Phần 2 – CÁC CẤU TRÚC DỮ LIỆU

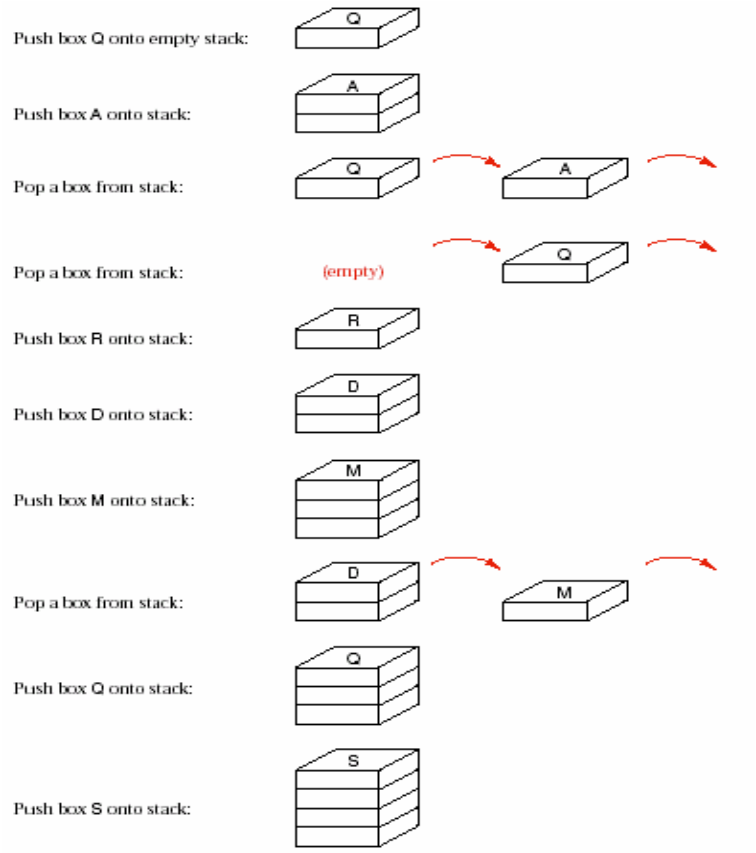
Chương 2 – NGĂN XẾP

Chúng ta sẽ tìm hiểu một CTDL đơn giản nhất, đó là ngăn xếp. Một cách nhất quán như phần giới thiệu môn học đã trình bày, mỗi CTDL đều được xây dựng theo đúng trình tự:

- Định nghĩa.
- Đặc tả.
- Phân tích các phương án hiện thực.
- Hiện thực.

2.1. Định nghĩa ngăn xếp

Với định nghĩa danh sách trong chương mở đầu, chúng ta hiểu rằng trong danh sách, mỗi phần tử, ngoại trừ phần tử cuối, đều có duy nhất một phần tử đứng sau nó. Ngăn xếp là một trường hợp của danh sách, được sử dụng trong các ứng dụng có liên quan đến sự đảo ngược. Trong CTDL ngăn xếp, việc thêm hay lấy dữ liệu chỉ được thực hiện tại một đầu. Dữ liệu thêm vào trước sẽ lấy ra sau, tính chất này còn được gọi là vào trước ra sau (*First In Last Out* - FILO).
Đầu thêm hay lấy dữ liệu của ngăn xếp còn gọi là đỉnh (*top*) của ngăn xếp.



Hình 2.1- Thêm phần tử vào và lấy phần tử ra khỏi ngăn xếp.

Vậy chúng ta có định nghĩa của ngăn xếp dưới đây, không khác gì đối với định nghĩa danh sách, ngoại trừ cách thức mà ngăn xếp cho phép thay đổi hoặc truy xuất đến các phần tử của nó.

Định nghĩa: Một ngăn xếp các phần tử kiểu T là một chuỗi nối tiếp các phần tử của T, kèm các tác vụ sau:

1. Tạo một đối tượng ngăn xếp rỗng.

2. Đẩy (`push`) một phần tử mới vào ngăn xếp, giả sử ngăn xếp chưa đầy (phần tử dữ liệu mới luôn được thêm tại đỉnh).

3. Lấy (`pop`) một phần tử ra khỏi ngăn xếp, giả sử ngăn xếp chưa rỗng (phần tử bị loại là phần tử đang nằm tại đỉnh).

4. Xem phần tử tại đỉnh ngăn xếp (`top`).

Lưu ý rằng định nghĩa này không quan tâm đến cách hiện thực của kiểu dữ liệu trừu tượng ngăn xếp. Chúng ta sẽ tìm hiểu một vài cách hiện thực khác nhau của ngăn xếp và tất cả chúng đều phù hợp với định nghĩa này.

2.2. Đặc tả ngăn xếp

Ngoài các tác vụ chính trên, các phương thức khác có thể bổ sung tuỳ vào nhu cầu mà chúng ta thấy cần thiết:

+ `empty`: cho biết ngăn xếp có rỗng hay không.
+ `full`: cho biết ngăn xếp có đầy hay chưa.
+ `clear`: xóa sạch tất cả dữ liệu và làm cho ngăn xếp trở nên rỗng.

Chúng ta lưu ý rằng, khi thiết kế các phương thức cho mỗi lớp CTDL, ngoài một số phương thức chính để thêm vào hay lấy dữ liệu ra, chúng ta có thể bổ sung thêm nhiều phương thức khác. Việc thêm dựa vào quan niệm của mỗi người về sự tiện dụng của lớp CTDL đó. Nhưng điều đặc biệt quan trọng ở đây là các phương thức đó không thể mâu thuẫn với định nghĩa ban đầu cũng như các chức năng mà chúng ta đã định ra cho lớp. Chẳng hạn, trong trường hợp ngăn xếp của chúng ta, để bảo đảm quy luật "Vào trước ra sau" thì trật tự của các phần tử trong ngăn xếp là rất quan trọng. Chúng ta không thể cho chúng một phương thức có thể thay đổi trật tự của các phần tử đang có, hoặc phương thức lấy một phần tử tại một vị trí bất kỳ nào đó của ngăn xếp.

Trên đây là những phương thức liên quan đến các thao tác dữ liệu trên ngăn xếp.

Đối với bất kỳ lớp CTDL nào, chúng ta cũng không thể không xem xét đến hai phương thức cực kỳ quan trọng: đó chính là hai hàm dựng lớp và hủy lớp: ***constructor*** và ***destructor***. Trong C++ các hàm *constructor* và *destructor* được

trình biên dịch gọi khi một đối tượng vừa được tạo ra hoặc sắp bị hủy. Chúng ta cần bảo đảm cho mỗi đối tượng CTDL được tạo ra có trạng thái ban đầu là hợp lệ. Sự hợp lệ này sẽ tiếp tục được duy trì bởi các phương thức thao tác dữ liệu bên trên.

Trạng thái ban đầu hợp lệ là trạng thái rỗng không chứa dữ liệu nào hoặc trạng thái đã chứa một số dữ liệu theo như mong muốn của người lập trình sử dụng CTDL. Như vậy, mỗi lớp CTDL luôn có một *constructor* mặc định (không có thông số) để tạo đối tượng rỗng, các *constructor* có thông số khác chúng ta có thể thiết kế bổ sung nếu thấy hợp lý và cần thiết.

Một đối tượng CTDL khi bị hủy phải đảm bảo không để lại rác trong bộ nhớ. Chúng ta đã biết rằng, với các biến cấp phát tĩnh, trình biên dịch sẽ tự lấy lại những vùng nhớ đã cấp phát cho chúng. Với các biến cấp phát động thì ngược lại, vùng nhớ phải được chương trình giải phóng khi không còn sử dụng đến. Như vậy, đối với mỗi phương án hiện thực cụ thể cho mỗi lớp CTDL mà có sử dụng vùng nhớ cấp phát động, chúng ta sẽ xây dựng *destructor* cho nó để lo việc giải phóng vùng nhớ trước khi đối tượng bị hủy.

Trong C++, *constructor* có cùng tên với lớp và không có kiểu trả về. *Constructor* của một lớp được gọi một cách tự động khi một đối tượng của lớp đó được khai báo.

Đặc tả *constructor* cho lớp ngăn xếp, mà chúng ta đặt tên là lớp `Stack`, như sau:

```
template <class Entry>
Stack<Entry>::Stack();
```
pre: không có.
post: đối tượng ngăn xếp vừa được tạo ra là rỗng.
uses: không có.

Để đặc tả tiếp cho các phương thức khác, chúng ta chọn ra các trị của `ErrorCode` đủ để sử dụng cho lớp `Stack` là:

`success, overflow, underflow`

Các thông số dành cho các phương thức dưới đây được thiết kế tùy vào chức năng và nhu cầu của từng phương thức.

Phương thức loại một phần tử ra khỏi ngăn xếp:

```
template <class Entry>
ErrorCode Stack<Entry>::pop();
```
pre: không có.
post: nếu ngăn xếp không rỗng, phần tử tại đỉnh ngăn xếp được lấy đi, `ErrorCode` trả về là `success`; nếu ngăn xếp rỗng, `ErrorCode` trả về là `underflow`, ngăn xếp không đổi.
uses: không có.

Phương thức thêm một phần tử dữ liệu vào ngăn xếp:

```
template <class Entry>
ErrorCode Stack<Entry>::push(const Entry &item);
```

pre: không có.

post: nếu ngăn xếp không đầy, `item` được thêm vào trên đỉnh ngăn xếp, `ErrorCode` trả về là `success`; nếu ngăn xếp đầy, `ErrorCode` trả về là `overflow`, ngăn xếp không đổi.

uses: không có.

Lưu ý rằng `item` trong thông số của `push` đóng vai trò là tham trị nên được khai báo là tham chiếu hằng (*const reference*). Trong khi đó trong phương thức `top, item` là tham biến.

Hai phương thức `top` và `empty` được khai báo **`const`** vì chúng không làm thay đổi ngăn xếp.

```
template <class Entry>
ErrorCode Stack<Entry>:: top(Entry &item) const;
```

pre: không có

post: nếu ngăn xếp không rỗng, phần tử tại đỉnh ngăn xếp được chép vào `item`, `ErrorCode` trả về là `success`; nếu ngăn xếp rỗng, `ErrorCode` trả về là `underflow`; cả hai trường hợp ngăn xếp đều không đổi.

uses: không có.

```
template <class Entry>
bool Stack<Entry>::empty() const;
```

pre: không có

post: nếu ngăn xếp rỗng, hàm trả về `true`; nếu ngăn xếp không rỗng, hàm trả về `false`, ngăn xếp không đổi.

uses: không có.

Sinh viên có thể đặc tả tương tự cho phương thức **`full, clear`**, hay các phương thức bổ sung khác.

Từ nay về sau, chúng ta quy ước rằng nếu hai phần *precondition* hoặc *uses* không có thì chúng ta không cần phải ghi ra.

Chúng ta có phần giao tiếp mà lớp `Stack` dành cho người lập trình sử dụng như sau:

```
template<class Entry>
class Stack {
public:
   Stack();
   bool empty() const;
   ErrorCode pop();
   ErrorCode top(Entry &item) const;
   ErrorCode push(const Entry &item);
};
```

Với một đặc tả như vậy chúng ta đã hoàn toàn có thể sử dụng lớp Stack trong các ứng dụng. Sinh viên nên tiếp tục xem đến phần trình bày các ứng dụng về ngăn xếp trong chương 14. Dưới đây là chương trình minh họa việc sử dụng ngăn

xếp thông qua các đặc tả trên. Chương trình giải quyết bài toán in các số theo thứ tự ngược với thứ tự nhập vào đã được trình bày trong phần mở đầu.

Ví dụ:

Chương trình sẽ đọc vào một số nguyên n và n số thực kế đó. Mỗi số thực nhập vào sẽ được lưu vào ngăn xếp. Cuối cùng các số được lấy từ ngăn xếp và in ra.

```
#include   <Stack> //Sử dụng lớp Stack.
int main()
/*
pre:  Người sử dụng nhập vào một số nguyên n và n số thực.
post:  Các số sẽ được in ra theo thứ tự ngược.
uses:  lớp Stack và các phương thức của Stack.
*/
{
   int n;
   double item;
   Stack<double> numbers;
   cout <<"Type in an integer n followed by n decimal numbers."<< endl;
   cout << " The numbers will be printed in reverse order." << endl;
   cin  >> n;

   for (int i = 0; i < n; i++) {
      cin >> item;
      numbers.push(item);
   }

   cout << endl << endl;
   while (!numbers.empty()) {
      numbers.top(item)
      cout << item << " ";
      numbers.pop();
   }
   cout << endl;
}
```

Che dấu thông tin: khi sử dụng lớp `Stack` chúng ta không cần biết nó được lưu trữ trong bộ nhớ như thế nào và các phương thức của nó hiện thực ra sao. Đây là vấn đề che dấu thông tin (*information hiding*).

Một CTDL có thể có nhiều cách hiện thực khác nhau, nhưng mọi cách hiện thực đều có chung phần đặc tả các giao tiếp đối với bên ngoài. Nhờ đó mà các chương trình ứng dụng giữ được sự độc lập với các hiện thực khác nhau của cùng một lớp CTDL. Khi cần thay đổi hiện thực của CTDL mà ứng dụng đang sử dụng, chúng ta không cần chỉnh sửa mã nguồn của ứng dụng.

Tính khả thi và hiệu quả của ứng dụng: Tuy ứng dụng cần phải độc lập với hiện thực của cấu trúc dữ liệu, nhưng việc chọn cách hiện thực nào ảnh hưởng đến tính khả thi và hiệu quả của ứng dụng. Chúng ta cần hiểu các ưu nhược điểm của mỗi cách hiện thực của cấu trúc dữ liệu để lựa chọn cho phù hợp với tính chất của ứng dụng.

Tính trong sáng của chương trình: Ưu điểm khác của che dấu thông tin là tính trong sáng của chương trình. Những tên gọi quen thuộc dành cho các thao tác trên cấu trúc dữ liệu giúp chúng ta hình dung rõ ràng giải thuật của chương trình. Chẳng hạn với thao tác trên ngăn xếp, người ta thường quen dùng các từ: `push` – đẩy vào ngăn xếp, `pop` – lấy ra khỏi ngăn xếp.

Thiết kế từ trên xuống: Sự tách rời giữa việc sử dụng cấu trúc dữ liệu và cách hiện thực của nó còn giúp chúng ta thực hiện tốt hơn quá trình thiết kế từ trên xuống (*top-down design*) cả cho cấu trúc dữ liệu và cả cho chương trình ứng dụng.

2.3. Các phương án hiện thực ngăn xếp

Trong phần này chúng ta sẽ tìm hiểu các phương án hiện thực cho lớp ngăn xếp. Các ưu nhược điểm của các cách hiện thực khác nhau đối với một đặc tả CTDL thường liên quan đến những vấn đề sau đây:

- Cho phép hay không cho phép lưu trữ và thao tác với lượng dữ liệu lớn.
- Tốc độ xử lý của các phương thức.

Vì ngăn xếp là một trường hợp đặc biệt của danh sách, nên đã đến lúc chúng ta bàn đến cách lưu trữ các phần tử trong một danh sách. Có hai phương án lưu trữ chính:

- Các phần tử nằm kế nhau trong bộ nhớ mà chúng ta hay dùng từ liên tục (*contiguous*) để nói đến.
- Các phần tử không nằm kế nhau trong bộ nhớ mà chúng ta dùng công cụ là các biến kiểu con trỏ (*pointer*) để quản lý, trường hợp này chúng ta gọi là danh sách liên kết (*linked list*).

Hiện thực liên tục đơn giản nhưng bị hạn chế ở chỗ kích thước cần được biết trước. Nếu dùng mảng lớn quá sẽ bị lãng phí, nhưng nhỏ quá dễ bị đầy. Hiện thực liên kết linh động hơn, nó chỉ bị đầy khi bộ nhớ thực sự không còn chỗ trống nữa.

2.4. Hiện thực ngăn xếp

2.4.1. Hiện thực ngăn xếp liên tục

Để hiện thực lớp ngăn xếp liên tục (*contiguous stack*), chúng ta dùng một mảng (*array* trong C++) để chứa các phần tử của ngăn xếp và một thuộc tính `count` cho biết số phần tử hiện có trong ngăn xếp.

```
const int max = 10;    // Dùng số nhỏ để kiểm tra chương trình.
template <class Entry>
class Stack {
public:
   Stack();
```

```
   bool empty() const;
   ErrorCode pop();
   ErrorCode top(Entry &item) const;
   ErrorCode push(const Entry &item);
private:
   int count;
   Entry entry[max];
};
```

Push, pop, và các phương thức khác

Ý tưởng chung của các phương thức này như sau:

- Việc thêm dữ liệu mới chỉ thực hiện được khi ngăn xếp còn chỗ trống.
- Việc loại phần tử khỏi ngăn xếp hoặc xem phần tử trên đỉnh ngăn xếp chỉ thực hiện được khi ngăn xếp không rỗng.
- Do `count` là số phần tử hiện có trong ngăn xếp và chỉ số của *array* trong C++ được bắt đầu từ 0, nên `count-1` chính là chỉ số của phần tử tại đỉnh ngăn xếp khi cần xem hoặc cần loại bỏ khỏi ngăn xếp.
- Khi cần thêm phần tử mới, `count` là chỉ số chỉ đến vị trí còn trống ngay trên đỉnh ngăn xếp, cũng là chỉ số của phần tử mới nếu được thêm vào.
- Khi ngăn xếp được thêm hoặc lấy phần tử thì thuộc tính `count` luôn phải được cập nhật lại.
- *Constructor* tạo đối tượng ngăn xếp rỗng bằng cách gán thuộc tính `count` bằng 0. Lưu ý rằng chúng ta không cần quan tâm đến trị của những phần tử nằm từ vị trí `count` cho đến hết mảng (`max -1`), các vị trí từ 0 đến `count-1` mới thực sự chứa những dữ liệu đã được đưa vào ngăn xếp.

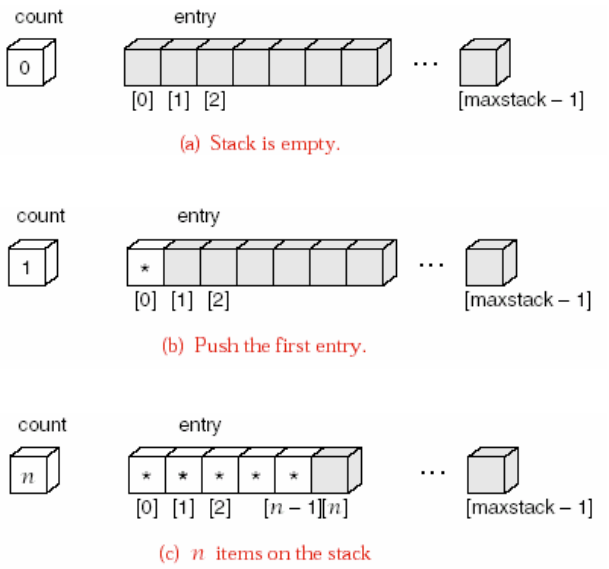
```
template <class Entry>
ErrorCode Stack<Entry>::push(const Entry &item)
/*
post: nếu ngăn xếp không đầy, item được thêm vào trên đỉnh ngăn xếp, ErrorCode trả về là
      success; nếu ngăn xếp đầy, ErrorCode trả về là overflow, ngăn xếp không đổi.
*/
{
   ErrorCode outcome = success;
   if (count >= max)
      outcome = overflow;
   else
      entry[count++] = item;
   return outcome;
}

template <class Entry>
ErrorCode Stack<Entry>::pop()
/*
post: nếu ngăn xếp không rỗng, phần tử tại đỉnh ngăn xếp được lấy đi, ErrorCode trả về là
      success; nếu ngăn xếp rỗng, ErrorCode trả về là underflow, ngăn xếp không đổi.
*/
{
   ErrorCode outcome = success;
   if (count == 0)
      outcome = underflow;
```

```
   else --count;
   return outcome;
}
```

```
template <class Entry>
ErrorCode Stack<Entry>::top(Entry &item) const
/*
post: nếu ngăn xếp không rỗng, phần tử tại đỉnh ngăn xếp được chép vào item, ErrorCode trả
      về là success; nếu ngăn xếp rỗng, ErrorCode trả về là underflow; cả hai trường hợp
      ngăn xếp  đều không đổi.
*/
{
   ErrorCode outcome = success;
   if (count == 0)
      outcome = underflow;
   else
      item = entry[count - 1];
   return outcome;
}
```

```
template <class Entry>
bool Stack<Entry>::empty() const
/*
post: nếu ngăn xếp rỗng, hàm trả về true; nếu ngăn xếp không rỗng, hàm trả về false, ngăn
      xếp không đổi.
*/
{
   bool outcome = true;
   if (count > 0) outcome = false;
   return outcome;
}
```



Hình 2.2- Biểu diễn của dữ liệu trong ngăn xếp liên tục.

Constructor sẽ khởi tạo một ngăn xếp rỗng.

```
template <class Entry>
Stack<Entry>::Stack()
/*
post: ngăn xếp được khởi tạo rỗng.
*/
{
   count = 0;
}
```

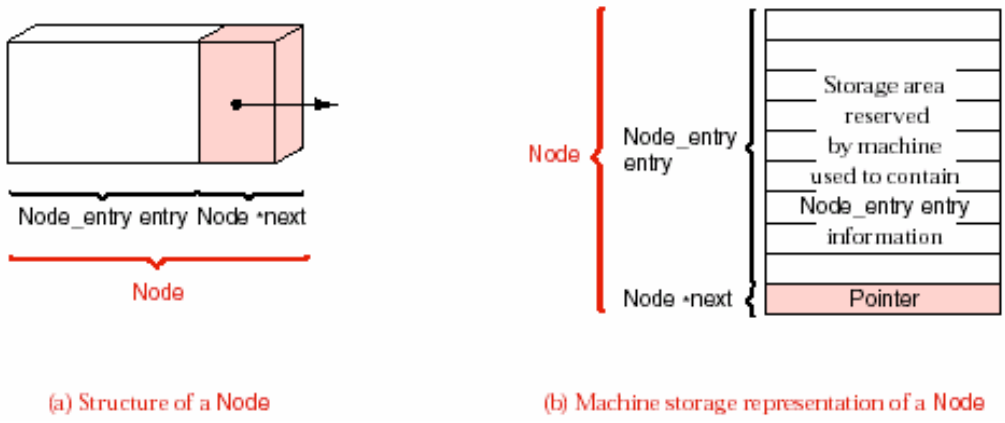
2.4.2. Hiện thực ngăn xếp liên kết

Cấu trúc liên kết được tạo bởi các phần tử , mỗi phần tử chứa hai phần: một chứa thông tin chính là dữ liệu của phần tử, một chứa con trỏ tham chiếu đến phần tử kế, và được khai báo trong C++ như sau:

```
template <class Entry>
struct Node {
//  data members
   Entry entry;
   Node<Entry> *next;
//  constructors
   Node();
   Node(Entry item, Node<Entry> *add_on = NULL);
};
```

Và đây là hình ảnh của một phần tử trong cấu trúc liên kết:

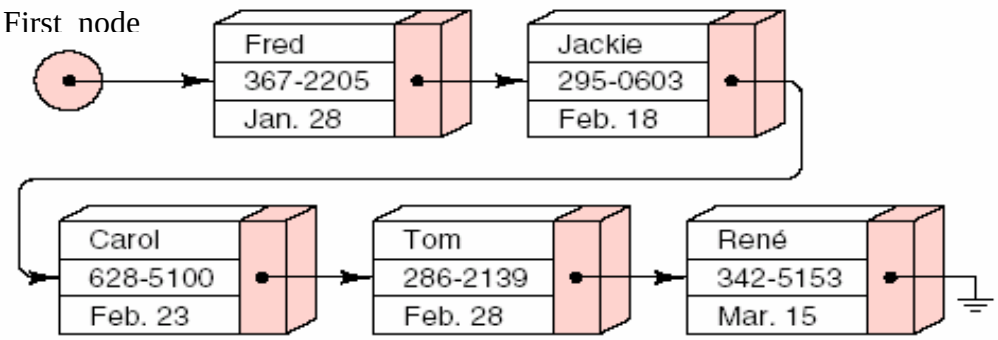
Hình dưới đây biểu diễn một cấu trúc liên kết có con trỏ chỉ đến phần tử đầu là `First_node`.



Hình 2.3- Cấu trúc `Node` chứa con trỏ

Vấn đề đặt ra là chúng ta nên chọn phần tử đầu hay phần tử cuối của cấu trúc liên kết làm đỉnh của ngăn xếp. Thoạt nhìn, dường như việc thêm một `node` mới vào cuối cấu trúc liên kết là dễ hơn (tương tự như đối với ngăn xếp liên tục). Nhưng việc loại một phần tử ở cuối là khó bởi vì việc xác định phần tử ngay trước

phần tử bị loại không thể thực hiện nhanh chóng. Lý do là các con trỏ trong cấu trúc liên kết chỉ theo một chiều. Khi loại đi một phần tử ở cuối cấu trúc liên kết, chúng ta phải bắt đầu từ đầu, lần theo các con trỏ, mới xác định được phần tử cuối. Do đó, cách tốt nhất là việc thêm vào hay loại phần tử đều được thực hiện ở phần tử đầu của cấu trúc liên kết. Đỉnh của ngăn xếp liên kết được chọn là phần tử đầu của cấu trúc liên kết.



Hình 2.4- Cấu trúc liên kết

Mỗi cấu trúc liên kết cần một thành phần con trỏ chỉ đến phần tử đầu tiên. Đối với ngăn xếp liên kết, thành phần này luôn chỉ đến đỉnh của ngăn xếp. Do mỗi phần tử trong cấu trúc liên kết có tham chiếu đến phần tử kế nên từ phần tử đầu tiên chúng ta có thể đến mọi phần tử trong ngăn xếp liên kết bằng cách lần theo các tham chiếu này. Từ đó, thông tin duy nhất cần thiết để có thể truy xuất dữ liệu trong ngăn xếp liên kết là địa chỉ của phần tử tại đỉnh. Phần tử tại đáy ngăn xếp luôn có tham chiếu là `NULL`.

```
template <class Entry>
class Stack {
public:
   Stack();
   bool empty() const;
   ErrorCode push(const Entry &item);
   ErrorCode pop();
   ErrorCode top(Entry &item) const;
protected:
   Node<Entry> *top_node;
};
```

Do lớp `Stack` giờ đây chỉ chứa một phần tử dữ liệu, chúng ta có thể cho rằng việc dùng lớp có thể không cần thiết, thay vào đó chúng ta dùng luôn một biến để chứa địa chỉ của đỉnh ngăn xếp. Tuy nhiên, ở đây có bốn lý do để sử dụng lớp `Stack`.

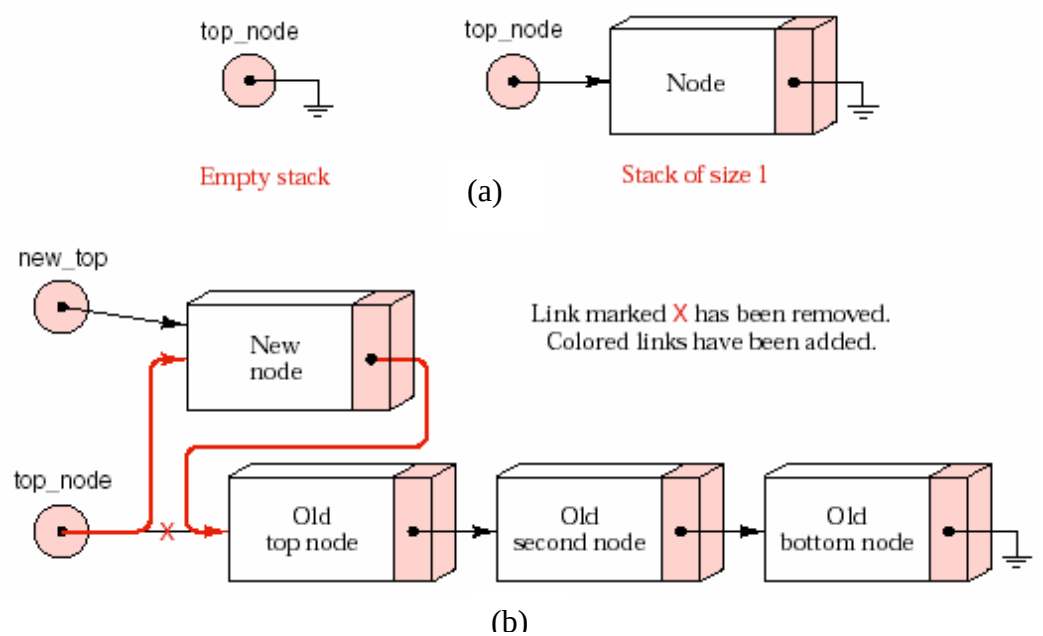
- Lý do quan trọng nhất là sự duy trì tính đóng kín: nếu chúng ta không sử dụng lớp ngăn xếp, chúng ta không thể xây dựng các phương thức cho ngăn xếp.

- Lý do thứ hai là để duy trì sự khác biệt luận lý giữa lớp ngăn xếp, mà bản thân được tạo bởi các phần tử là các `node`, với `top` của ngăn xếp là một con trỏ tham chiếu đến chỉ một `node`. Việc chúng ta chỉ cần nắm giữ `top` của ngăn xếp, là có thể tìm đến mọi phần tử khác của ngăn xếp tuy là hiển nhiên, nhưng không thích đáng với cấu trúc luận lý này.
- Lý do thứ ba là để duy trì tính nhất quán với các cấu trúc dữ liệu khác cũng như các cách hiện thực khác nhau của một cấu trúc dữ liệu: một cấu trúc dữ liệu bao gồm các dữ liệu và một tập các thao tác.
- Cuối cùng, việc xem ngăn xếp như một con trỏ đến đỉnh của nó không được phù hợp với các kiểu dữ liệu. Thông thường, các kiểu dữ liệu phải có khả năng hỗ trợ trong việc *debug* chương trình bằng cách cho phép trình biên dịch thực hiện việc kiểm tra kiểu một cách tốt nhất.

Chúng ta hãy bắt đầu bằng một ngăn xếp rỗng, top_node == NULL, và xem xét việc thêm phần tử đầu tiên vào ngăn xếp. Chúng ta cần tạo một node mới chứa bản sao của thông số item nhận vào bở phương thức push. Node này được truy xuất bởi biến con trỏ new_top. Sau đó địa chỉ chứa trong new_top sẽ được chép vào top_node của Stack (hình 2.5a):

```
Node *new_top = new Node<Entry>(item);
top_node = new_top;
```

Chú ý rằng ở đây, *constructor* khi tạo một `node` mới đã gán `next` của nó bằng `NULL`, và chúng ta hoàn toàn an tâm vì không bao giờ có con trỏ mang trị ngẫu nhiên.



Hình 2.5- Thêm một phần tử vào ngăn xếp liên kết.

Nếu trung thành với nguyên tắc "Không bao giờ để một biến con trỏ mang trị ngẫu nhiên", chúng ta sẽ giảm được gánh nặng đáng kể trong công sức lập trình vì không phải mất quá nhiều thì giờ và đau đầu do những lỗi mà nó gây ra.

Để tiếp tục, xem như chúng ta đã có một ngăn xếp không rỗng. Để đưa thêm phần tử vào ngăn xếp, chúng ta cần thêm một `node` vào ngăn xếp. Trước hết chúng ta cần tạo một `node` mới được tham chiếu bởi con trỏ `new_top`, `node` này phải có dữ liệu là `item` và liên kết `next` tham chiếu đến `top` cũ của ngăn xếp. Sau đó chúng ta sẽ thay đổi `top_node` của ngăn xếp tham chiếu đến `node` mới này (hình 2.5b). Thứ tự của hai phép gán này rất quan trọng: nếu chúng ta làm theo thứ tự ngược lại, việc thay đổi `top_node` sớm sẽ làm mất khả năng truy xuất các phần tử đã có của ngăn xếp. Chúng ta có phương thức `push` như sau:

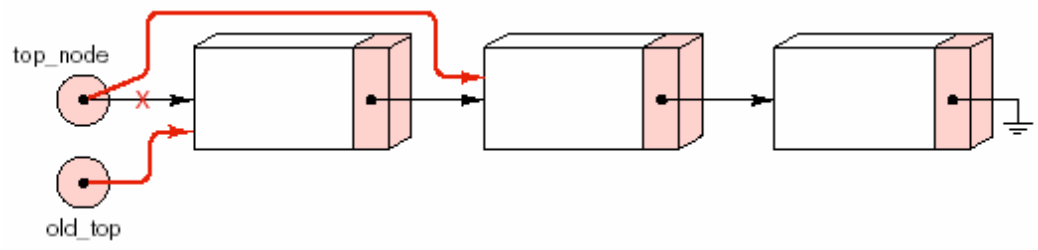
```
template <class Entry>
ErrorCode Stack<Entry>::push(const Entry &item)
/*
post: nếu ngăn xếp không đầy, item được thêm vào trên đỉnh ngăn xếp, ErrorCode trả về là
      success; nếu ngăn xếp đầy, ErrorCode trả về là overflow, ngăn xếp không đổi.
*/
{
   Node *new_top = new Node<Entry>(item, top_node);
   if (new_top == NULL) return overflow;
   top_node = new_top;
   return success;
}
```

Khi thay đổi các tham chiếu (các biến con trỏ), thứ tự các phép gán luôn cần được xem xét một cách kỹ lưỡng.

.

Phương thức `push` trả về `ErrorCode` là `overflow` trong trường hợp bộ nhớ động không tìm được chỗ trống để cấp phát cho phần tử mới, toán tử `new` trả về trị `NULL`.

Việc lấy một phần tử ra khỏi ngăn xếp thực sự đơn giản:



Hình 2.6- Lấy một phần tử ra khỏi ngăn xếp liên kết.

```
template <class Entry>
ErrorCode Stack<Entry>::pop()
/*
post: nếu ngăn xếp không rỗng, phần tử tại đỉnh ngăn xếp được lấy đi, ErrorCode trả về là
      success; nếu ngăn xếp rỗng, ErrorCode trả về là underflow, ngăn xếp không đổi.
*/
{
   Node *old_top = top_node;
   if (top_node == NULL) return underflow;
   top_node = old_top->next;
   delete old_top;
   return success;
}
```

Lưu ý rằng trong phương thức pop, chỉ cần gán top_node của ngăn xếp tham chiếu đến phần tử thứ hai trong ngăn xếp thì phần tử thứ nhất xem như đã được loại khỏi ngăn xếp. Tuy nhiên, nếu không thực hiện việc giải phóng phần tử trên đỉnh ngăn xếp, chương trình sẽ gây ra rác. Trong ứng dụng nhỏ, phương thức pop vẫn chạy tốt. Nhưng nếu ứng dụng lớn gọi phương thức này rất nhiều lần, số lượng rác sẽ lớn lên đáng kể dẫn đến không đủ vùng nhớ để chương trình chạy tiếp.

Khi một cấu trúc dữ liệu được hiện thực, nó phải được xử lý tốt trong mọi trường hợp để có thể được sử dụng trong nhiều ứng dụng khác nhau.

2.4.3. Ngăn xếp liên kết với sự an toàn

Khi sử dụng các phương thức mà chúng ta vừa xây dựng cho ngăn xếp liên kết, người lập trình có thể vô tình gây nên rác hoặc phá vỡ tính đóng kín của các đối tượng ngăn xếp. Trong phần này chúng ta sẽ xem xét chi tiết về các nguy cơ làm mất đi tính an toàn và tìm hiểu thêm ba phương thức mà C++ cung cấp để khắc phục vấn đề này, đó là các tác vụ hủy đối tượng (***destructor***), tạo đối tượng bằng cách sao chép từ đối tượng khác (***copy constructor***) và phép gán được định nghĩa lại (***overloaded assignment***). Hai tác vụ đầu không được gọi tường minh bởi người lập trình, chúng sẽ được trình biên dịch gọi lúc cần thiết; riêng tác vụ thứ ba được gọi bởi người lập trình khi cần gán hai đối tượng. Như vậy, việc bổ sung nhằm bảo đảm tính an toàn cho lớp Stack không làm thay đổi vẻ bề ngoài của Stack đối với người sử dụng.

2.4.3.1. *Hàm hủy đối tượng (Destructor)*

Giả sử như người lập trình viết một vòng lặp đơn giản trong đó khai báo một đối tượng ngăn xếp có tên là small và đưa dữ liệu vào. Chẳng hạn chúng ta xem xét đoạn lệnh sau:

```
for (int i=0; i < 1000000; i++) {
   Stack<Entry> small;
   small.push(some_data);
}
```

Trong mỗi lần lặp, đối tượng `small` được tạo ra, dữ liệu thêm vào thuộc vùng bộ nhớ cấp phát động, sau đó đối tượng `small` không còn tồn tại khi ra khỏi tầm vực hoạt động của nó (*scope*). Giả sử chương trình sử dụng ngăn xếp liên kết được hiện thực như hình 2.4. Ngay khi đối tượng `small` không còn tồn tại , dữ liệu trong ngăn xếp trở thành rác, vì bản thân đối tượng `small` chỉ chứa con trỏ `top_node`, vùng nhớ mà con trỏ này chiếm sẽ được trả về cho hệ thống, còn các dữ liệu mà con trỏ này tham chiếu đến thuộc vùng nhớ cấp phát động vẫn chưa được trả về hệ thống. Vòng lặp trên được thực hiện hàng triệu lần, và rác sẽ bị tích lũy rất nhiều. Trong trường hợp này không thể buộc tội người lập trình: do vòng lặp sẽ chẳng gây ra vấn đề gì nếu người lập trình sử dụng hiện thực ngăn xếp liên tục, mọi vùng nhớ dành cho dữ liệu trong ngăn xếp liên tục đều được giải phóng khi ngăn xếp ra khỏi tầm vực.

Một điều chắc chắn rằng khi hiện thực ngăn xếp liên kết, chúng ta cần phải cảnh báo người sử dụng không được để một đối tượng ngăn xếp không rỗng ra khỏi tầm vực, hoặc chúng ta phải làm rỗng ngăn xếp trước khi nó ra khỏi tầm vực.

Ngôn ngữ C++ cung cấp cho lớp phương thức ***destructor*** để giải quyết vấn đề này. Đối với mọi lớp, *destructor* là một phương thức đặc biệt được thực thi cho đối tượng của lớp ngay trước khi đối tượng ra khỏi tầm vực. Người sử dụng không cần phải gọi *destructor* một cách tường minh và thậm chí cũng không cần biết đến sự tồn tại của nó. Đối với người sử dụng, một lớp có *destructor* có thể được thay thế một cách đơn giản bởi một lớp mà không có nó.

Destructor thường được sử dụng để giải phóng các đối tượng cấp phát động mà chúng có thể tạo nên rác. Trong trường hợp của chúng ta, chúng ta nên bổ sung thêm *destructor* cho lớp ngăn xếp liên kết. Sau hiệu chỉnh này, người sử dụng sẽ không thể gây ra rác khi để một đối tượng ngăn xếp không rỗng ra khỏi tầm vực.

Destructor được khai báo như một phương thức của lớp, không có thông số và không có trị trả về. Tên của *destructor* là tên lớp có thêm dấu ~ phía trước.

```
Stack::~Stack();
```

Do phương thức `pop` được lập trình để loại một phần tử ra khỏi ngăn xếp, chúng ta có thể hiện thực *destructor* cho ngăn xếp bằng cách lặp nhiều lần việc gọi `pop`:

```
template <class Entry>
Stack::~Stack() //  Destructor
/*
post: ngăn xếp đã được làm rỗng.
*/
{
   while (!empty())
      pop();
}
```

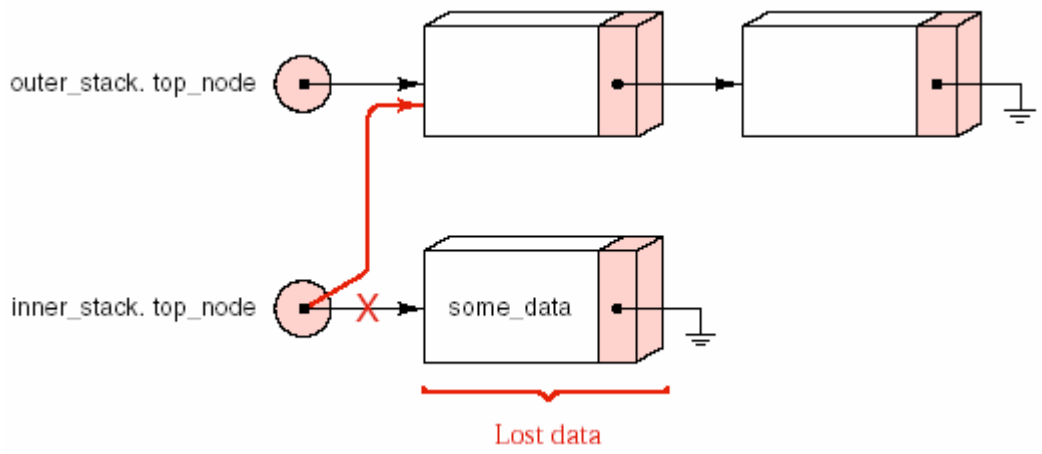
Đối với mọi cấu trúc liên kết chúng ta cần viết destructor để dọn dẹp các đối tượng trước khi chúng ra khỏi tầm vực.

2.4.3.2. *Định nghĩa lại phép gán*

Ngay khi chúng ta đã bổ sung *destructor* cho ngăn xếp liên kết, người sử dụng cũng có thể tạo rác khi viết vòng lặp tựa như ví dụ sau.

```
Stack<Entry> outer_stack;
for (int i=0; i < 1000000; i++) {
   Stack<Entry> inner_stack;
   inner_stack.push(some_data);
   inner_stack = outer_stack;
}
```

Đầu tiên là đối tượng outer_stack được tạo ra, sau đó vòng lặp được thực hiện. Mỗi lần lặp là một lần tạo một đối tượng inner_stack, đưa dữ liệu vào inner_stack, gán outer_stack vào inner_stack. Lệnh gán này gây ra một vấn đề nghiêm trọng cho hiện thực ngăn xếp liên kết của chúng ta. Thông thường, C++ thực hiện phép gán các đối tượng bằng cách chép các thuộc tính của các đối tượng. Do đó, outer_stack.top_node được ghi đè lên inner_stack.top_node, làm cho dữ liệu cũ tham chiếu bởi inner_stack.top_node trở thành rác. Cũng giống như phần trước, nếu hiện thực ngăn xếp liên tục được sử dụng thì không có vấn đề gì xảy ra. Như vậy, lỗi là do hiện thực ngăn xếp liên kết còn thiếu sót.



Hình 2.7- Ứng dụng chép ngăn xếp.

Hình 2.7 cho thấy tác vụ gán không được thực hiện như mong muốn. Sau phép gán, cả hai ngăn xếp cùng chia xẻ các `node`. Cuối mỗi lần lặp, *destructor* của `inner_stack` sẽ giải phóng mọi dữ liệu của `outer_stack`. Việc giải phóng dữ liệu của `outer_stack` còn làm cho `outer_stack.top_node` trở thành tham chiếu treo, có nghĩa là tham chiếu đến vùng nhớ không xác định.

Vấn đề sinh ra bởi phép gán trên ngăn xếp liên kết là do nó chép các tham chiếu chứ không phải chép các trị. Phép gán trong trường hợp này được gọi là phép gán có ngữ nghĩa tham chiếu (*reference semantics*) . Ngược lại, khi phép gán thực sự chép dữ liệu trong CTDL chúng ta gọi là phép gán có ngữ nghĩa trị (*value semantics*). Trong hiện thực lớp ngăn xếp liên kết, hoặc chúng ta phải cảnh báo cho người sử dụng rằng phép gán chỉ là phép gán có ngữ nghĩa tham chiếu, hoặc chúng ta phải làm cho trình biên dịch C++ thực hiện phép gán có ngữ nghĩa trị.

Trong C++, chúng ta hiện thực một phương thức đặc biệt gọi là phép gán được định nghĩa lại (*overloaded assignment*) để định nghĩa lại cách thực hiện phép gán. Khi trình biên dịch C++ dịch một phép gán có dạng x = y, nó ưu tiên chọn phép gán được định nghĩa lại trước nếu như lớp x có định nghĩa. Chỉ khi không có phương thức này, trình biên dịch mới dịch phép gán như là phép gán từng bit đối với các thuộc tính của đối tượng (nếu thuộc tính là con trỏ thì phép gán trở thành phép gán có ngữ nghĩa tham chiếu). Chúng ta cần định nghĩa lại để phép gán cho ngăn xếp liên kết trở thành phép gán có ngữ nghĩa trị.

Có một vài cách để khai báo và hiện thực phép gán được định nghĩa lại. Cách đơn giản là khai báo như sau:

```
void Stack<Entry>::operator= ( const Stack &original );
```

Phép gán này có thể được gọi như sau:

```
x.operator = (y);
```

hoặc gọi theo cú pháp thường dùng:

```
x = y;
```

Phép gán định nghĩa lại cho ngăn xếp liên kết cần làm những việc sau:

- Chép dữ liệu từ ngăn xếp được truyền vào thông qua thông số.
- Giải phóng vùng nhớ chiếm giữ bởi dữ liệu của đối tượng ngăn xếp đang được thực hiện lệnh gán.
- Chuyển các dữ liệu vừa chép được cho đối tượng ngăn xếp được gán.

```
template <class Entry>
void Stack::operator = (const Stack<Entry> &original) //  Overload assignment
/*
post: đối tượng ngăn xếp được gán chứa các dữ liệu giống hệt ngăn xếp được truyền vào qua
      thông số.
*/
{
   Node<Entry> *new_top, *new_copy, *original_node = original.top_node;
   if (original_node == NULL) new_top = NULL;
   else {                          //  Tạo bản sao các node.
      new_copy = new_top = new Node<Entry>(original_node->entry);
```

```
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node<Entry>(original_node->entry);
         new_copy = new_copy->next;
      }
   }
   while (!empty())                  // Làm rỗng ngăn xếp sẽ được gán.
      pop();
   top_node = new_top;               // Ngăn xếp được gán sẽ nắm giữ bản sao.
}
```

Lưu ý rằng trong phương thức trên chúng ta tạo ra một bản sao từ ngăn xếp `original` trước, rồi mới dọn dẹp ngăn xếp sẽ được gán bằng cách gọi phương thức `pop` nhiều lần. Nhờ vậy nếu trong ứng dụng có phép gán `x = x` thì dữ liệu cũng không bị sai.

Phép gán được định nghĩa lại như trên thỏa yêu cầu là phép gán có ngữ nghĩa trị, tuy nhiên để có thể sử dụng trong trường hợp: `first_stack=second_stack=third_stack=...`, phép gán phải trả về **`Stack&`** (tham chiếu đến lớp `Stack`).

2.4.3.3. *Copy constructor*

Trường hợp cuối cùng về sự thiếu an toàn của các cấu trúc liên kết là khi trình biên dịch cần chép một đối tượng. Chẳng hạn, một đối tượng cần được chép khi nó là **tham trị** gởi cho một hàm. Trong C++, tác vụ chép mặc nhiên là chép các thuộc tính thành phần của lớp. Cũng giống như minh họa trong hình 2.7, tác vụ chép mặc nhiên này sẽ dẫn đến việc các đối tượng cùng chia xẻ dữ liệu. Nói một cách khác, tác vụ chép mặc định có ngữ nghĩa tham chiếu khi đối tượng có thuộc tính kiểu con trỏ. Điều này làm cho người sử dụng có thể vô tình làm mất dữ liệu:

```
void destroy_the_stack (Stack<Entry> copy)
{
   …
}
int main()
{
   Stack<Entry> vital_data;
   destroy_the_stack(vital_data);
}
```

Hàm `destroy_the_stack` nhận một bản sao `copy` của `vital_data`. Bản sao này cùng chia xẻ dữ liệu với `vital_data`, do đó khi kết thúc hàm, *destructor* thực hiện trên bản sao `copy` sẽ làm mất luôn dữ liệu của `vital_data`.

C++ cho phép lớp có thêm phương thức ***copy constructor*** để tạo một đối tượng mới giống một đối tượng đã có. Nếu chúng ta hiện thực *copy constructor*

cho lớp Stack thì trình biên dịch C++ sẽ ưu tiên chọn tác vụ chép này thay cho tác vụ chép mặc định. Chúng ta cần hiện thực *copy constructor* để có được ngữ nghĩa trị.

Đối với mọi lớp, khai báo chuẩn cho *copy constructor* cũng giống như khai báo *constructor* nhưng có thêm thông số là tham chiếu hằng đến đối tượng của lớp:

```
Stack<Entry>::Stack ( const Stack<Entry> &original );
```

Do đối tượng gọi *copy constructor* là một đối tượng rỗng vừa được tạo mới nên chúng ta không phải lo dọn dẹp dữ liệu như trường hợp đối với phép gán được định nghĩa lại. Chúng ta chỉ việc chép node đầu tiên và sau đó dùng vòng lặp để chép tiếp các node còn lại.

```
template <class Entry>
Stack<Entry>::Stack(const Stack<Entry> &original) //  copy constructor
/*
post: đối tượng ngăn xếp vừa được tạo ra có dữ liệu giống với ngăn xếp original
*/
{
   Node<Entry> *new_copy, *original_node = original.top_node;
   if (original_node == NULL) top_node = NULL;
   else {                          //  Tạo bản sao cho các node.
      top_node = new_copy = new Node<Entry>(original_node->entry);
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node<Entry>(original_node->entry);
         new_copy = new_copy->next;
      }
  a(
}
```

Một cách tổng quát, đối với mọi lớp liên kết, hoặc chúng ta bổ sung *copy constructor*, hoặc chúng ta cảnh báo người sử dụng rằng việc chép đối tượng có ngữ nghĩa tham chiếu.

2.4.4. Đặc tả ngăn xếp liên kết đã hiệu chỉnh

Chúng ta kết thúc phần này bằng đặc tả đã được hiệu chỉnh dưới đây cho ngăn xếp liên kết. Phần đặc tả này có được mọi đặc tính an toàn mà chúng ta đã phân tích.

```
template <class Entry>
class Stack {
public:
//  Các phương thức chuẩn của lớp Stack:
   Stack();
   bool empty() const;
   ErrorCode push(const Entry &item);
   ErrorCode pop();
   ErrorCode top(Entry &item) const;
```

```
//  Các đặc tả đảm bảo tính an toàn cho cấu trúc liên kết:
   ~Stack();
   Stack(const Stack<Entry> &original);
   void operator =(const Stack<Entry> &original);
protected:
   Node<Entry> *top_node;
};
```

Trên đây là phần trình bày đầy đủ nhất về những yêu cầu cần có đối với ngăn xếp liên kết, nhưng nó cũng đúng với các cấu trúc liên kết khác. Trong các phần sau của giáo trình này sẽ không giải thích thêm về 3 tác vụ này nữa, sinh viên tự phải hoàn chỉnh khi hiện thực bất kỳ CTDL nào có thuộc tính kiểu con trỏ.