Chương 18 – ỨNG DỤNG DANH SÁCH LIÊN KẾT VÀ BẢNG BĂM

Đây là một ứng dụng có sử dụng CTDL danh sách và bảng băm. Thông qua ứng dụng này sinh viên có dịp nâng cao kỹ năng thiết kế hướng đối tượng, giải quyết bài toán từ ngoài vào trong. Ngoài ra, đây cũng là một ví dụ rất hay về việc sử dụng một CTDL đúng đắn không những đáp ứng được yêu cầu bài toán mà còn làm tăng hiệu quả của chương trình lên rất nhiều.

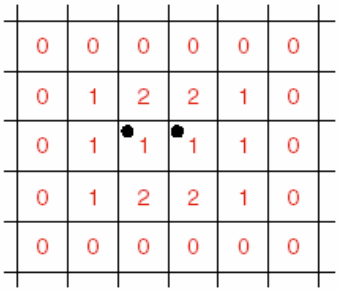
18.1. Giới thiệu về chương trình `Game_Of_Life`

`Game_Of_Life` là một chương trình giả lặp một sự tiến triển của sự sống, không phải là một trò chơi với người sử dụng. Trên một lưới chữ nhật không có giới hạn, mỗi ô hoặc là ô trống hoặc đang có một tế bào chiếm giữ. Ô có tế bào được gọi là ô sống, ngược lại là ô chết. Mỗi thời điểm ổn định của toàn bộ lưới chúng ta gọi là một trạng thái. Để chuyển sang trạng thái mới, một ô sẽ thay đổi tình trạng sống hay chết tùy thuộc vào số ô sống chung quanh nó trong trạng thái cũ theo các quy tắc sau:

1. Một ô có tám ô kế cận.
2. Một ô đang sống mà không có hoặc chỉ có 1 ô kế cận sống thì ô đó sẽ chết do đơn độc.
3. Một ô đang sống mà có từ 4 ô kế cận trở lên sống thì ô đó cũng sẽ chết do quá đông.
4. Một ô đang sống mà có 2 hoặc 3 ô kế cận sống thì nó sẽ sống tiếp trong trạng thái sau.
5. Một ô đang chết trở thành sống trong trạng thái sau nếu nó có chính xác 3 ô kế cận sống.
6. Sự chuyển trạng thái của các ô là đồng thời, có nghĩa là căn cứ vào số ô kế cận sống hay chết trong một trạng thái để quyết định sự sống chết của các ô ở trạng thái sau.

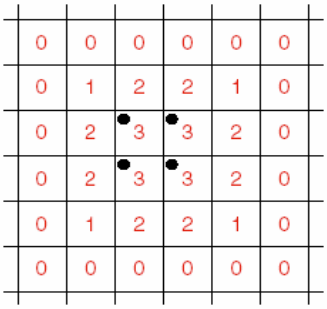
18.2. Các ví dụ

Chúng ta gọi một đối tượng lưới chứa các ô sống và chết như vậy là một cấu hình. Trong hình 18.1, con số ở mỗi ô biểu diễn số ô sống chung quanh nó, theo quy tắc thì cấu hình này sẽ không còn ô nào sống ở trạng thái sau. Trong khi đó cấu hình ở hình 18.2 sẽ bền vững và không bao giờ thay đổi.

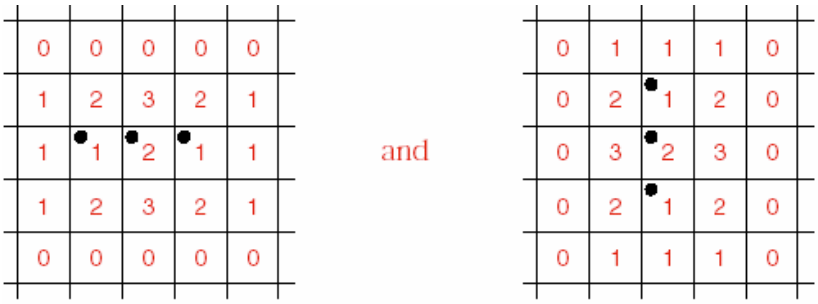


Hình 18.1- Một trang thái của Game of Life

Với một trạng thái khởi đầu nào đó, chúng ta khó lường trước được điều gì sẽ xảy ra. Một vài cấu hình đơn giản ban đầu có thể biến đổi qua nhiều bước để thành các cấu hình phức tạp hơn nhiều, hoặc chết dần một cách chậm chạp, hoặc sẽ đạt đến sự bền vững, hoặc chỉ còn là sự chuyển đổi lặp lại giữa một vài trạng thái.



Hình 18.2 – Cấu hình có trạng thái bền vững



Hình 18.3 – Hai cấu hình này luân phiên thay đổi nhau.

18.3. Giải thuật

Mục đích của chúng ta là viết một chương trình hiển thị các trạng thái liên tiếp nhau của một cấu hình từ một trạng thái ban đầu nào đó.

Giải thuật:

- Khởi tạo một cấu hình ban đầu có một số ô sống.
- In cấu hình đã khởi tạo.
- Trong khi người sử dụng vẫn còn muốn xem sự biến đổi của các trạng thái:
  - Cập nhật trạng thái mới dựa vào các quy tắc của chương trình.
  - In cấu hình.

Chúng ta sẽ xây dựng lớp **`Life`** mà đối tượng của nó sẽ có tên là **`configuration`**. Đối tượng này cần 3 phương thức: **`initialize()`** để khởi tạo, **`print()`** để in trạng thái hiện tại và **`update()`** để cập nhật trạng thái mới.

18.4. Chương trình chính cho `Game_Of_Life`

```
#include "utility.h"
#include "life.h"

int main()  //  Chương trình Game_Of_Life.
/*
pre:  Người sử dụng cho biết trạng thái ban đầu của cấu hình.
post: Chương trình in các trạng thái thay đổi của cấu hình cho đến khi người sử dụng muốn
      ngưng chương trình. Cách thức thay đổi trạng thái tuân theo các quy tắc của trò chơi.
uses: Lớp Life với các phương thức initialize(), print(), update().
      Các hàm phụ trợ instructions(),  user_says_yes().
*/

{
   Life configuration;
   instructions();
   configuration.initialize();
   configuration.print();
   cout << "Continue viewing new generations? " << endl;
   while (user_says_yes()) {
      configuration.update();
      configuration.print();
      cout << "Continue viewing new generations? " << endl;
   }
}
```

Với chương trình `Life` này chúng ta cần hiện thực những phần sau:

- Lớp **`Life`**.
- Phương thức **`initialize()`** khởi tạo cấu hình của **`Life`**.
- Phương thức **`print()`** hiển thị cấu hình của **`Life`**.
- Phương thức **`update()`** cập nhật đối tượng **`Life`** chứa cấu hình ở trạng thái mới.
- Hàm **`user_says_yes()`** để hỏi người sử dụng có tiếp tục xem trạng thái kế tiếp hay không.
- Hàm **`instruction()`** hiển thị hướng dẫn sử dụng chương trình.

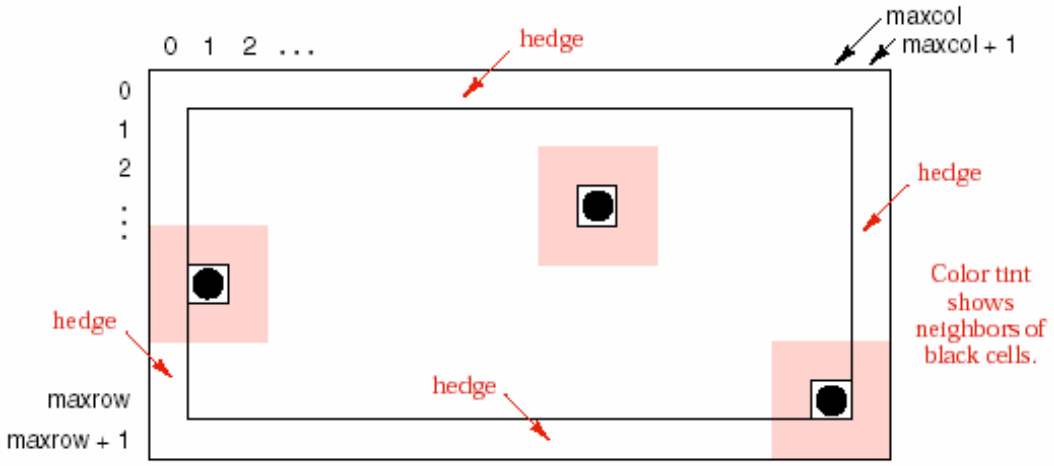
Với cách phác thảo này chúng ta có thể chuyển sang giai đoạn kế, đó là chọn lựa cách tổ chức dữ liệu để hiện thực lớp `Life`.

18.4.1. Phiên bản thứ nhất cho lớp `Life`

Trong phiên bản thứ nhất này, chúng ta chưa sử dụng một lớp CTDL có sẵn nào, mà chỉ suy nghĩ đơn giản rằng đối tượng `Life` cần một mảng hai chiều các số nguyên để biểu diễn lưới các ô. Trị 1 biểu diễn ô sống và trị 0 biểu diễn ô chết. Kích thước mảng lấy thêm bốn biên dự trữ để việc đếm số ô sống kế cận được thực hiện dễ dàng cho cả các ô nằm trên cạnh biên hay góc. Tất nhiên với cách chọn lựa này chúng ta đã phải lơ qua một đòi hỏi của chương trình: đó là lưới chữ nhật phải không có giới hạn.

Ngoài các phương thức `public`, lớp `Life` cần thêm một hàm phụ trợ **`neighbor_count`** để tính các ô sống kế cận của một ô cho trước.

```
const int maxrow = 20, maxcol = 60; // Kích thước để thử chương trình
class Life {
public:
   void initialize();
   void print();
   void update();
private:
   int grid[maxrow + 2][maxcol + 2];// Dự trữ thêm 4 biên như hình vẽ dưới đây
   int neighbor_count(int row, int col);
};
```



Hình 18.4 – Lưới các ô của `Life` có dự trữ bốn biên

Dưới đây là hàm **`neighbor_count`** được gọi bởi phương thức **`update`**.

```
int Life::neighbor_count(int row, int col)
/*
pre:  Đối tượng Life chứa trạng thái các ô sống, chết. row và col là tọa độ hợp lệ của một ô.
post: Trả về số ô đang sống chung quanh ô tại tọa độ row, col.
*/
{
   int i, j;
   int count = 0;
```

```
   for (i = row - 1; i <= row + 1; i++) // Quét tất cả 9 ô, kể cả tại (row, col)
      for (j = col - 1; j <= col + 1; j++)
         count += grid[i][j];  //  Nếu ô (i,j) sống thì có trị 1 và được cộng vào count
   count -= grid[row][col]; //  Trừ đi bản thân ô đang được xét
   return count;
}
```

Trong phương thức **update** dưới đây chúng ta cần một mảng tạm **new_grid** để lưu trạng thái mới vừa tính được.

```
void Life::update()
/*
pre:  Đối tượng Life đang chứa một trạng thái hiện tại.
post: Đối tượng Life chứa trạng thái mới.
*/
{
   int row, col;
   int new_grid[maxrow + 2][maxcol + 2];

   for (row = 1; row <= maxrow; row++)
      for (col = 1; col <= maxcol; col++)
         switch (neighbor_count(row, col)) {
         case 2:
            new_grid[row][col] = grid[row][col];   // giữ nguyên tình trạng cũ
            break;

         case 3:
            new_grid[row][col] = 1;                 // ô sẽ sống
            break;
         default:
            new_grid[row][col] = 0;                 // ô sẽ chết
         }

   for (row = 1; row <= maxrow; row++)
      for (col = 1; col <= maxcol; col++)
         grid[row][col] = new_grid[row][col];
}
```

Phương thức **initialize** nhận thông tin từ người sử dụng về các ô sống ở trạng thái ban đầu.

```
void Life::initialize()
/*
post: Đối tượng Life đang chứa trạng thái ban đầu mà người sử dụng mong muốn.
*/
{
   int row, col;

   for (row = 0; row <= maxrow+1; row++)
      for (col = 0; col <= maxcol+1; col++)
         grid[row][col] = 0;
   cout << "List the coordinates for living cells." << endl;
   cout << "Terminate the list with the special pair -1 -1" << endl;
   cin >> row >> col;
```

```
   while (row != -1 || col != -1) {
      if (row >= 1 && row <= maxrow)
         if (col >= 1 && col <= maxcol)
            grid[row][col] = 1;
         else
            cout << "Column " << col << " is out of range." << endl;
      else
         cout << "Row " << row << " is out of range." << endl;
      cin >> row >> col;
   }
}
```

```
void Life::print()
/*
pre:  Đối tượng Life đang chứa một trạng thái.
post: Các ô sống được in cho người sử dụng xem.
*/

{
   int row, col;
   cout << "\nThe current Life configuration is:" <<endl;
   for (row = 1; row <= maxrow; row++) {
      for (col = 1; col <= maxcol; col++)
         if (grid[row][col] == 1) cout << '*';
         else cout << ' ';
      cout << endl;
   }
   cout << endl;
}
```

Các hàm phụ trợ

Các hàm phụ trợ dưới đây có thể xem là khuôn mẫu và có thể được sửa đổi đôi chút để dùng cho các ứng dụng khác.

```
void instructions()
/*
post: In hướng dẫn sử dụng chương trình Game_Of_Life.
*/

{
   cout << "Welcome to Conway's game of Life." << endl;
   cout << "This game uses a grid of size "
        << maxrow << " by " << maxcol << " in which" << endl;
   cout << "each cell can either be occupied by an organism or not." << endl;
   cout << "The occupied cells change from generation to generation" << endl;
   cout << "according to the number of neighboring cells which are alive."
        << endl;
}
```

```
bool user_says_yes()
{
   int c;
   bool initial_response = true;
```

```
   do {  // Lặp cho đến khi người sử dụng gõ một ký tự hợp lệ.
      if (initial_response)
         cout << " (y,n)? " << flush;

      else
         cout << "Respond with either y or n: " << flush;

      do { //  Bỏ qua các khoảng trắng.
         c = cin.get();
      } while (c == '\n' || c ==' ' || c == '\t');
      initial_response = false;
   } while (c != 'y' && c != 'Y' && c != 'n' && c != 'N');
   return (c == 'y' || c == 'Y');
}
```

18.4.2. Phiên bản thứ hai với CTDL mới cho Life

Phiên bản trên giải quyết được bài toán Game_Of_Life nhưng với hạn chế là lưới các ô có kích thước giới hạn. Yêu cầu của bài toán là tấm lưới chứa các ô của Life là không có giới hạn. Chúng ta có thể khai báo lớp Life chứa một mảng thật lớn như sau:

```
class Life {
public:
   //  Các phương thức.
private:
   bool map[int][int];
   //  Các thuộc tính khác và các hàm phụ trợ.
};
```

nhưng cho dù nó có lớn mấy đi nữa thì cũng vẫn có giới hạn, đồng thời các giải thuật phải quét hết tất cả các ô trong lưới là hoàn toàn phí phạm. Điều không hợp lý ở đây là tại mỗi thời điểm chỉ có một số giới hạn các ô của Life là sống, tốt hơn hết chúng ta nên nhìn các ô sống này như là một ma trận thưa. Và chúng ta sẽ dùng các cấu trúc liên kết thích hợp.

18.4.2.1. Lựa chọn giải thuật

Chúng ta sẽ thấy, các công việc cần xử lý trên dữ liệu góp phần quyết định cấu trúc của dữ liệu.

Khi cần biết trạng thái của một ô đang sống hay chết, nếu chúng ta dùng phương pháp tra cứu của bảng băm thì giải thuật hiệu quả hơn rất nhiều: nếu ô có trong bảng thì có nghĩa là nó đang sống, ngược lại là nó đang chết. Việc duyệt danh sách để xác nhận sự có mặt của một phần tử hay không không hiệu quả bằng phương pháp băm như chúng ta đã biết. Đối với bất kỳ một ô nào có trong

cấu hình, chúng ta có thể xác định số ô sống chung quanh nó bằng cách tra cứu trạng thái của chúng.

Trong hiện thực mới của chúng ta cho phương thức **`update`**, chúng ta sẽ duyệt qua tất cả các ô có khả năng thay đổi trạng thái, xác định số ô sống chung quanh mỗi ô nhờ sử dụng bảng, và chọn ra những ô sẽ thực sự sống trong trạng thái kế.

18.4.2.2. Đặc tả cấu trúc dữ liệu

Tuy rằng bảng băm chứa tất cả các ô đang sống, nhưng nó chỉ tiện trong việc tra cứu trạng thái của từng ô mà thôi. Chúng ta cũng sẽ cần duyệt qua các ô sống trong cấu hình đó. Việc duyệt một bảng băm thường không hiệu quả. Do đó, ngoài bảng băm, chúng ta cần một danh sách các ô sống như là thành phần dữ liệu thứ hai của một cấu hình `Life`. Các đối tượng được lưu trong danh sách và bảng băm của cấu hình `Life` cùng chứa thông tin về các ô sống, nhưng chúng ta có hai cách truy cập khác nhau. Điều này phục vụ đắc lực cho giải thuật của bài toán như đã phân tích ở trên. Chúng ta sẽ biểu diễn các ô bằng các thể hiện của một cấu trúc gọi là **`Cell`**: mỗi ô cần một cặp tọa độ.

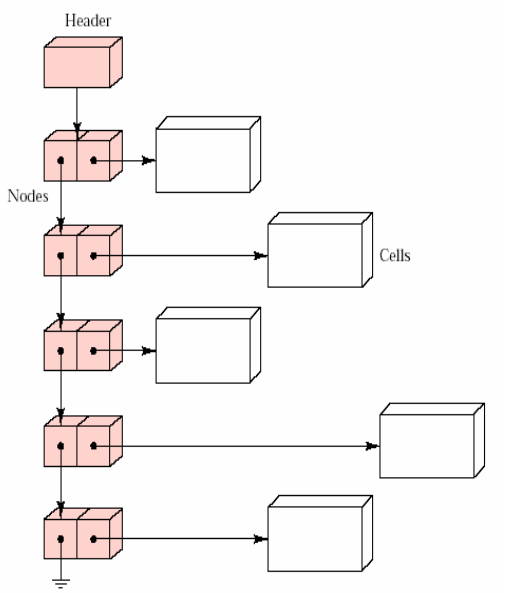
```
struct Cell {
   Cell() { row = col = 0; }   //  Các constructor
   Cell(int x, int y) { row = x;  col = y; }
   int row, col;
}
```

Khi cấu hình `Life` nới rộng, các ô ở bìa ngoài của nó sẽ xuất hiện dần dần. Như vậy một ô mới sẽ xuất hiện nhờ vào việc cấp phát động vùng nhớ, và nó sẽ chỉ được truy xuất đến thông qua con trỏ. Chúng ta sẽ dùng một `List` mà mỗi phần tử chứa con trỏ đến một ô (hình 18.5). Mỗi phần tử của `List` gồm hai con trỏ: một chỉ đến một ô đang sống và một chỉ đến phần tử kế trong `List`.

Cho trước một con trỏ chỉ một ô đang sống, chúng ta có thể xác định các tọa độ của ô đó bằng cách lần theo con trỏ rồi lấy hai thành phần `row` và `col` của nó. Như vậy, chúng ta có thể lưu các con trỏ chỉ đến các ô như là các bản ghi trong bảng băm; các toạ độ `row` và `col` của các ô, được xác định bởi con trỏ, sẽ là các khóa tương ứng.

Chúng ta cần lựa chọn giữa bảng băm địa chỉ mở và bảng băm nối kết. Các phần tử sẽ chứa trong bảng băm chỉ có kích thước nhỏ: mỗi phần tử chỉ cần chứa một con trỏ đến một ô đang sống. Như vậy, với bảng băm nối kết, kích thước của mỗi bản ghi sẽ tăng 100% do phải chứa thêm các con trỏ liên kết trong các danh sách liên kết. Tuy nhiên, bản thân bảng băm nối kết sẽ có kích thước rất nhỏ mà vẫn có thể chứa số bản ghi lớn gấp nhiều lần kích thước chính nó. Với bảng băm

địa chỉ mở, các bản ghi sẽ nhỏ hơn vì chỉ chứa địa chỉ các ô đang sống, nhưng cần phải dự trữ nhiều vị trí trống để tránh hiện tượng tràn xảy ra và để quá trình tìm kiếm không bị kéo dài quá lâu khi đụng độ thường xuyên xảy ra.



Hình 18.5 – Danh sách liên kết gián tiếp.

Để tăng tính linh hoạt, chúng ta quyết định sẽ dùng bảng băm nối kết có định nghĩa như sau:

```
class Hash_table {
public:
   Error_code insert(Cell *new_entry);
   bool retrieve(int row, int col) const;
private:
   List<Cell *> table[hash_size]; // Dùng danh sách liên kết.
};
```

Ở đây, chúng ta chỉ đặc tả hai phương thức: **`insert`** và **`retrieve`**. Việc truy xuất bảng là để biết bảng có chứa con trỏ chỉ đến một ô có tọa độ cho trước hay không. Do đó phương thức **`retrieve`** cần hai thông số chứa tọa độ `row` và `col` và trả về một trị `bool`. Chúng ta dành việc hiện thực hai phương thức này như là bài tập vì chúng rất tương tự với những gì chúng ta đã thảo luận về bảng băm nối kết trong chương 12.

Chúng ta lưu ý rằng `Hash_table` cần có những phương thức *constructor* và *destructor* của nó. Chẳng hạn, *destructor* của `Hash_table` cần gọi *destructor* của `List` cho từng phần tử của mảng **`table`**.

18.4.2.3. Lớp Life

Với các quyết định trên, chúng ta sẽ gút lại cách biểu diễn và những điều cần lưu ý cho lớp `Life`. Để cho việc thay đổi cấu hình được dễ dàng chúng ta sẽ lưu các thành phần dữ liệu một cách gián tiếp qua các con trỏ. Như vậy lớp `Life` cần có *constructor* và *destructor* để định vị cũng như giải phóng các vùng nhớ cấp phát động cho các cấu trúc này.

```
class Life {
public:
   Life();
   void initialize();
   void print();
   void update();
   ~Life();

private:
   List<Cell *> *living;
   Hash_table *is_living;
   bool retrieve(int row, int col) const;
   Error_code insert(int row, int col);
   int neighbor_count(int row, int col) const;
};
```

Các hàm phụ trợ **`retrieve`** và **`neighbor_count`** xác định trạng thái của một ô bằng cách truy xuất bảng băm. Hàm phụ trợ khác, **`insert`**, khởi tạo một đối tượng **`Cell`** cấp phát động và chèn nó vào bảng băm cũng như danh sách các ô trong đối tượng **`Life`**.

18.4.2.4. Các phương thức của Life

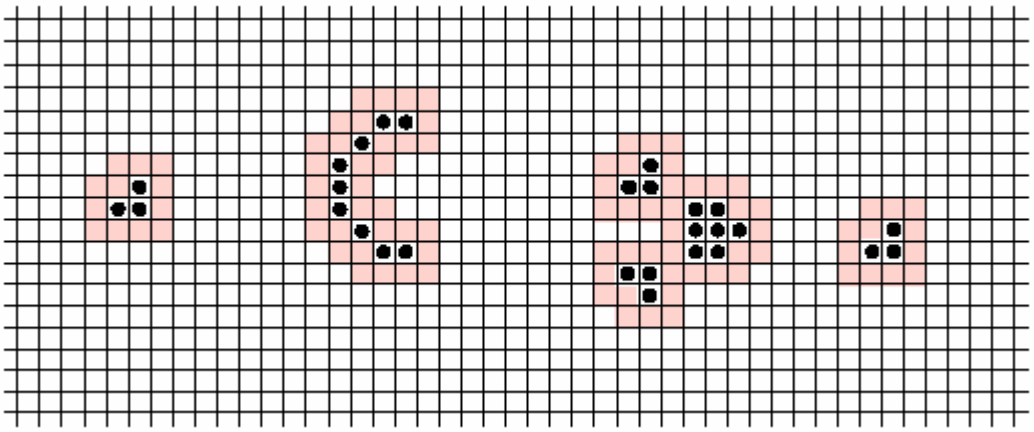
Chúng ta sẽ viết một vài phương thức và hàm của `Life` để minh họa cách xử lý các ô, các danh sách và những gì diễn ra trong bảng băm. Các hàm còn lại xem như bài tập.

Cập nhật cấu hình

Phương thức **`update`** có nhiệm vụ xác định cấu hình kế tiếp của `Life` từ một cấu hình cho trước. Trong phiên bản trước, chúng ta làm điều này bằng cách xét mọi ô có trong lưới chứa cấu hình **`grid`**, tính các ô kế cận chung quanh cho mỗi ô để xác định trạng thái kế tiếp của nó. Các thông tin này được chứa trong biến cục bộ **`new_grid`** và sau đó được chép vào **`grid`**.

Chúng ta sẽ lặp lại những công việc này ngoại trừ việc phải xét mọi ô có thể có trong cấu hình do đây là một lưới không có giới hạn. Thay vào đó, chúng ta nên giới hạn tầm nhìn của chúng ta chỉ trong các ô có khả năng sẽ sống trong trạng thái kế. Đó có thể là các ô nào? Rõ ràng đó chính là các ô đang sống trong trạng thái hiện tại, chúng có thể chết đi nhưng cũng có thể tiếp tục sống trong

trạng thái kế. Ngoài ra, một số ô đang chết cũng có thể trở nên sống trong trạng thái sau, nhưng đó chỉ là những ô đang chết nằm kề những ô đang sống (các ô có màu xám trong hình 9.18). Những ô xa hơn nữa không có khả năng sống dậy do chúng không có ô nào kế cận đang sống.



Hình 18.6 – Cấu hình Life với các ô chết viền chung quanh.

Trong phương thức **update**, biến cục bộ `Life` **new_configuration** dùng để chứa cấu hình mới: chúng ta thực hiện vòng lặp cho tất cả những ô đang sống và những ô kế cận của chúng, đối với mỗi ô như vậy, trước hết cần xác định xem nó đã được thêm vào **new_configuration** hay chưa, chúng ta cần phải cẩn thận để tránh việc thêm bị lặp lại hai lần cho một ô. Nếu quả thật ô đang xét chưa có trong **new_configuration**, chúng ta dùng hàm **neighbor_count** để quyết định việc có thêm nó vào cấu hình mới hay không.

Ở cuối phương thức, chúng ta cần hoán đổi các thành phần **List** và **Hash_table** giữa cấu hình hiện tại và cấu hình mới **new_configuration**. Sự hoán đổi này làm cho đối tượng **Life** có được cấu hình đã được cập nhật, ngoài ra, nó còn bảo đảm rằng các ô, danh sách, và bảng băm trong cấu hình cũ của đối tượng **Life** sẽ được giải phóng khỏi bộ nhớ cấp phát động (việc này được thực hiện do trình biên dịch tự động gọi *destructor* cho đối tượng cục bộ **new_configuration**).

```
void Life::update()
/*
post: Đối tượng Life chứa cấu hình ở trạng thái kế.
uses: Lớp Hash_table và lớp Life và các hàm phụ trợ.
*/
{
   Life new_configuration;
   Cell *old_cell;
```

```
    for (int i = 0; i < living->size(); i++) {
      living->retrieve(i, old_cell);       //  Lấy một ô đang sống.
      for (int row_add = -1; row_add < 2; row_add ++)
         for (int col_add = -1; col_add < 2; col_add++) {
            int new_row = old_cell->row + row_add,
                new_col = old_cell->col + col_add;

//  new_row, new_col là toạ độ của ô đang xét hoặc ô kế cận của nó.
            if (!new_configuration.retrieve(new_row, new_col))
               switch (neighbor_count(new_row, new_col)) {
               case 3:  //  Số ô sống kế cận là 3 thì ô đang xét trở thành sống.
                  new_configuration.insert(new_row, new_col);
                  break;

               case 2:  //  Số ô sống kế cận là 2 thì ô đang xét giữ nguyên trạng thái.
                  if (retrieve(new_row, new_col))
                     new_configuration.insert(new_row, new_col);
                  break;

               default: //  Ô sẽ chết.
                  break;
               }
         }
   }
// Tráo dữ liệu trong configuration với dữ liệu trong new_configuration.
                  new_configuration sẽ được dọn dẹp bởi destructor của nó.
   List<Cell *> *temp_list = living;
   living = new_configuration.living;
   new_configuration.living = temp_list;
   Hash_table *temp_hash = is_living;
   is_living = new_configuration.is_living;
   new_configuration.is_living = temp_hash;
}
```

In cấu hình

Để in được tất cả các ô đang sống, chúng ta có thể liệt kê lần lượt mỗi dòng một ô với tọa độ `row, col` của nó. Ngược lại nếu muốn biểu diễn đúng vị trí `row, col` trên lưới, chúng ta nhận thấy rằng chúng ta không thể hiển thị nhiều hơn là một mẩu nhỏ của một cấu hình `Life` không có giới hạn lên màn hình. Do đó chúng ta sẽ in một cửa sổ chữ nhật để hiển thị trạng thái của 20 x 80 các vị trí ở trung tâm của cấu hình `Life`.

Đối với mỗi ô trong cửa sổ, chúng ta truy xuất trạng thái của nó từ bảng băm và in một ký tự trắng hoặc khác trắng tương ứng với trạng thái chết hoặc sống của nó.

```
void Life::print()
/*
post: In một trạng thái của đối tượng Life.
uses: Life::retrieve.
*/
```

```
{
   int row, col;
   cout << endl << "The current Life configuration is:" << endl;

   for (row = 0; row < 20; row++) {
      for (col = 0; col < 80; col++)
         if (retrieve(row, col)) cout << '*';
         else cout << ' ';
      cout << endl;
   }
   cout << endl;
}
```

Tạo và thêm các ô mới

Phương thức **`insert`** tạo một ô mới với tọa độ cho trước và thêm nó vào bảng băm **`is_living`** và vào danh sách **`living`**.

```
ErrorCode Life::insert(int row, int col)
/*
pre:  Ô có tọa độ (row,col) không thuộc về đối tượng Life (ô đang chết)
post: Ô có tọa độ (row,col) được bổ sung vào đối tượng Life (ô trở nên sống). Nếu việc
      thêm vào List hoặc Hash_table không thành công thì lỗi sẽ được trả về.
uses: Các lớp List, Hash_table, và cấu trúc Cell.
*/
{
   Error_code outcome;
   Cell *new_cell = new Cell(row, col);

   int index = living->size();
   outcome = living->insert(index, new_cell);
   if (outcome == success)
      outcome = is_living->insert(new_cell);
   if (outcome != success)
      cout << " Warning: new Cell insertion failed" << endl;
   return outcome;
}
```

Constructor và *destructor* cho các đối tượng `Life`

Chúng ta cần cung cấp *constructor* và *destructor* cho lớp `Life` của chúng ta để định vị và giải phóng các thành phần cấp phát động của nó. *Constructor* cần thực hiện toán tử new cho các thuộc tính con trỏ.

```
Life::Life()
/*
post: Các thuộc tính thành phần của đối tượng Life được cấp phát động và được khởi tạo.
uses: Các lớp Hash_table, List.
*/
{
   living = new List<Cell *>;
   is_living = new Hash_table;
}
```

Destructor cần giải phóng mọi phần tử được cấp phát động bởi bất kỳ một phương thức nào đó của lớp `Life`. Bên cạnh hai con trỏ **`living`** và **`is_living`** được khởi tạo nhờ *constructor* còn có các đối tượng **`Cell`** mà chúng tham chiếu đến đã được cấp phát động trong phương thức **`insert`**. *Destructor* cần giải phóng các đối tượng **`Cell`** này trước khi giải phóng **`living`** và **`is_living`**.

```
Life::~Life()
/*
post: Các thuộc tính cấp phát động của đối tượng Life và các đối tượng do chúng tham chiếu
      được giải phóng.
uses: Các lớp Hash_table, List.
*/
{
   Cell *old_cell;
   for (int i = 0; i < living->size(); i++) {
      living->retrieve(i, old_cell);
      delete old_cell;
   }
   delete is_living;          //  Gọi destructor của Hash_table.
   delete living;             //  Gọi destructor của List.
}
```

Đối với những cấu trúc có nhiều liên kết bằng con trỏ như thế này, chúng ta luôn phải đặt vấn đề về khả năng tạo rác do những sơ suất của chúng ta. Thông thường thì *destructor* của một lớp luôn làm tốt nhiệm vụ của nó, nhưng nó chỉ dọn dẹp những gì thuộc đối tượng của nó, chứ không biết đến những gì mà đối tượng của nó tham chiếu đến. Nếu chúng ta chủ quan, việc dọn dẹp không diễn ra đúng như chúng ta tưởng. Khi chạy, chương trình có thể là quên dọn dẹp, cũng có thể là dọn dẹp nhiều hơn một lần đối với một vùng nhớ được cấp phát động. Đây cũng là một vấn đề khá lý thú mà sinh viên nên tự suy nghĩ thêm.

Hàm băm

Hàm băm ở đây có hơi khác với những gì chúng ta đã gặp trong những phần trước, thông số của nó có đến hai thành phần (`row` và `col`), nhờ vậy chúng ta có thể dễ dàng sử dụng một vài dạng của phép trộn. Trước khi quyết định nên làm như thế nào, chúng ta hãy xét trường hợp một mảng chữ nhật nhỏ có ánh xạ một một dưới đây chính là một hàm chỉ số. Có chính xác là **`maxrow`** phần tử trong mỗi hàng, các chỉ số i, j được ánh xạ đến `i + maxrow*j` để đặt mảng chữ nhật vào một chuỗi các vùng nhớ liên tục, hàng này kế tiếp hàng kia.

Chúng ta nên dùng cách ánh xạ tương tự cho hàm băm của chúng ta và sẽ thay thế `maxrow` bằng một số thích hợp, chẳng hạn như một số nguyên tố, để việc phân phối được rải đều và giảm sự đụng độ.

```
const int factor = 101;

int hash(int row, int col)
/*
post: Trả về giá trị hàm băm từ 0 đến hash_size - 1 tương ứng với tọa độ (row, col).
*/
{
   int value;
   value = row + factor * col;
   value %= hash_size;
   if (value < 0) return value + hash_size;
   else return value;
}
```

Các chương trình con khác

Các phương thức còn lại cuûa `Life` như `initialize`, `retrieve`, và `neighbor_count` đều được xem xét tương tự như các hàm vừa rồi hoặc như các hàm tương ứng trong phiên bản thứ nhất. Chúng ta dành chúng lại như bài tập.