
};
```

- Replace red-colored ones by appropriate parameter names
- Provide a specializing argument when another template class name is used inside the class definition
- Finally, define class variables using parametered classes
  - header train; => header<railroad_car> train;

Re-Interpretation of Link and Header Classes

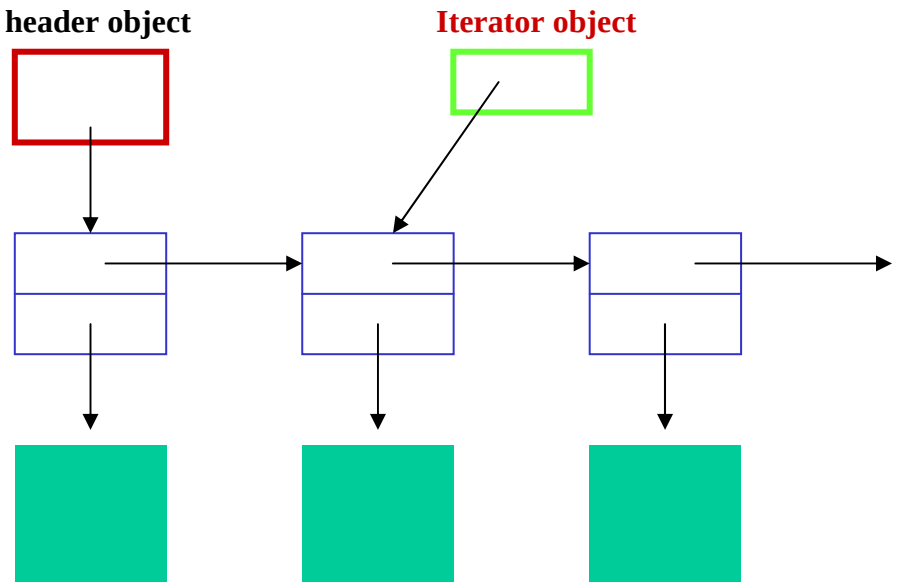
```
template <class link_parameter> class link {
  friend class header<link_parameter>;
  private:
          link *next_link_pointer;
          link_parameter *element_pointer;
          link (link_parameter *e, link *l) {
               …
          }
};
template <class header_parameter> class header {
  public:   …
          void add (header_parameter *new) {
          first_link_pointer = new
                        link<header_parameter>(new,first_link_pointer);
            …
          }
          header_parameter * access ( ) {
               return current_link_pointer -> element_pointer;
          }
           ….
  private:  link< header_parameter > *first_link_pointer;
            link< header_parameter > *current_link_pointer;
};
```


Converting into Template Class Definitions

- Finally, define class variables using parametered classes
  - `header train;` => `header<railroad_car>  train;`
- This will cause the header template class to be instantiated so as to deal with header objects belonging to `railroad_car` class
- You can instantiate it using other classes for other type lists

Iteration Class Objects

- Previously, we can have only one traversal for the list
  - Using the `current_link_pointer` in the `header` class
- What if we have multiple traversals going simultaneously?
- We use iteration class for more than one traversal
  - Which separates list construction and list traversal



Iterator Template Class

```
template <class iterator_parameter>
class iterator {
  public:  ...
    iterator_parameter* access ( ) {
      return current_link_pointer -> element_pointer;
    }
    void advance ( ) {
      current_link_pointer = current_link_pointer -> next_link_pointer;
    }
    int endp ( ) {
      return ! current_link_pointer;
    }
    void reset ( ) {
      current_link_pointer = first_link_pointer;
    }
  private link<iterator_parameter>* current_link_pointer;
          link<iterator_parameter>* first_link_pointer;
};
```

- Why do we need `first_link_pointer`?
  - For reuse of the iterator

Iterator Class Constructor

- Defining iterator constructor is not simple
  - Constructor needs to get the first link with the header object argument
- How do we declare a header and an iterator variable?
  - `header<railroad_car> train;`
  - `iterator<railroad_car> train_iterator (train);`
- The format of the constructor would be

```
iterator (header<iterator_parameter> & header)
{
  first_link_pointer = header.first_link_pointer;
  current_link_pointer = first_link_pointer;
};
```

- `Iterator` class should be a friend class of `link` class

Analyze Program

```
template <class iterator_parameter>
class iterator {
  public:
    iterator (header<iterator_parameter>& header) {
      first_link_pointer = header.first_link_pointer;
      current_link_pointer = first_link_pointer;
    }
    iterator_parameter* access ( ) {
      return current_link_pointer -> element_pointer;
    }
    void advance ( ) {
      current_link_pointer = current_link_pointer -> next_link_pointer;
    }
    int endp ( ) {
      return ! current_link_pointer;
    }
    void reset ( ) {
      current_link_pointer = first_link_pointer;
    }
  private: link<iterator_parameter>* current_link_pointer;
           link<iterator_parameter>* first_link_pointer;
};
```


Analyze Program

```
template <class link_parameter>
class link {
  friend class iterator<link_parameter>;
  friend class header<link_parameter>;
  private:
    link *next_link_pointer;
    link_parameter *element_pointer;
    link (link_parameter *e, link *l) {
      element_pointer = e;
      next_link_pointer = l;
    }
};
```


Analyze Program

```
template <class header_parameter>
class header {
  friend class iterator<header_parameter>;
  public:
    header ( ) {
      first_link_pointer = NULL;
    }
    void add (header_parameter *new_element) {
      first_link_pointer =
        new link<header_parameter> (new_element, first_link_pointer);
    }
  private:
    link<header_parameter> *first_link_pointer;
};
```


Analyze Program

```
header<railroad_car> train;
char input_buffer[100];
enum {eng_code = 'E', box_code = 'B', tnk_code = 'T', cab_code = 'C'};
char extract_car_code (char *input_buffer) {return input_buffer[4];}

main () {
  // No initialization or increment expressions:
  for (; cin >> input_buffer;)
    switch (extract_car_code (input_buffer)) {
      case eng_code: train.add (new engine (input_buffer));   break;
      case box_code: train.add (new box_car (input_buffer));  break;
      case tnk_code: train.add (new tank_car (input_buffer)); break;
      case cab_code: train.add (new caboose (input_buffer));  break;
    }
  // Define and initialize iterator class object:
  iterator<railroad_car> train_iterator (train);
  // Iterate:
  train_iterator.reset ();
  // No initialization; incremernt expression advances list:
  for (; !train_iterator.endp ( ); train_iterator.advance ())
    // Display number, short name, and capacity and terminate the line:
    cout << train_iterator.access () -> serial_number << "   "
         << train_iterator.access () -> short_name () << "   "
         << train_iterator.access () -> capacity () << endl;
}
```