Chương 5 – CHUỖI KÝ TỰ

Trong phần này chúng ta sẽ hiện thực một lớp biểu diễn một chuỗi nối tiếp các ký tự. Ví dụ ta có các chuỗi ký tự: "Đây là một chuỗi ký tự", "Tên?" trong đó cặp dấu " " không phải là bộ phận của chuỗi ký tự. Một chuỗi ký tự rỗng được ký hiệu "". Chuỗi ký tự cũng là một danh sách các ký tự. Tuy nhiên, các tác vụ trên chuỗi ký tự có hơi đặc biệt và khác với các tác vụ trên một danh sách trừu tượng mà chúng ta đã định nghĩa, chúng ta sẽ không dẫn xuất lớp chuỗi ký tự từ một lớp `List` nào trước đây.

Trong các tác vụ thao tác trên chuỗi ký tự, tác vụ tìm kiếm là khó khăn nhất. Chúng ta sẽ tìm hiểu hai giải thuật tìm kiếm vào cuối chương này. Trong phần đầu, chúng ta đặc biệt quan tâm đến việc khắc phục tính thiếu an toàn của chuỗi ký tự trong ngôn ngữ C mà đa số người lập trình đã từng sử dụng. Do đó phần trình bày tiếp theo đây liên quan chặt chẽ đến ngôn ngữ C và C++.

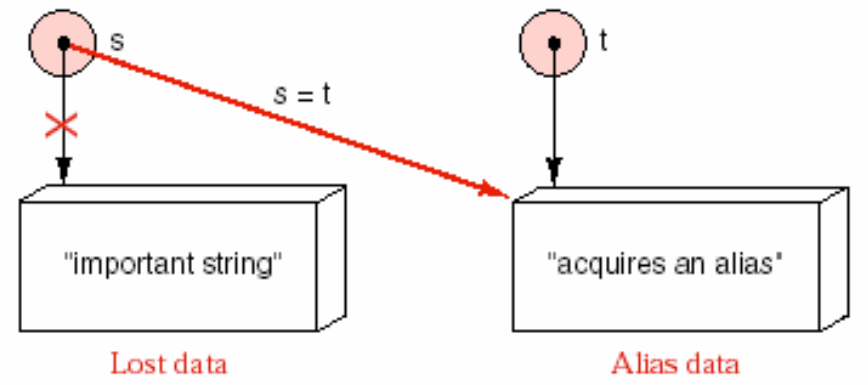
5.1. Chuỗi ký tự trong C và trong C++

Ngôn ngữ `C++` cung cấp hai cách hiện thực chuỗi ký tự. Cách nguyên thủy là hiện thực `string` của `C`. Giống như những phần khác, hiện thực `string` của ngôn ngữ `C` có thể chạy trong mọi hiện thực của `C++`. Chúng ta sẽ gọi các đối tượng `string` cung cấp bởi `C` là **`C-String.`** `C-String` thể hiện cả các điểm mạnh và cả các điểm yếu của ngôn ngữ `C`: chúng rất phổ biến, rất hiệu quả nhưng cũng rất hay bị dùng sai. `C-String` liên quan đến một loạt các tập quán mà chúng ta sẽ xem lại dưới đây.

Một `C-String` có kiểu **`char*`**. Do đó, một `C-String` tham chiếu đến một địa chỉ trong bộ nhớ; địa chỉ này là điểm bắt đầu của tập các bytes chứa các ký tự trong chuỗi ký tự. Vùng nhớ chiếm bởi một chuỗi ký tự phải được kết thúc bằng một ký tự đặc biệt **`'\0'`**. Trình biên dịch không thể kiểm tra giúp quy định này, sự thiếu sót sẽ gây lỗi thời gian chạy. Nói cách khác, `C-String` không có tính đóng kín và thiếu an toàn.

Tập tin chuẩn `<cstring>` chứa thư viện các hàm xử lý `C-String`. Trong các trình biên dịch `C++` cũ, tập tin này thường có tên là `<string.h>`. Các hàm thư viện này rất tiện lợi, hiệu quả và chứa hầu hết các tác vụ trên chuỗi ký tự mà chúng ta cần. Giả sử s và t là các `C-String`. Tác vụ `strlen(s)` trả về chiều dài của s, `strcmp(s,t)` so sánh từng ký tự của s và t, và `strstr(s,t)` trả về con trỏ tham chiếu đến vị trí bắt đầu của t trong s. Ngoài ra, trong `C++` tác vụ xuất `<<` được định nghĩa lại cho `C-String`, nhờ vậy, lệnh đơn giản `<< s` sẽ in chuỗi ký tự s.

Mặc dù hiện thực `C-String` có nhiều ưu điểm tuyệt vời, nhưng nó cũng có những nhược điểm nghiêm trọng. Thực vậy, nó có những vấn đề mà chúng ta đã gặp phải khi nghiên cứu CTDL ngăn xếp liên kết trong chương 2 cũng như các CTDL có chứa thuộc tính con trỏ nói chung. Thật dễ dàng khi người sử dụng có thể tạo bí danh cho chuỗi ký tự, cũng như gây nên rác. Trong hình 5.1, chúng ta thấy rõ phép gán s = t dẫn đến cả hai vấn đề trên.



Hình 5.1- Sự thiếu an toàn của `C-String`.

Một vấn đề khác cũng thường nảy sinh trong các ứng dụng có sử dụng `C-String`. Một `C-String` chưa khởi tạo cần được gán `NULL`. Tuy nhiên, rất nhiều hàm thư viện của `C-String` sẽ gặp sự cố trong thời gian chạy khi gặp đối tượng `C-String` là NULL. Chẳng hạn, lệnh

```
char* x = NULL;
cout << strlen(x);
```

được một số trình biên dịch chấp nhận, nhưng với nhiều hiện thực khác của thư viện `C-String`, thì gặp lỗi trong thời gian chạy. Do đó, người sử dụng phải kiểm tra kỹ lưỡng điều kiện trước khi gọi các hàm thư viện.

Trong `C++`, việc đóng gói `string` vào một lớp có tính đóng kín và an toàn được thực hiện dễ dàng. Thư viện chuẩn STL có lớp `String` an toàn chứa trong tập tin `<string>`. Thư viện này hiện thực lớp có tên `std::String` vừa tiện lợi, an toàn vừa hiệu quả.

Trong phần này chúng ta sẽ tự xây dựng một lớp `String` để có dịp hiểu kỹ về cách tạo nên một CTDL có tính đóng kín và an toàn cao. Chúng ta sẽ không phải viết lại toàn bộ mà chỉ sử dụng lại thư viện đã có `C-String`.

5.2. Đặc tả của lớp `String`

Để tạo một hiện thực lớp `String` an toàn, chúng ta đóng gói `C-String` như một thuộc tính thành phần của nó và để thuận tiện hơn, chúng ta thêm một thuộc tính chiều dài cho chuỗi ký tự. Do thuộc tính char* là một con trỏ, chúng ta cần thêm các tác vụ gán định nghĩa lại (*overloaded assignment*), *copy constructor*, *destructor*, để lớp `String` của chúng ta tránh được các vấn đề bí danh, tạo rác, hoặc việc sử dụng đối tượng mà chưa được khởi tạo.

5.2.1. Các phép so sánh

Với một số ứng dụng, sẽ hết sức thuận tiện nếu chúng ta bổ sung thêm các tác vụ so sánh <, >, <=, >=, ==, != để so sánh từng cặp đối tượng `String` theo từng ký tự. Vì thế, lớp `String` của chúng ta sẽ chứa các tác vụ so sánh được định nghĩa lại (*overloaded comparison operators*).

5.2.2. Một số *constructor* tiện dụng

Tạo đối tượng `String` từ một `C-String`

Chúng ta sẽ xây dựng *constructor* với thông số `char*` cho lớp `String`. *Constructor* này cung cấp một cách chuyển đổi thuận tiện một đối tượng `C-String` sang đối tượng `String`. Việc chuyển đổi thông qua cách gọi tường minh như sau:

```
String s(“some_string”);
```

Trong lệnh này, đối tượng `String` s được tạo ra chứa dữ liệu là `“some_string”.`

Constructor này đôi khi còn được gọi một cách không tường minh bởi trình biên dịch mỗi khi chương trình cần đến sự ép kiểu (*type cast*) từ kiểu `char*` sang `String`. Lấy ví dụ,

```
String s;
s = “some_string”;
```

Để chạy lệnh thứ hai, trình biên dịch `C++` trước hết gọi *constructor* của chúng ta để chuyển `“some_string”` thành một đối tượng `String` tạm. Sau đó phép gán định nghĩa lại của `String` được gọi để chép đối tượng tạm này vào s. Cuối cùng *destructor* cho đối tượng tạm được thực hiện.

Tạo đối tượng `String` từ một danh sách các ký tự

Tương tự, chúng ta cũng nên có *constructor* để chuyển một danh sách các ký tự sang một đối tượng `String`. Chẳng hạn, khi đọc một chuỗi ký tự từ người sử dụng, chúng ta nên đọc từng ký tự vào một danh sách các ký tự do chưa biết trước

chiều dài của nó. Sau đó chúng ta sẽ chuyển đổi danh sách này sang một đối tượng `String`.

Chuyển từ một đối tượng `String` sang một `C-String`

Cuối cùng, nếu có thể chuyển đổi ngược từ một đối tượng `String` sang một đối tượng `C-String` thì sẽ rất có lợi cho những trường hợp `string` cần được xem là `char*`. Đó là những lúc chúng ta cần sử dụng lại các hàm thư viện của `C-String` cho các đối tượng `String`. Phương thức này sẽ được gọi là `c_str()` và phải trả về `const char*` là một con trỏ tham chiếu đến dữ liệu biểu diễn `String`. Phương thức `c_str()` có thể được gọi như sau:

```
String s = “some_String”;
const char* new_s = s.c_str();
```

Điều quan trọng ở đây là `c_str()` trả về một `C-String` như là các ký tự hằng. Chúng ta có thể thấy được sự cần thiết này nếu chúng ta xem xét đến vùng nhớ chiếm bởi chuỗi ký tự `new_s`. Vùng nhớ này rõ ràng là thuộc đối tượng của lớp `String`. Chúng ta thấy rằng lớp `String` nên chịu trách nhiệm về vùng nhớ này, vì điều đó cho phép chúng ta hiện thực hàm chuyển đổi một cách hiệu quả, đồng thời tránh được cho người sử dụng khỏi phải chịu trách nhiệm về việc quên xóa một `C-String` đã được chuyển đổi từ một đối tượng `String`. Do đó, chúng ta khai báo `c_str()` trả về `const char*` để người sử dụng không thể sử dụng con trỏ trả về này mà thay đổi các ký tự dữ liệu được tham chiếu đến, sự thay đổi này chỉ thuộc quyền của lớp `String` mà thôi.

Với một số ít đặc tính được mô tả trên chúng ta có được một cách xử lý chuỗi ký tự vô cùng linh hoạt, hiệu quả và an toàn. Lớp `String` của chúng ta là một ADT đóng kín hoàn toàn, nhưng nó cung cấp một giao diện thật đầy đủ.

Chúng ta có đặc tả lớp `String` như sau:

```
class String {
public:
   String();
   ~String();
   String (const String &copy);   //  copy constructor
   String (const char * copy);    //  Chuyển đổi từ C-string
   String (List<char> &copy);     //  Chuyển đổi từ List các ký tự

   void operator =(const String &copy);
   const char *c_str() const;     //  Chuyển đổi sang C-string

protected:
   char *entries;
   int length;
};
```

```
bool operator ==(const String &first, const String &second);
bool operator >(const String &first, const String &second);
bool operator <(const String &first, const String &second);
bool operator >=(const String &first, const String &second);
bool operator <=(const String &first, const String &second);
bool operator !=(const String &first, const String &second);
```

5.3. Hiện thực lớp **String**

Các *constructor* chuyển đổi C-String và danh sách các ký tự sang đối tượng String:

```
String::String (const char *in_string)
/*
pre:  Con trỏ in_string tham chiếu đến một C-string.
post: Đối tượng String được khởi tạo từ chuỗi ký tự C-string in_string, và nó nắm giữ
      một bản sao của in_string, chuỗi ký tự trong in_string không thay đổi.
*/
{
   length = strlen(in_string);
   entries = new char[length + 1];
   strcpy(entries, in_string);
}
```

```
String::String (List<char> in_list)
/*
post: Đối tượng String được khởi tạo từ danh sách các ký tự trong đối tượng List, và nó nắm
      giữ một bản sao khác, đối tượng in_list không thay đổi.
*/
{
   length = in_list.size();
   entries = new char[length + 1];
   for (int i = 0; i < length; i++) in_list.retrieve(i,entries[i]);
   entries[length] = '\0';
}
```

Chúng ta chọn cách hiện thực phương thức chuyển đổi đối tượng String sang const char* như sau:

```
const char*String::c_str() const
/*
post: trả về con trỏ chỉ ký tự đầu tiên của chuỗi ký tự trong đối tượng String. Lưu ý rằng ở đây
      có việc chia sẻ cùng một chuỗi ký tự.
*/
{
   return (const char *) entries;
}
```

Cách hiện thực này cũng không hoàn toàn thích đáng do nó cho phép truy xuất dữ liệu bên trong của đối tượng String. Tuy nhiên chúng ta sẽ thấy những

cách giải quyết khác cũng gặp một số vấn đề. Cách giải quyết này còn có được ưu điểm là tính hiệu quả.

Phương thức `c_str()` trả về con trỏ chỉ đến mảng các ký tự chỉ có thể đọc chứ không thể sửa đổi do chúng ta đã ép kiểu sang `const char*`. Tuy nhiên người lập trình có thể ép kiểu ngược trở lại và gán vào một con trỏ khác làm phá vỡ tính đóng kín của dữ liệu của chúng ta. Một vấn đề nghiêm trọng hơn chính là bí danh được tạo bởi phương thức này. Chúng ta thấy rằng người lập trình nên sử dụng con trỏ trả về ngay sau khi vừa gọi phương thức, nếu không những gì xảy ra sẽ không lường trước được. Lấy ví dụ sau:

```
String s = "abc";
const char *new_string = s.c_str();
s = "def";
cout << new_string;
```

Lệnh `s = "def"` đã làm thay đổi dữ liệu mà `new_string` chỉ đến.

Một chiến lược khác cho phương thức `c_str()` có thể là định vị vùng nhớ động mới để chép dữ liệu của đối tượng `String` sang. Cách hiện thực này rõ ràng là kém hiệu quả hơn, đặc biệt đối với `String` dài. Ngoài ra nó còn có một nhược điểm nghiêm trọng, đó là khả năng tạo rác. `String` mà `c_str()` trả về không còn chia sẻ dữ liệu với đối tượng `String` nữa, và như vậy người lập trình phải nhớ `delete` nó khi không còn sử dụng. Chẳng hạn, nếu chỉ việc in ra như dưới đây thì trong bộ nhớ đã để lại rác do cách hiện thực vừa nêu.

```
String s = "Some very long string";
cout << s.c_str();
```

Tóm lại, tuy chúng ta vẫn giữ phương án đầu tiên cho phương thức `c_str()`, nhưng người lập trình không nên sử dụng phương thức này vì nó phá vỡ tính đóng kín của đối tượng `String`, trừ khi muốn sử dụng lại các hàm thư viện của `C-String` và đã hiểu thật rõ về bản chất của sự việc.

Cuối cùng, chúng ta xem xét các tác vụ so sánh được định nghĩa lại. Hiện thực sau đây của phép so sánh bằng được định nghĩa lại thật ngắn gọn và hiệu quả nhờ phương thức `c_str()`.

```
bool operator ==(const String &first, const String &second)
/*
post: Trả về true nếu đối tượng first giống đối tượng second. Ngược lại trả về false.
*/
{
   return strcmp(first.c_str(), second.c_str()) == 0;
}
```

Các tác vụ so sánh định nghĩa lại khác có hiện thực hầu như tương tự.

5.4. Các tác vụ trên String

Chúng ta sẽ phát triển một số tác vụ làm việc trên các đối tượng String. Trong nhiều trường hợp, các hàm của C-String có thể được gọi trực tiếp cho các đối tượng String đã chuyển đổi:

```
String s = "some_string";
cout << s.c_str() << endl;
cout << strlen(s.c_str()) << endl;
```

Đối với những hàm không thay đổi các thông số String như strcpy, chúng ta sẽ viết các phiên bản định nghĩa lại có thông số là đối tượng String thay vì char*. Như chúng ta đã biết, trong C++, một hàm được gọi là có định nghĩa lại nếu hai hoặc ba phiên bản khác nhau của nó có trong cùng một chương trình. Chúng ta đã có các *constructor* và các tác vụ gán định nghĩa lại. Khi một hàm được định nghĩa lại, chúng phải có các thông số khác nhau. Căn cứ vào các thông số được gởi khi gọi hàm, trình biên dịch biết được cần phải sử dụng phiên bản nào.

Phiên bản định nghĩa lại cho strcat có khai báo như sau:

```
void strcat(String &add_to, const String &add_on)
```

Người sử dụng có thể gọi strcat(s,t) để nối chuỗi ký tự t vào chuỗi ký tự s. s là một String, t có thể là String hoặc C-String. Nếu t là C-String thì trước hết *constructor* có thông số char* sẽ thực hiện để chuyển t thành một đối tượng String cho hợp kiểu thông số mà strcat yêu cầu.

```
void strcat(String &add_to, const String &add_on)
/*
post: String add_on được nối vào sau String add_to.
*/
{
   const char *cfirst = add_to.c_str();
   const char *csecond = add_on.c_str();
   char *copy = new char[strlen(cfirst) + strlen(csecond) + 1];
   strcpy(copy, cfirst);
   strcat(copy, csecond);
   add_to = copy;
   delete []copy;
}
```

Trong phương thức trên có gọi strcat với hai thông số là char* và const char*, tại đây trình biên dịch sẽ gọi đúng hàm thư viện của C-String chứ không phải gọi đệ quy chính phương thức này.

Do add_to là đối tượng String, lệnh add_to = copy trước hết gọi *constructor* để chuyển copy kiểu char* sang đối tượng String, sau đó mới gọi phép gán định nghĩa lại của lớp String. Nói cách khác, điều này dẫn đến việc

chép chuỗi ký tự hai lần. Để tránh điều này chúng ta hãy thử thay đổi dòng lệnh. Chẳng hạn, một cách đơn giản chúng ta khai báo `strcat` là một hàm *friend* của lớp `String`. Khi đó chúng ta có thể truy cập đến thuộc tính `entries` của lớp `String: add_to.entries = copy.`

Chúng ta cần hàm để đọc các đối tượng `String`. Chúng ta có thể thực hiện tương tự như đối với `C-String`, tác vụ << sẽ được định nghĩa lại để nhận thông số là một `String`. Tuy nhiên, chúng ta cũng có thể dùng cách khác để xây dựng hàm `read_in` như sau:

```
String read_in(istream &input)
/*
post: Trả về một đối tượng String đọc từ thông số istream (ký tự kết thúc chuỗi ký tự được
      quy ước là ký tự xuống hàng hoặc kết thúc tập tin)
*/
{
   List<char> temp;
   int size = 0;
   char c;
   while ((c = input.peek()) != EOF && (c = input.get()) != '\n')
      temp.insert(size++, c);
   String answer(temp);
   return answer;
}
```

Hàm trên sử dụng một đối tượng `temp` để gom các ký tự từ thông số `input`, sau đó gọi *constructor* để chuyển đổi `temp` này thành đối tượng `String`. Ký tự kết thúc chuỗi ký tự là ký tự xuống hàng hoặc ký tự kết thúc tập tin.

Một phiên bản được đề nghị khác cho hàm `read_in` là thêm thông số thứ hai để chỉ ra ký tự kết thúc chuỗi ký tự mong muốn:

```
String read_in(istream &input, int &terminator);

post: Trả về một đối tượng String đọc từ thông số istream (ký tự kết thúc chuỗi ký tự được quy ước
      là ký tự xuống hàng hoặc kết thúc tập tin, ký tự này cũng được trả về thông qua tham biến
      terminator.)
```

Tương tự chúng ta có phương thức để in một đối tượng `String`:

```
void write(String &s)
/*
post: Đối tượng String s được in ra cout.
*/
{
   if (strlen(s.c_str())>0)
      cout << s.c_str() << endl;
}
```

Trong các phần tiếp theo chúng ta sẽ sử dụng các hàm thư viện cho lớp `String` như sau:

```
void strcpy(String &copy, const String &original);
```
post: Hàm chép **String original** sang **String copy**.

```
void strncpy(String &copy, const String &original, int n);
```
post: Hàm chép nhiều nhất là n ký tự từ **String original** sang **String copy**.

```
int strstr(const String &text, const String &target);
```
post: Nếu **String target** là chuỗi con (*subtring*) của **String text**, hàm trả về vị trí xuất hiện đầu tiên của `target` trong `text`; ngược lại, hàm trả về -1.

Các hiện thực của các hàm này theo cách sử dụng lại thư viện `C-String` được xem như bài tập.

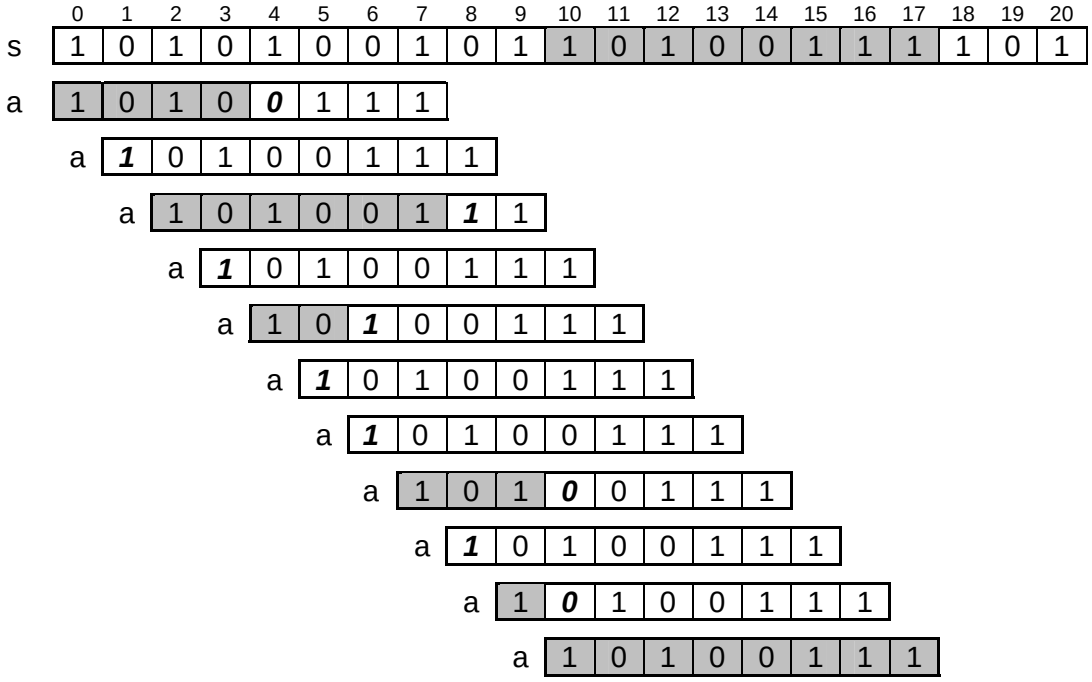
5.5. Các giải thuật tìm một chuỗi con trong một chuỗi

Phần sau đây chúng ta sẽ tìm hiểu lại cách hiện thực của một vài hàm thư viện của `C-String`. Các phép xử lý cơ bản trên chuỗi ký tự bao gồm: tìm một chuỗi con trong một chuỗi, thay thế một chuỗi con bằng một chuỗi khác, chèn một chuỗi con vào một chuỗi, loại một chuỗi con trong một chuỗi,... Trong đó phép tìm một chuỗi con trong một chuỗi có thể xem là phép cơ bản nhất, những phép còn lại có thể được thực hiện dễ dàng sau khi đã xác định được vị trí của chuỗi con. Chúng ta sẽ tìm hiểu hai giải thuật tìm chuỗi con trong một chuỗi cho trước.

5.5.1. Giải thuật Brute-Force

Ý tưởng giải thuật này vô cùng đơn giản, đó là thử so trùng chuỗi con tại mọi vị trí bắt đầu trong chuỗi đã cho (Hình 5.2). Giả sử chúng ta cần tìm vị trí của chuỗi a trong chuỗi s. Các vị trí bắt đầu so trùng a trên s là 0, 1, 2, ...Mỗi lần so trùng, chúng ta lần lượt so sánh từng cặp ký tự của a và s từ trái sang phải. Khi gặp hai ký tự khác nhau, chúng ta lại phải bắt đầu so trùng từ đầu chuỗi a với vị trí mới. Vị trí bắt đầu so trùng trên s lần thứ i sẽ là vị trí bắt đầu so trùng trên s lần thứ i-1 cộng thêm 1. Các ký tự in nghiêng trong hình vẽ bên dưới là vị trí thất bại trong một lần so trùng, phần có nền xám bên trái chúng là những ký tự so trùng đã thành công. Một lần so trùng nào đó mà chúng ta đã duyệt qua được hết chiều dài của a xem như đã tìm thấy a trong s và giải thuật dừng.

Cho i là chỉ số chạy trên s và j là chỉ số chạy trên a, j luôn được gán về 0 khi bắt đầu một lần so trùng. Khi gặp thất bại trong một lần so trùng nào đó thì cả i và j đều đã tiến được j bước so với lúc bắt đầu so trùng. Do đó để bắt đầu so trùng cho lần sau, i cần lùi về j-1 bước.



Hình 5.2- Minh họa giải thuật Brute-Force

```
// Giải thuật Brute-Force
int strstr(const String &s, const String &a);
/*
post: Nếu chuỗi a là chuỗi con của chuỗi  s, hàm trả về vị trí xuất hiện đầu tiên của a trong
s; ngược lại, hàm trả về -1.
*/
{
   int   i = 0, // Chỉ số chạy trên s.
         j = 0, // Chỉ số chạy trên a.
         ls = s.strlen(); // Số ký tự của s.
         la = a.strlen(), // Số ký tự của a.
   const char* pa = a.c_str(); //Địa chỉ ký tự đầu tiên của a.
   const char* ps = s.c_str(); //Địa chỉ ký tự đầu tiên của s.
   do {
      if (pa[j] == ps[i]){
         i++;
         j++;
      };
      else {
         i = i - (j - 1); // Lùi về cho lần so trùng kế tiếp.
         j = 0;
      }
   } while ((j<la) && (i<ls));
   if (j>=la) return i - la;
   else return -1;
}
```

Trường hợp xấu nhất của giải thuật Brute-Force là chuỗi con a trùng với phần cuối cùng của chuỗi s. Khi đó chúng ta đã phải lập lại `ls-la+1` lần so trùng, với

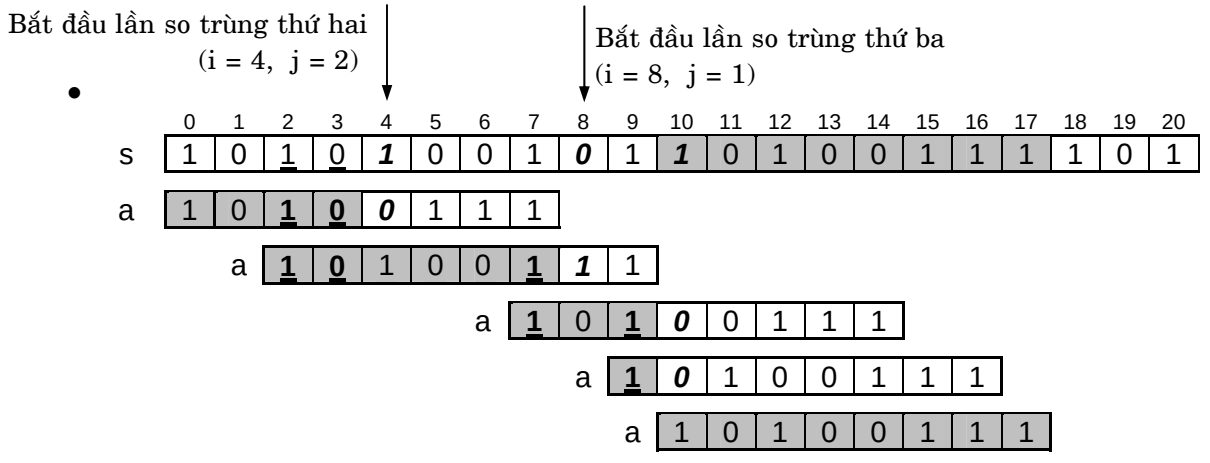
`ls` và `la` là chiều dài của chuỗi s và chuỗi a. Mỗi lần so trùng đã phải so sánh `la` ký tự. Số lần so sánh tối đa là `la*(ls-la+1)` ≈ `la*ls`.

5.5.2. Giải thuật Knuth-Morris-Pratt

Giải thuật này do Knuth, Morris và Pratt đưa ra, còn gọi là giải thuật KMP-Search.

Trong ví dụ trên chúng ta thấy giải thuật Brute-Force phải so trùng đến lần thứ 11 mới phát hiện được vị trí cần tìm. Giải thuật `KMP-Search` dưới đây tiết kiệm được một số lần so trùng và chỉ phải so trùng đến lần thứ 5. Hơn thế nữa, chỉ số i chạy trên s cũng không bao giờ phải lùi lại. Để có được điều này, chúng ta hãy cố gắng rút ra nhận xét từ hình 5.3 bên dưới. Trong lần so trùng thứ nhất, khi i=4 thì $a_j \neq s_i$, khi đó a sẽ được dịch chuyển về phía phải sao cho đoạn đầu của a trùng khớp với đoạn cuối của a trong phần đã được duyệt qua (chỉ tính phần màu xám). Trong hình vẽ là hai ký tự 1 và 0 có gạch dưới. Lần so trùng kế tiếp chính là từ vị trí này, và những lần so trùng trung gian giữa hai lần này có thể bỏ qua. Điều này có thể lý giải như sau: nếu phần đầu của a trùng với phần cuối của a thì nó cũng trùng với phần tương ứng của s bên trên, do phần cuối của a vừa mới được so trùng thành công với phần tương ứng của s. Được như vậy thì i mới hoàn toàn không phải lùi lại. Trong lần so trùng mới, chính s_i này sẽ được so sánh với a_j, với j sẽ được tính toán thích hợp mà chúng ta sẽ bàn đến sau. Trong ví dụ chúng ta thấy j = 2, lần so sánh đầu tiên của lần so trùng thứ hai là so sánh giữa s_4 và a_2.

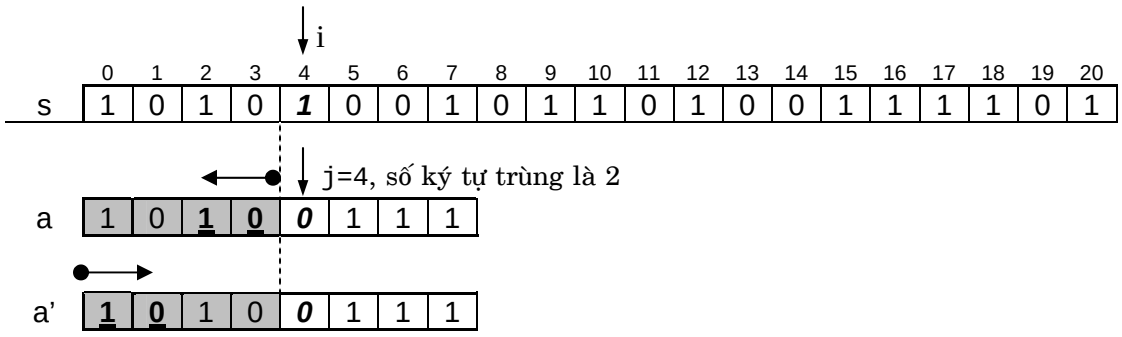
Tương tự, khi lần so trùng thứ hai thất bại tại s_8, chuỗi con a sẽ được dịch chuyển rất xa, tiết kiệm được rất nhiều lần so trùng. Chúng ta dễ dàng kiểm chứng, với những vị trí trung gian khác, phần đầu của a không trùng với phần cuối (chỉ tính phần màu xám) của a, nên cũng không thể trùng với phần tương ứng trên s, có thực hiện so trùng cũng sẽ thất bại mà thôi.



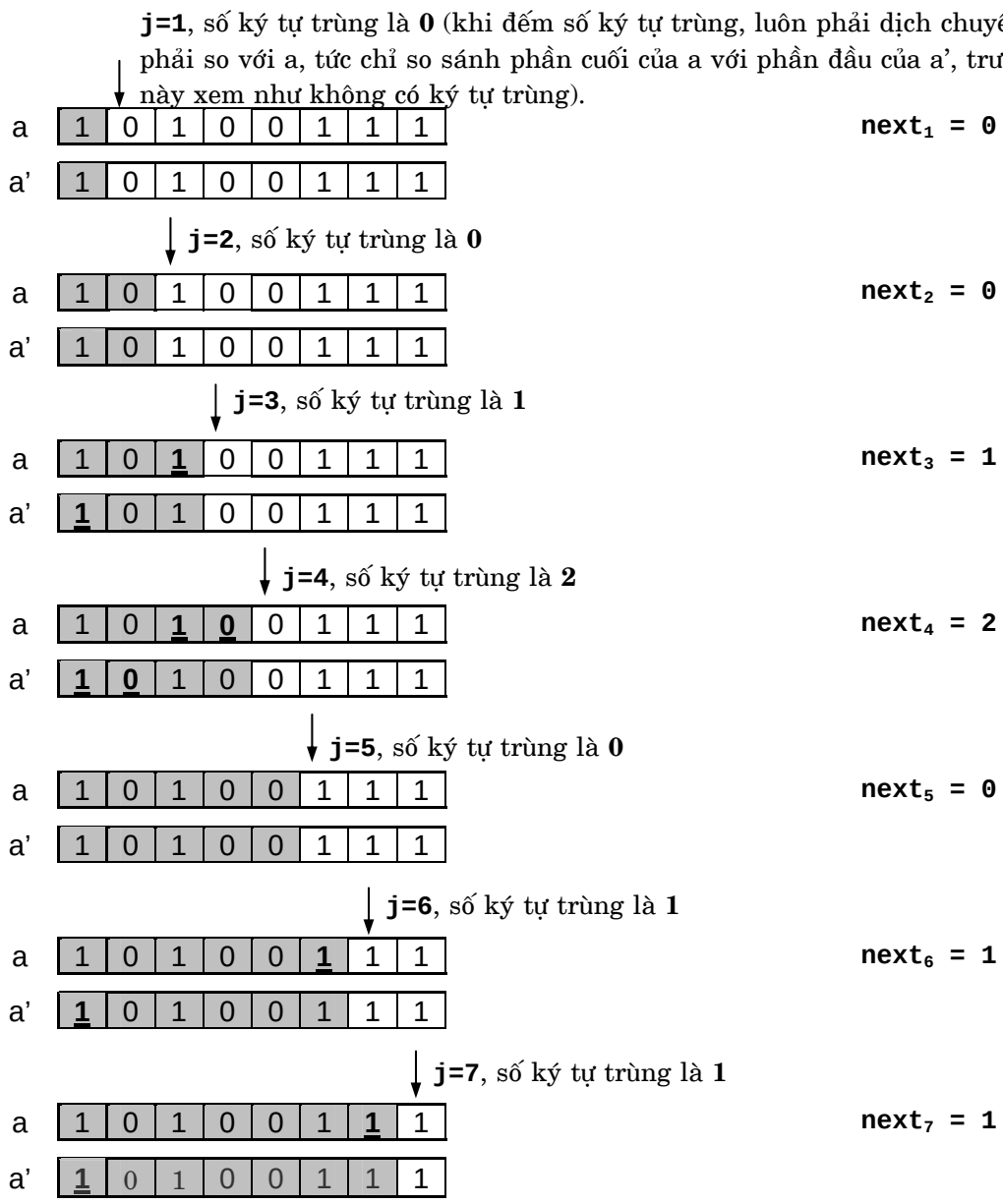
Hình 5.3- Minh họa giải thuật Knuth-Morris-Pratt

Hình vẽ dưới đây giúp chúng ta hiểu được cách tính chỉ số j thích hợp cho đầu mỗi lần so trùng (trong khi i không lùi về mà giữ nguyên để tiếp tục tiến tới).

Trích từ hình vẽ trên, chúng ta có được kết quả sau đây.
Xét vị trí `i = 4, j = 4`, do so sánh s_i với a_j thất bại, chúng ta đang muốn biết phần cuối của `a` kể từ điểm này trở về trước (tức chỉ tính phần màu xám) và phần đầu của `a` trùng được bao nhiêu ký tự. Gọi `a' = a`. Chúng ta sẽ nhìn quét từ cuối phần màu xám của `a` và từ đầu của `a'`, chúng ta sẽ biết được có bao nhiêu ký tự trùng. Đó là hai ký tự `1` và `0` được gạch dưới.



Như vậy, điều này hoàn toàn không còn phụ thuộc vào s nữa. Chúng ta có thể tính số ký tự trùng theo j dựa trên a và a'. Đồng thời ta thấy số ký tự trùng này cũng là chỉ số mà j phải lùi về cho lần so trùng kế tiếp a_j với s_i, i không đổi. Chúng ta bắt đầu với j = 1 và xem hình 5.4 sau đây.



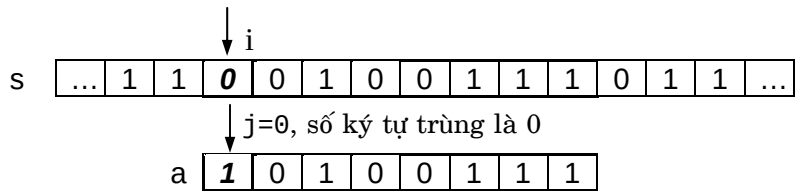
Hình 5.4- Minh họa giải thuật Knuth-Morris-Pratt

Giả sử chúng ta đã tạo được danh sách **next** mà phần tử thứ j chứa trị mà j phải lùi về khi đang so sánh a_j với s_i mà thất bại ($a_j \neq s_i$), để bắt đầu lần so trùng kế tiếp (i giữ nguyên không đổi). Hình 5.4 cho thấy $next_1$ luôn bằng 0 với mọi a. Chúng ta có giải thuật `KMP-Search` như dưới đây.

Lần so trùng thứ nhất luôn bắt đầu từ ký tự đầu của s và a, nên hai chỉ số i và j đều là 0.

- Trường hợp dễ hiểu nhất là trong khi mà $a_j=s_i$ thì i và j đều được nhích tới. Điều kiện dừng của vòng lặp hoàn toàn như giải thuật Brute-Force trên, có nghĩa là j đi được hết chiều dài của a (tìm thấy a trong s), hoặc i đi quá chiều dài của s (việc tìm kết thúc thất bại).

- Trường hợp $a_j \neq s_i$ (với $j \neq 0$) trong một lần so trùng nào đó thì như đã nói ở trên, chỉ việc cho j lùi về vị trí đã được chứa trong phần tử thứ j trong danh sách next. Nhờ vậy trong lần lặp kế tiếp sẽ tiếp tục so sánh a_j này với s_i mà i không đổi.
- Riêng trường hợp đặc biệt, với j = 0 mà $a_j \neq s_i$, ta xem hình dưới đây



Bất cứ một lần so sánh s_i nào đó với a_0 mà thất bại thì chuỗi a cũng phải dịch chuyển sang phải một bước, để lần so sánh kế tiếp (cũng là lần so trùng mới) có thể so sánh $\mathbf{a_0}$ với $\mathbf{s_{i+1}}$. Như vậy ta chỉ cần tăng i và giữ nguyên j mà thôi.

```
// Giải thuật Knuth- Morris – Pratt

int strstr(const String &s, const String &a);
/*
post: Nếu a là chuỗi con của s, hàm trả về vị trí xuất hiện đầu tiên của a trong s;
      ngược lại, hàm trả về -1.
*/
{
   List<int> next;
   int   i = 0, // Chỉ số chạy trên s.
         j = 0, // Chỉ số chạy trên a.
         ls = s.strlen(); // Số ký tự của s.
         la = a.strlen(), // Số ký tự của a.
   const char* pa = a.c_str(); //Địa chỉ ký tự đầu tiên của a.
   const char* ps = s.c_str(); //Địa chỉ ký tự đầu tiên của s.
   InitNext(a, next);    // Khởi gán các phần tử next1, next2,…,nextla-1.
                         // Không sử dụng next0.
   do {
      if (pa[j]==ps[i]){// Vẫn còn ký tự trùng trong một lần so trùng
            i++;         // nào đó, i và j được quyền nhích tới.
            j++;
      }
      else
         if (j == 0)     // Đây là trường hợp đặc biệt, phải dịch a sang phải
            i++;         // một bước, có nghĩa là cho i nhích tới.
         else
            next.retrieve(j, j); // Cho j lùi về trị đã chứa trong nextj.
   } while ((j<la) && (i<ls));
   if (j>=la) return i – la;
   else return –1;
}
```

Sau đây chúng ta sẽ viết hàm InitNext gán các trị cho các phần tử của next, tức là tìm số phần tử trùng theo hình vẽ 5.4. Có một điều khá thú vị trong giải thuật này, đó chính là hàm tạo danh sách next lại sử dụng ngay chính danh sách này. Chúng ta thấy rằng để tìm số phần tử trùng như đã nói, chúng ta cần dịch chuyển a′ về bên phải so với a, mà việc dịch chuyển a′ trên a cũng hoàn toàn giống như việc dịch chuyển a trên s trong khi đi tìm a trong s.

```
// Hàm phụ trợ gán các phần tử cho danh sách next.

void InitNext(const String &a, List<int> &next);
/*
post: Gán các trị cho các phần tử của next dựa trên chuỗi ký tự a.
*/
{
   int   i = 1, // Chỉ số chạy trên a.
         j = 0, // Chỉ số chạy trên a'.
         la = a.strlen(), // Số ký tự của a (cũng là của a').
   const char* pa = a.c_str(); //Địa chỉ ký tự đầu tiên của a (cũng là của a').
   next.clear();
   next.insert(1, 0); // Luôn đúng với mọi a.
   do {
      if (pa[j]==pa[i]){    // Vẫn còn ký tự trùng trong một lần so trùng
      i++;                 // nào đó, i và j được quyền nhích tới.
         j++;                 // Từ vị trí i trên a trở về trước, j xem như đã
         next.insert(i, j);// quét được số phần tử trùng của a' so với a.
      }
      else
         if (j == 0){    // Trường hợp đặc biệt, phải dịch a sang phải
             i++;         // một bước, có nghĩa là cho i nhích tới.
            next.insert(i, j);
         };
         else
            next.retrieve(j, j); // Cho j lùi về trị đã chứa trong next_j.
   } while (i<la); // i=la là đã gán xong la phần tử của next,
                   // không sử dụng next_0.
}
```

Hàm tạo next được chép lại từ giải thuật KMP-Search trên, chỉ có vài điểm bổ sung như sau: với i chạy trên a và j chạy trên a′, và a′ luôn phải dịch phải so với a, chúng ta khởi gán i=1 và j=0.

Do i tăng đến đâu là chúng ta xem như đã so trùng xong phần cuối của a (kể từ vị trí i này trở về trước) với phần đầu của a′, nên $next_i$ đã được xác định. Trong quá trình so trùng, trong khi mà a_i vẫn còn bằng a'_j, i và j đều nhích tới. Vì vậy, chúng ta dễ thấy rằng j chính là số phần tử đã trùng được của a' so với a, chúng ta có phép gán $next_i$=j.

Khi $a_i \neq a'_j$, chúng ta sử dụng ý tưởng của KMP-Search là cho j lùi về $next_j$. Vấn đề còn lại cần kiểm chứng là giá trị của $next_j$ phải có trước khi nó được sử dụng. Do chúng ta đã gán vào $\mathbf{next_i}$ và đã sử dụng $\mathbf{next_j}$, mà i luôn luôn đi trước j, nên chúng ta hoàn toàn yên tâm về điều này.

Cuối cùng, chỉ còn một điều nhỏ mà chúng ta cần xem xét. Đó là trường hợp có nhiều phương án cho số ký tự trùng nhau. Chẳng hạn với a là "10101010111…" và j=8, số ký tự trùng khi dịch a'=a về bên phải so với a là:

```
                      ↓ Vị trí j đang xét
a        10101010111...          Số ký tự trùng là 6
a'         101010 10111...

a        1010101 0111...
a'           1010 1010111...     Số ký tự trùng là 4

a        101010 10111...
a'             10 101010111...   Số ký tự trùng là 2
```

Sinh viên hãy tự suy nghĩ xem cách chọn phương án nào là đúng đắn nhất và kiểm tra lại các đoạn chương trình trên xem chúng có cần phải được sửa đổi gì hay không.

Ngoài ra, giải thuật KMP-Search còn có thể cải tiến một điểm nhỏ, đó là trước khi gán $next_i$=j trong InitNext, chúng ta kiểm tra nếu pa_j=pa_i thì sẽ gán $next_i$=$next_j$. Do khi so trùng pa_i mà thất bại thì có lùi về pa_{nexti}=paj cũng sẽ thất bại, chúng ta nên lùi hẳn về pa_{nextj}.

Số lần so sánh tối đa trong KMP-Search là ls+la.