Containers in memory

As you already know from the previous chapters, an object takes memory space on one of the memory segments provided to the process. Most of the time, we are interested in the stack or heap memory. An automatic object takes space on the stack. The following two declarations both reside on the stack:

```
struct Email

{

// code omitted for brevity

};

int main() {

Email obj;

Email* ptr;

}
```

Although ptr represents a pointer to an Email object, it takes space on the stack. It can point to a memory location allocated on the heap, but the pointer itself (the variable storing the address of a memory location) resides on the stack. This is crucial to understand and remember before going further with vectors and lists. As we saw earlier in the chapter, implementing a vector involves encapsulating a pointer to an inner buffer that represents an array of elements of the specified type. When we declare a `Vector` object, it takes the necessary amount of stack memory to store its member data. The `Vector` class has the following three members:

```
template <typename T>

class Vector

{

public:

// code omitted for brevity

private:

int capacity_;

int size_;

T* buffer_;

};
```

Supposing that an integer takes 4 bytes and a pointer takes 8 bytes, the following `Vector` object declaration will take at least 16 bytes of stack memory:

```
int main()

{
```

```
Vector<int> v;
}
```

Here's how we picture the memory layout for the preceding code:

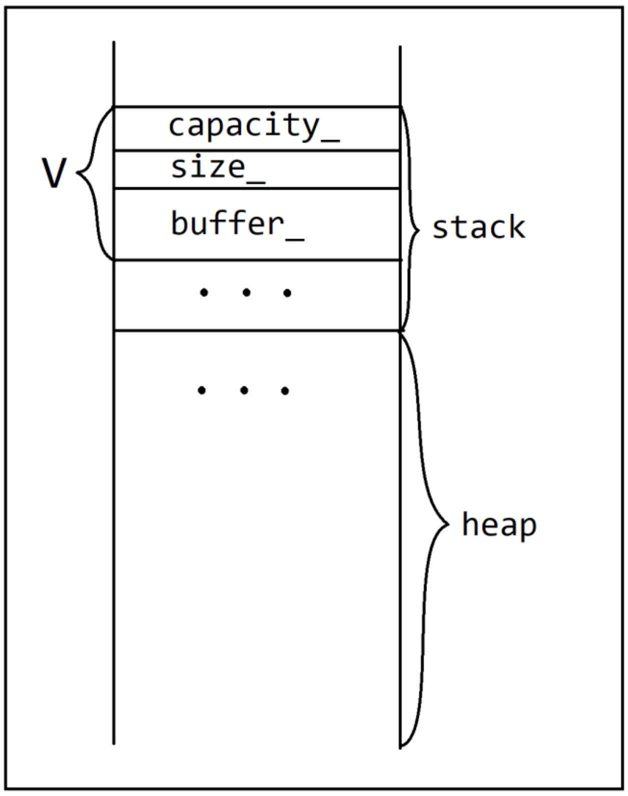

Figure 6.13: Vector representation in the memory layout

After inserting elements, the size of the vector on the stack will stay the same. The heap comes to the scene. The buffer_ array points to a memory location allocated using the new[] operator. For example, look at the following code:

```
// we continue the code from previous listing
v.push_back(17);
v.push_back(21);
v.push_back(74);
```

Each new element that we push to the vector will take space on the heap, as shown in the following diagram:

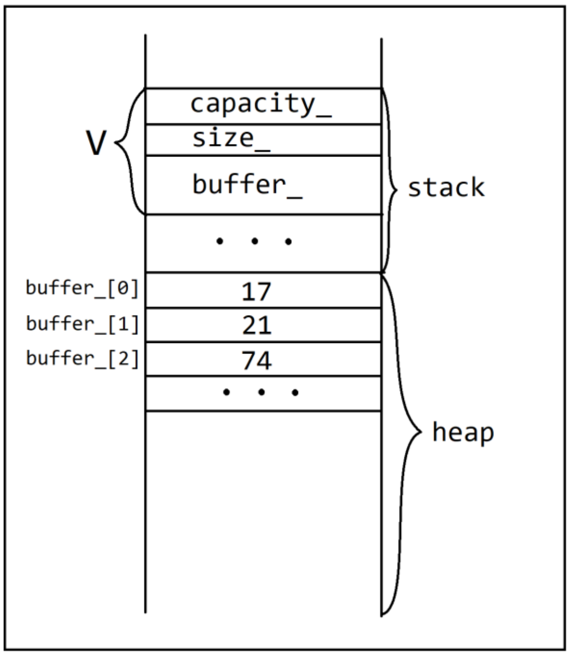

Figure 6.14: Vector representation in the memory layout after pushing elements

Each newly inserted element resides right after the last element of the `buffer_` array. That's why we can say the vector is a cache-friendly container.

Declaring a linked-list object also takes memory space on the stack for its data members. If we discuss the simple implementation that stores only the `head_` pointer, the following list object declaration will take at least 8 bytes of memory (for the `head_` pointer only):

```
int main()

{

LinkedList<int> list;

}
```

The following illustration depicts the memory layout for the preceding code (page 24):

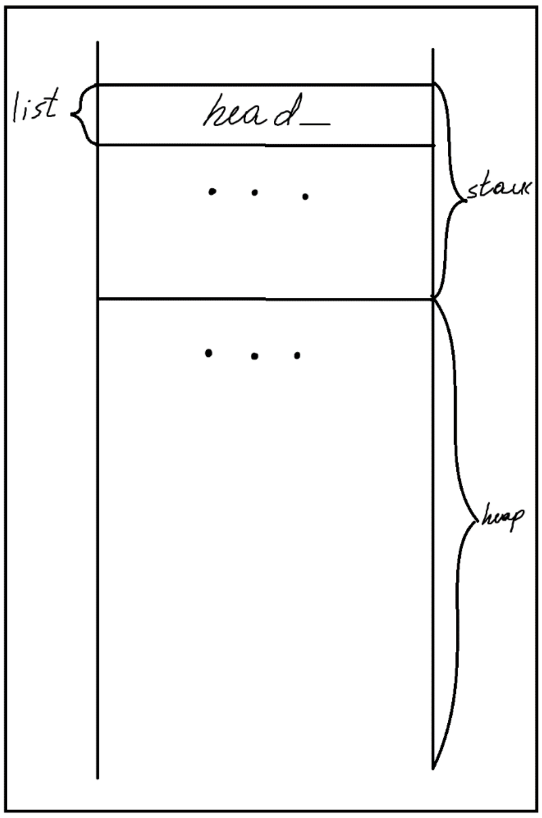

Figure 6.15: List representation in the memory layout

Inserting a new element creates an object of type `node` on the heap. Take a look at the following line:

```
list.push_back(19);
```

Here's how the memory illustration will change after inserting a new element:

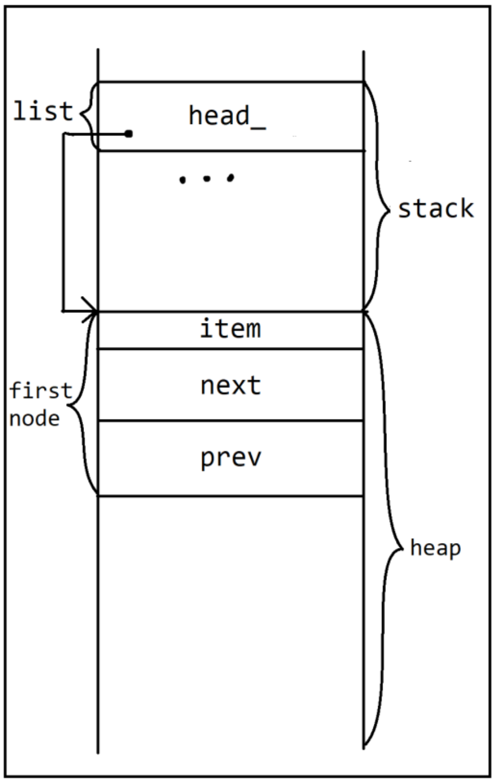

Figure 6.16: List representation in the memory layout after inserting an element

Take care that the node with all its data members resides on the heap. The item stores the value that we have inserted. When we insert another element, a new node will be created again. This time, the next pointer of the first node will point to the newly inserted element. And the newly inserted node's `prev` pointer will point to the previous node of the list. The following illustration depicts the linked list's memory layout after inserting the second element:

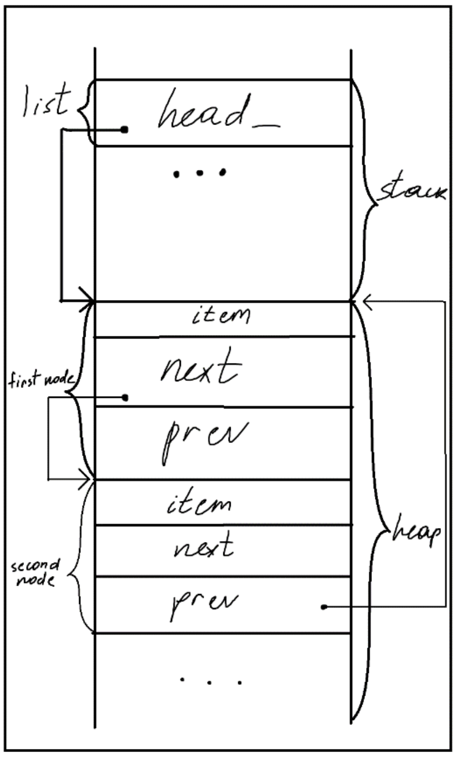

Figure 6.17: List representation in the memeory layout after inserting more elements

An interesting thing happens when we allocate some random objects on the heap in between inserting elements into the list. For example, the following code inserts a node into the list, then allocates space for an integer (not related to the list). Finally, it again inserts an element into the list:

```
int main()
{
LinkedList<int> list;
list.push_back(19);
int* random = new int(129);
list.push_back(22);
}
```

This intermediate random object declaration spoils the order of list elements, as shown in the following diagram:

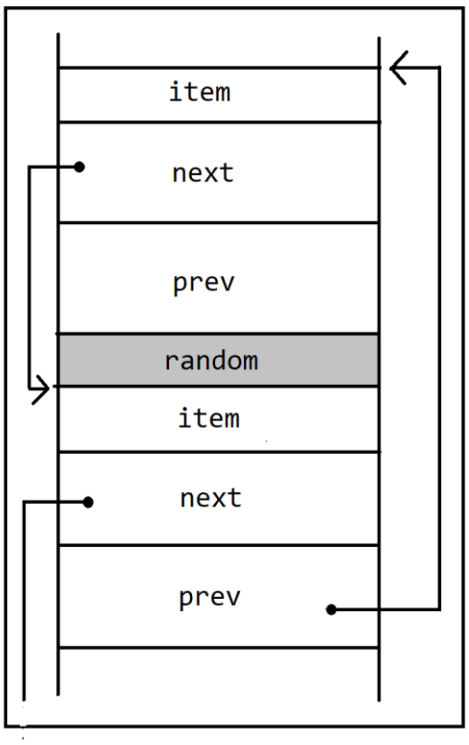

Figure 6.18: The representation of the list elements randomly arranged in a memory layout

The preceding diagram gives us a hint that the list is not a cache-friendly container, because of its structure and the allocation of its elements.

Pay attention to the memory overhead created by incorporating each new node into the code. We pay an additional 16 bytes (considering the pointer takes 8 bytes of memory) for one element. Thus, lists lose the game of optimal memory use to vectors.

We can try to fix the situation by introducing a preallocated buffer in the list. Each new node creation will then pass via the **placement new** operator. However, it is wiser to choose a data structure that better fits the problem of interest.

In real-world application development, programmers rarely implement their own vectors or linked lists. They usually use tested and stable library versions. C++ provides standard containers for both vectors and linked lists. Moreover, it provides two separate containers for singly and doubly linked lists.

The next in line for data structures that help us solve different kinds of problems are Graphs and trees.