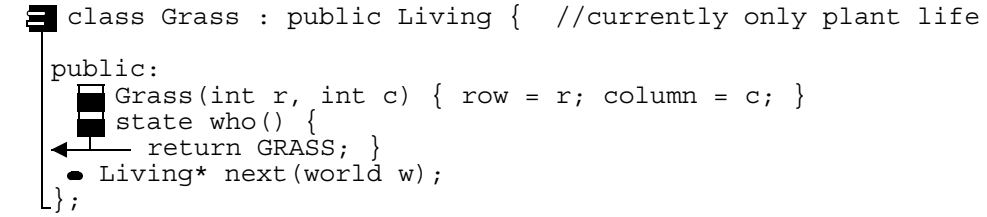
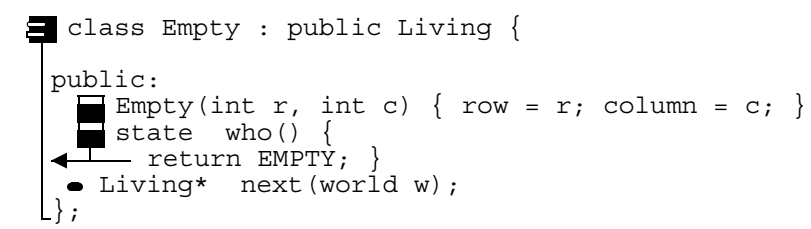
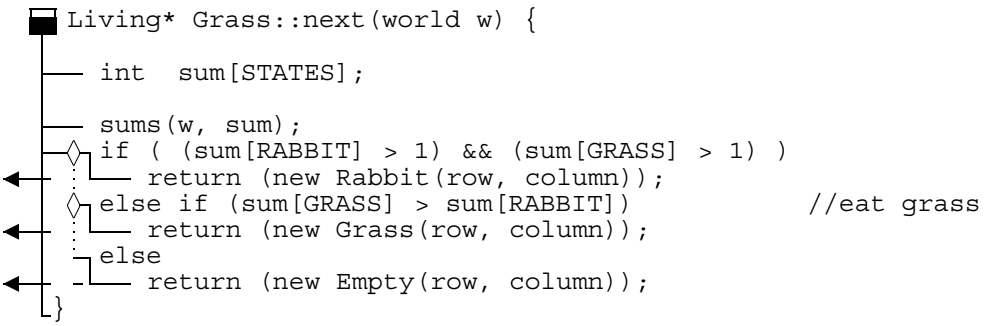
```
/*********************************************************************
  Filename:  predator.cpp (modified version)
  C++ for C Programmers, Edition 3     By Ira Pohl
  SOURCE CODE ANNOTATED WITH CSD
*********************************************************************/

#include <iostream.h>
#include <stdlib.h>

//Predator-Prey simulation using class living

 const int  N = 19;                  //size of square board

 enum state { EMPTY , GRASS , RABBIT , FOX, STATES };

 const int DRAB = 3, DFOX = 6, CYCLES = 40;

 class Living;                       //forward declaration
 typedef Living* world[N][N];

 class Living {                       //what lives in world

 public:
  virtual state  who() = 0;          //state identification
  virtual Living*  next(world w) = 0;

 protected:
  int  row, column;                  //location
  void sums(world w,int sm[]);
 };


 void Living::sums(world w, int sm[]) {
   int  i, j;

   sm[EMPTY] = sm[GRASS] = sm[RABBIT] = sm[FOX] = 0;
   for (i = -1; i <= 1; ++i)
     for ( j = -1; j <= 1; ++j)
       sm[w[row + i][column +j] -> who()]++;
 }


 class Fox : public Living {  //currently only predator class

 public:
   Fox(int r, int c, int a = 0) : age(a)
    { row = r; column = c; }
   state  who() {
     return FOX; }   //deferred method for foxes
  Living*  next(world w);

 protected:
  int  age;                      //used to decide on dying
 };


 class Rabbit : public Living { //currently only prey class

 public:
   Rabbit(int r, int c, int a = 0) : age(a)
    { row = r; column = c; }
   state  who() {
     return RABBIT; }
  Living*  next(world w);

 protected:
  int  age;
 };
```



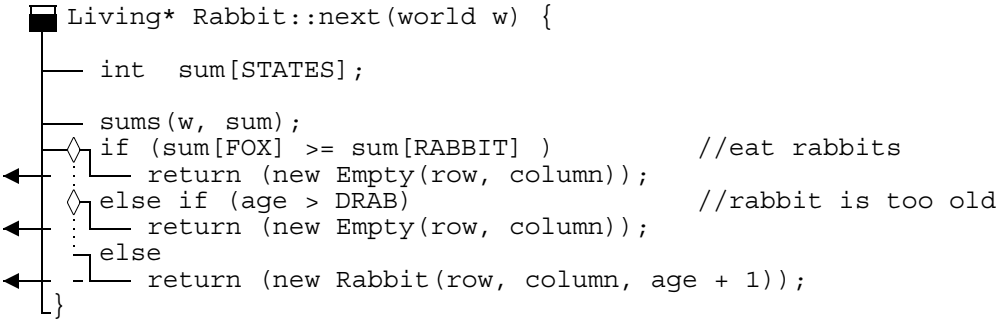
```
class Grass : public Living {  //currently only plant life

public:
  Grass(int r, int c) { row = r; column = c; }
  state who() {
    return GRASS; }
  Living* next(world w);
};
```



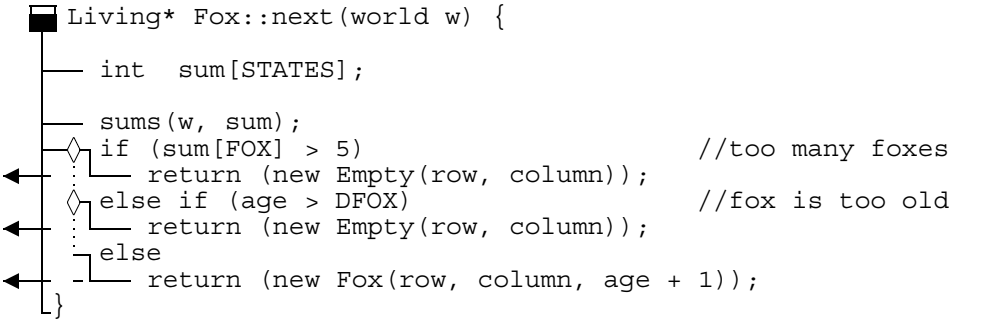
```
//nothing lives here

class Empty : public Living {

public:
  Empty(int r, int c) { row = r; column = c; }
  state  who() {
    return EMPTY; }
  Living*  next(world w);
};
```



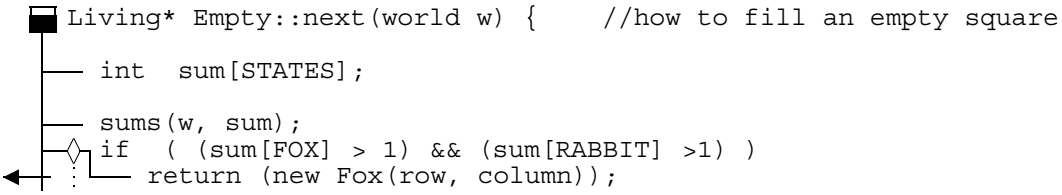
```
Living* Grass::next(world w) {

  int  sum[STATES];

  sums(w, sum);
  if ( (sum[RABBIT] > 1) && (sum[GRASS] > 1) )
    return (new Rabbit(row, column));
  else if (sum[GRASS] > sum[RABBIT])           //eat grass
    return (new Grass(row, column));
  else
    return (new Empty(row, column));
}
```



```
Living* Rabbit::next(world w) {

  int  sum[STATES];

  sums(w, sum);
  if (sum[FOX] >= sum[RABBIT] )          //eat rabbits
    return (new Empty(row, column));
  else if (age > DRAB)                   //rabbit is too old
    return (new Empty(row, column));
  else
    return (new Rabbit(row, column, age + 1));
}
```



```
Living* Fox::next(world w) {

  int  sum[STATES];

  sums(w, sum);
  if (sum[FOX] > 5)                      //too many foxes
    return (new Empty(row, column));
  else if (age > DFOX)                   //fox is too old
    return (new Empty(row, column));
  else
    return (new Fox(row, column, age + 1));
}
```



```
Living* Empty::next(world w) {    //how to fill an empty square

  int  sum[STATES];

  sums(w, sum);
  if  ( (sum[FOX] > 1) && (sum[RABBIT] >1) )
    return (new Fox(row, column));
```

```
    else if ( (sum[RABBIT] > 1) && (sum[GRASS] > 1) )
        return (new Rabbit(row, column));
    else if (sum[GRASS])
        return (new Grass(row, column));
    else
        return (new Empty(row, column));
}
```

```
void init(world w) {   //world is all empty
    int  i, j;

    for (i = 0; i < N; ++i)
        for (j = 0; j < N; ++j)
            w[i][j] = new Empty(i,j);
}
```

```
//new world w_new is computed from old world w_old

void update(world w_new, world w_old) {
    int  i, j;

    for (i = 1; i < N - 1; ++i)          //borders are taboo
        for (j = 1; j < N - 1; ++j)
            w_new[i][j] = w_old[i][j] -> next(w_old);
}
```

```
void dele(world w) {   //clean world up
    int  i, j;

    for (i = 1; i < N - 1; ++i)
        for (j = 1; j < N - 1; ++j)
            delete(w[i][j]);
}
```

```
//********* one way to do exercise

void random_eden(world w) {
    int i,j,rand_n;

    for (i = 2; i < N - 2; ++i)
        for (j = 2; j < N - 2; ++j) {
            rand_n= rand(); //stdlib function, returning int within [0, RAND_MAX]
            rand_n= rand_n%6;
            switch(rand_n) {
                case 0: w[i][j] = new Empty(i, j);
                    break;
                case 1: case 2:
                case 3: w[i][j] = new Rabbit(i, j);
                    break;
                case 4: w[i][j] = new Grass(i, j);
                    break;
                case 5: w[i][j] = new Fox(i, j);
                    break;
            }
        }
}
```

```
void print_state(world w) {
    int i, j, a;

    for(i = 0; i < N; ++i) {
        cout << endl;
```

```
    for(j = 0; j < N; ++j) {
      a = (int)(w[i][j] -> who());
      switch(a) {
        case EMPTY:  cout << ".";
          break;
        case GRASS:  cout << "_";
          break;
        case RABBIT: cout << "r";
          break;
        case FOX:    cout << "F";
          break;
      }
    }
  }
  cout << endl;
  cout << endl;
  cin >> i;
}

//*********************************

void main() {

  world  odd, even;
  int    i;

  init(odd);  init(even);
  //eden(even);                    //generate initial world
  random_eden(even);
  print_state(even);                 //print Garden of Eden state

  for (i = 0; i < CYCLES; ++i) {     //simulation
    if (i % 2) {
      update(even, odd);
      print_state(even);
      dele(odd);
    }
    else {
      update(odd, even);
      print_state(odd);
      dele(even);
    }
  }

  int look; cin >> look;
}
```