Step-by-Step Example of a List Class in C++

Antonio Carzaniga

We walk through various implementations of a *linked list* that illustrate various features of C++. We start by implementing a list of `int` objects (numbers) with a very simple abstract interface:

Add x adds element x to the list

Size x returns the number of elements contained in the list

We implement this structure with a singly-linked list. Each "link" in the list is represented by a `list_node` object, and the list itself is represented by a `list` object as follows:

```
#include <iostream>

struct list_node {
    int v;
    list_node * next;
};

class list {
    list_node * head;           // head of the linked list
    unsigned int n;             // number of elements in the list
};
```

Notice that we define `list_node` as a `struct` and `list` as a `class`. However, there is almost no difference between the two. A `struct` is exactly like a `class`, except that the members of a `struct` are *public*, whereas the members of a class are by default *private*.
We then add the basic operations as member functions (methods), including the constructors and, for `list`, also a destructor. The destructor is necessary because we need to deallocate the memory used by the linked list.

```
struct list_node {
    int v;
    list_node * next;

    list_node(int k, list_node * n) : v(k), next(n) {}
};

class list {
    list_node * head;
    unsigned int n;

public:
    list() : head(nullptr), n(0) {}
    ~list() {
while (head) {
    list_node * tmp = head;
    head = head->next;
    delete(tmp);
}
    }

    unsigned int size() const {
return n;
    }

    void add(int i) {
head = new list_node(i, head);
n += 1;
    }
};
```

Notice that we include all the method *definitions* within the class definition. An often preferable alternative is to separate the method declarations, within the class definitions, from their definitions, which could well be in a separate file and in any case outside the class definition. This is what we do from now on.

```
struct list_node {
    int v;
    list_node * next;

    list_node(int k, list_node * n);
};

class list {
    list_node * head;
    unsigned int n;

public:
    list();
    ~list();

    unsigned int size() const;
    void add(int i);
};

list_node::list_node(int k, list_node * n)
    : v(k), next(n) {}

list::list()
    : head(nullptr), n(0) {}

list::~list() {
    while (head) {
list_node * tmp = head;
head = head->next;
delete(tmp);
    }
}

unsigned int list::size() const {
    return n;
}

void list::add(int i) {
    head = new list_node(i, head);
    n += 1;
}
```

Now that we have a minimally functional implementation, we can also add a test program (`main`).

```
#include <iostream>
#include <cassert>

struct list_node {
    int v;
    list_node * next;

    list_node(int k, list_node * n);
};

class list {
    list_node * head;
    unsigned int n;

public:
    list();
    ~list();

    unsigned int size() const;
    void add(int i);
};

list_node::list_node(int k, list_node * n)
    : v(k), next(n) {}

list::list()
    : head(nullptr), n(0) {}

list::~list() {
    while (head) {
list_node * tmp = head;
head = head->next;
delete(tmp);
    }
}

unsigned int list::size() const {
    return n;
}
```

```
void list::add(int i) {
    head = new list_node(i, head);
    n += 1;
}

int main() {
    list l;
    l.add(3);
    l.add(1);
    l.add(4);
    l.add(1);
    l.add(5);
    l.add(9);
    l.add(2);
    l.add(6);
    l.add(5);
    assert(l.size() == 9);
    std::cout << "PASS" << std::endl;
}
```

Of course, a list that only allows one to add elements and then get the number of elements added is not very useful. At a minimum, the data structure should also support an *iteration*. The way we do that is through the classic iteration pattern used extensively in the C++ standard library, namely through *iterator* objects.
An iterator is analogous to a *pointer*. Consider the following code:

```
int A[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int main() {
    for (int * itr = A; itr != A + 10; ++itr)
*itr = 0;
}
```

Here the data structure is a simple array and the application uses a pointer (`itr`) to iterate over the array. We could then modularize the array inside a specific data structure that would still allow the application to iterate over the array in the same way:

```
#include <iostream>

class A {
    int data[10];

public:
    int * begin() { return data; }
    int * end() { return data + 10; }
};

int main() {
    A a;
    int i = 0;
    for (int * itr = a.begin(); itr != a.end(); ++itr)
*itr = ++i;
```

```
    for (int * itr = a.begin(); itr != a.end(); ++itr)
std::cout << *itr << std::endl;
}
```

We want to do the same now for out `list` data structure. In order to do that, we need to have a sort of pointer, meaning something that would allow the application to iterate through the elements (`++itr`) and to access the elements (`*itr`). This is the role of an *iterator*. In addition to that, the way an application uses iterators a requires proper initialization and also a comparison for iterators. We therefore define a specific iterator class for our linked list. In this case, the iterator class, which we implement as an inner class within `list`, is essentially a pointer to a wrapper for a pointer to a `list_node` object.

```
#include <iostream>
#include <cassert>

struct list_node {
    int v;
    list_node * next;

    list_node(int k, list_node * n);
};

class list {
    list_node * head;
    unsigned int n;

public:
    list();
    ~list();

    unsigned int size() const;
    void add(int i);

    class iterator;

    iterator begin() const;
    iterator end() const;
};

list_node::list_node(int k, list_node * n)
    : v(k), next(n) {}

list::list()
    : head(nullptr), n(0) {}

list::~list() {
    while (head) {
list_node * tmp = head;
head = head->next;
delete(tmp);
    }
}
```

```
unsigned int list::size() const {
    return n;
}

void list::add(int i) {
    head = new list_node(i, head);
    n += 1;
}

class list::iterator {
    list_node * p;
public:
    iterator(list_node *x) : p(x) {};
    iterator(const iterator & other) : p(other.p) {};

    int & operator * () const;
    bool operator != (const iterator & other) const;
    iterator & operator ++ ();
};

int & list::iterator::operator * () const {
    return p->v;
}

bool list::iterator::operator != (const iterator & other) const {
    return p != other.p;
}

list::iterator & list::iterator::operator ++ () {
    p = p->next;
    return *this;
}

list::iterator list::begin() const {
    return iterator(head);
}

list::iterator list::end() const {
    return iterator(nullptr);
}

int main() {
    list l;
    l.add(3);
    l.add(1);
    l.add(4);
    l.add(1);
    l.add(5);
    l.add(9);
    l.add(2);
    l.add(6);
    l.add(5);
    assert(l.size() == 9);

    for (list::iterator itr = l.begin(); itr != l.end(); ++itr)
std::cout << *itr << std::endl;
}
```

Notice the definition of the various operators applied to `iterator` objects. We define the increment operator (unary prefix ++) to move to the next element. We then define the dereference operator (unary `*`) to access an element. In this case, notice that the dereference operator returns a *reference* to the element, meaning a reference to the `int` object pointed to by the iterator. This is necessary to be able not only to read the values of the elements in the list, but also to assign their values (e.g., `*itr = 7`). Notice also that this definition of the dereference operator is consistent with the derefenrence of a normal pointer value. For example, if we have an object `int * p`, then `*p` is an l-value, meaning it refers to an object that can therefore be used as the target in an assignment.