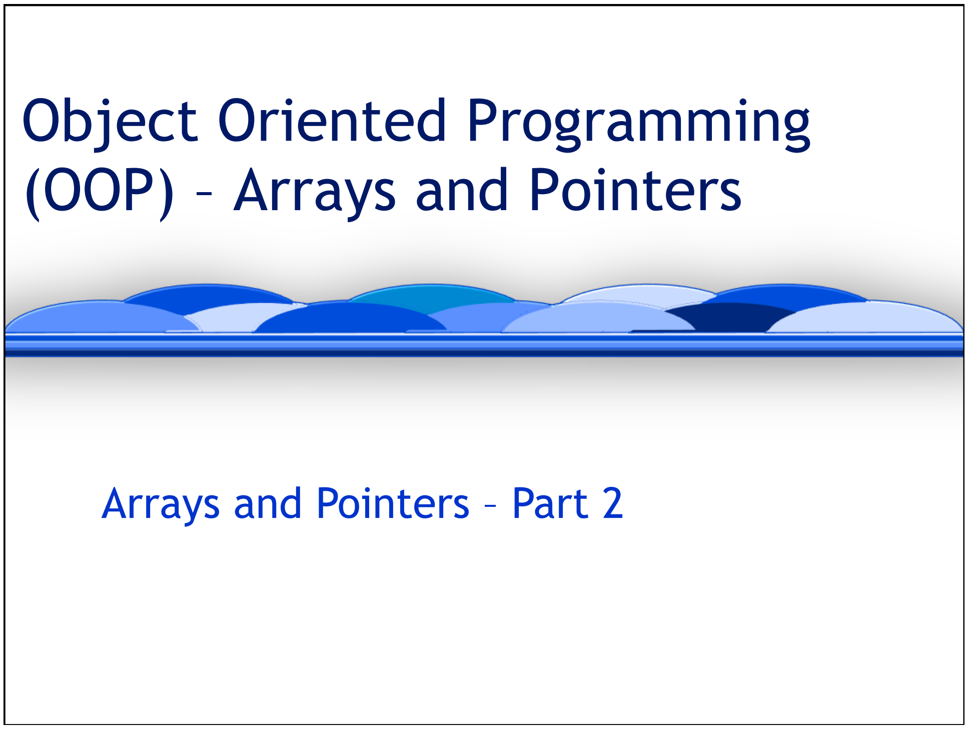
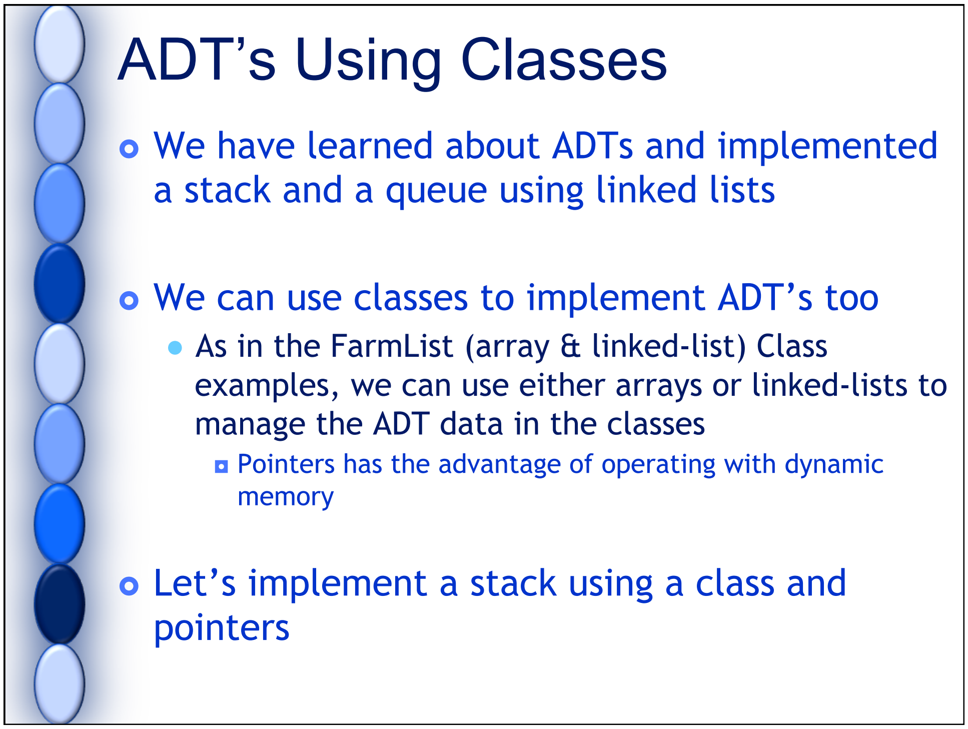


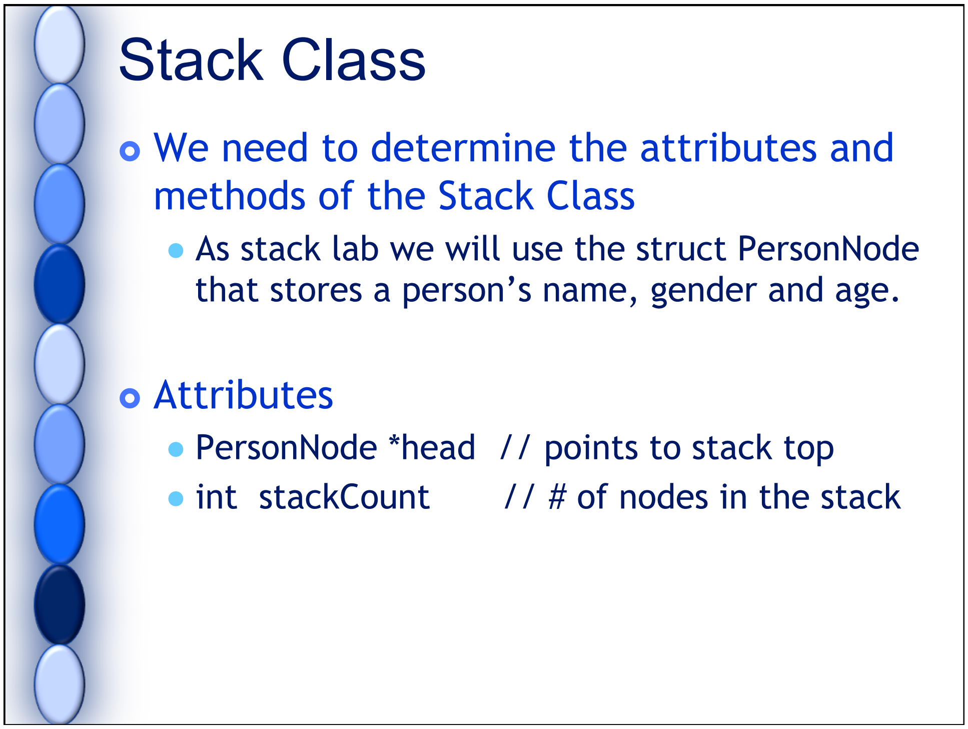
Object Oriented Programming (OOP) – Arrays and Pointers

Arrays and Pointers – Part 2



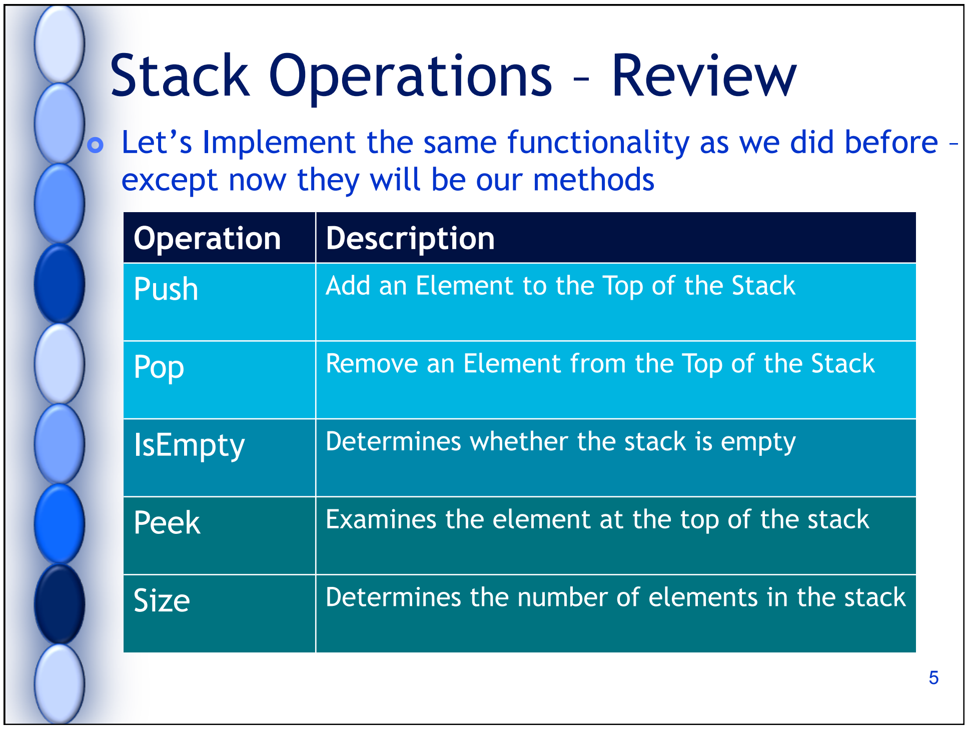
ADT's Using Classes

- We have learned about ADTs and implemented a stack and a queue using linked lists
- We can use classes to implement ADT's too
  - As in the FarmList (array & linked-list) Class examples, we can use either arrays or linked-lists to manage the ADT data in the classes
    - Pointers has the advantage of operating with dynamic memory
- Let's implement a stack using a class and pointers



Stack Class

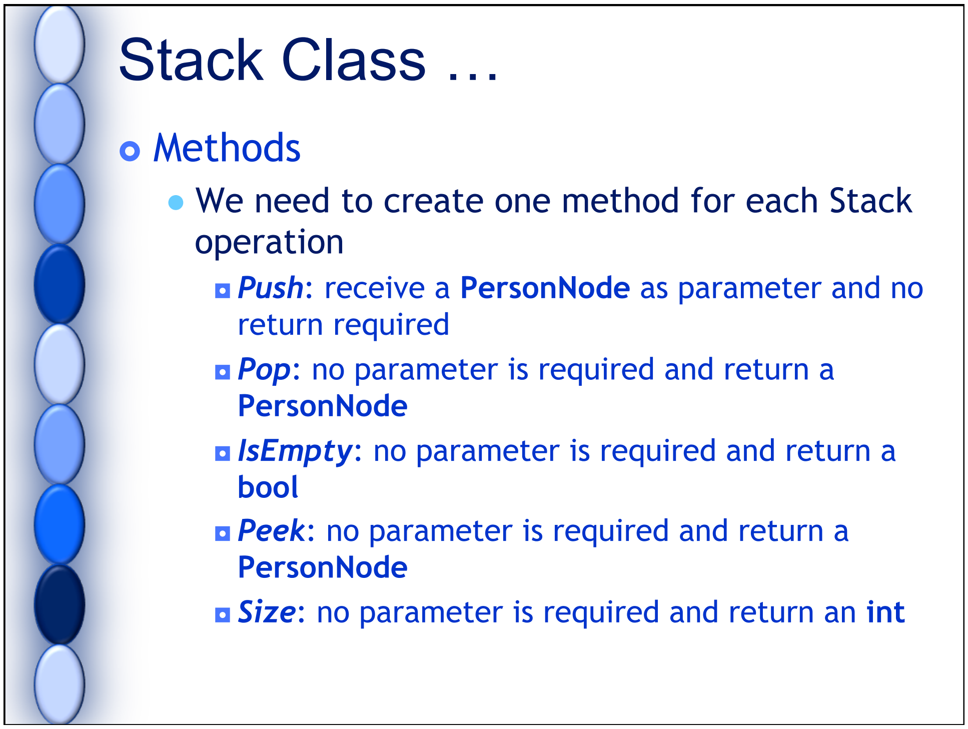
- We need to determine the attributes and methods of the Stack Class
  - As stack lab we will use the struct PersonNode that stores a person’s name, gender and age.
- Attributes
  - PersonNode *head // points to stack top
  - int stackCount   // # of nodes in the stack



Stack Operations – Review

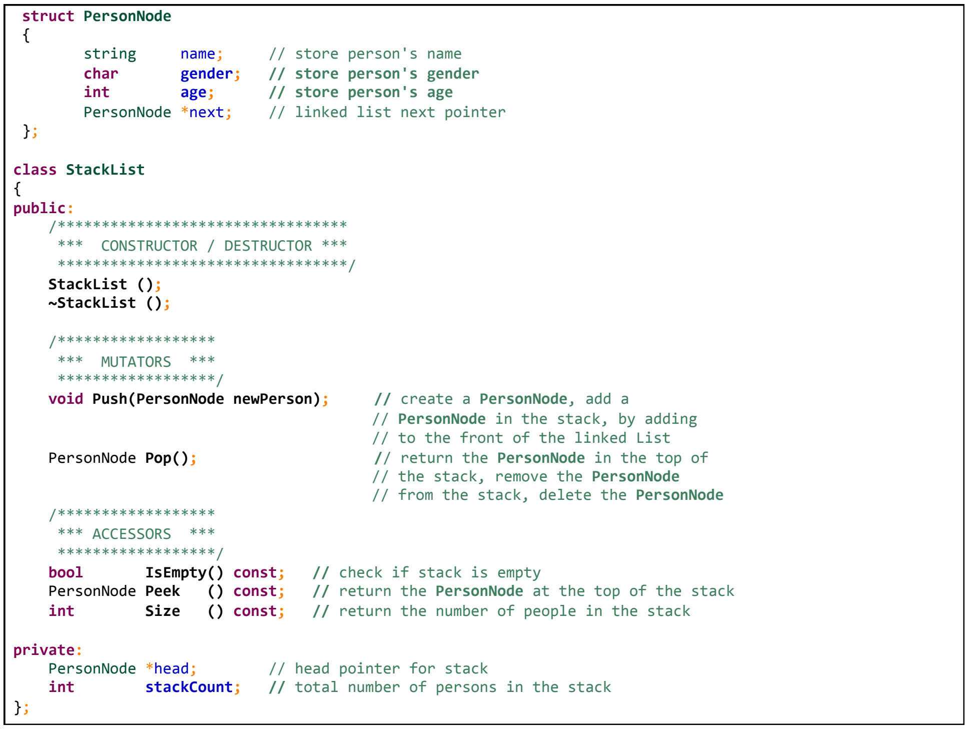
- Let’s Implement the same functionality as we did before – except now they will be our methods

Operation	Description
Push	Add an Element to the Top of the Stack
Pop	Remove an Element from the Top of the Stack
IsEmpty	Determines whether the stack is empty
Peek	Examines the element at the top of the stack
Size	Determines the number of elements in the stack

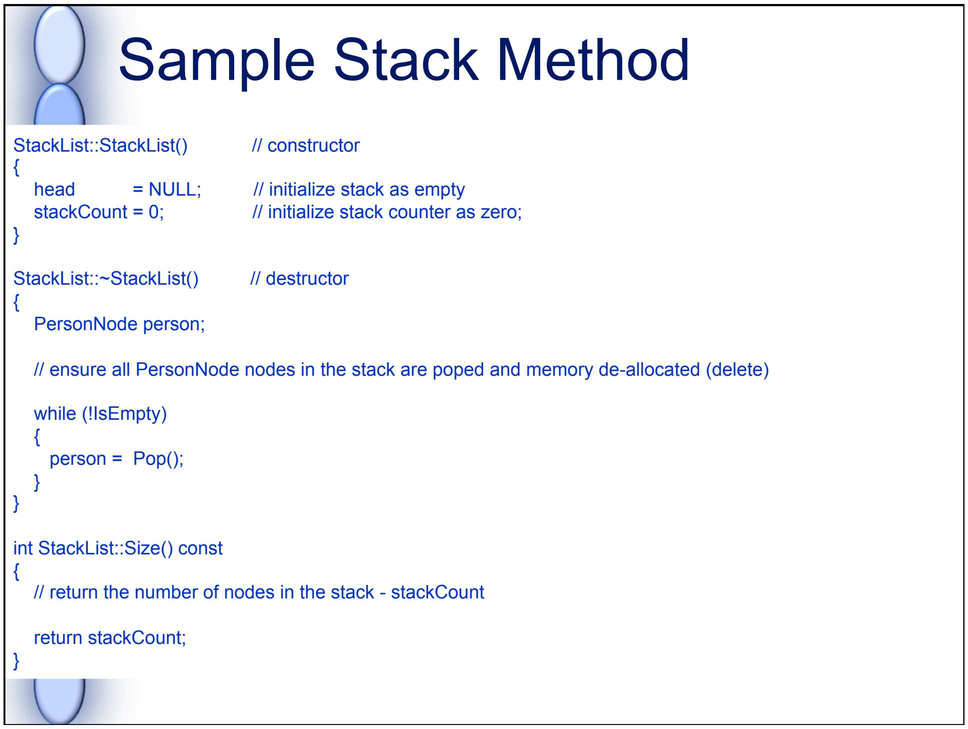


Stack Class …

- Methods
  - We need to create one method for each Stack operation
    - ***Push***: receive a **PersonNode** as parameter and no return required
    - ***Pop***: no parameter is required and return a **PersonNode**
    - ***IsEmpty***: no parameter is required and return a **bool**
    - ***Peek***: no parameter is required and return a **PersonNode**
    - ***Size***: no parameter is required and return an **int**

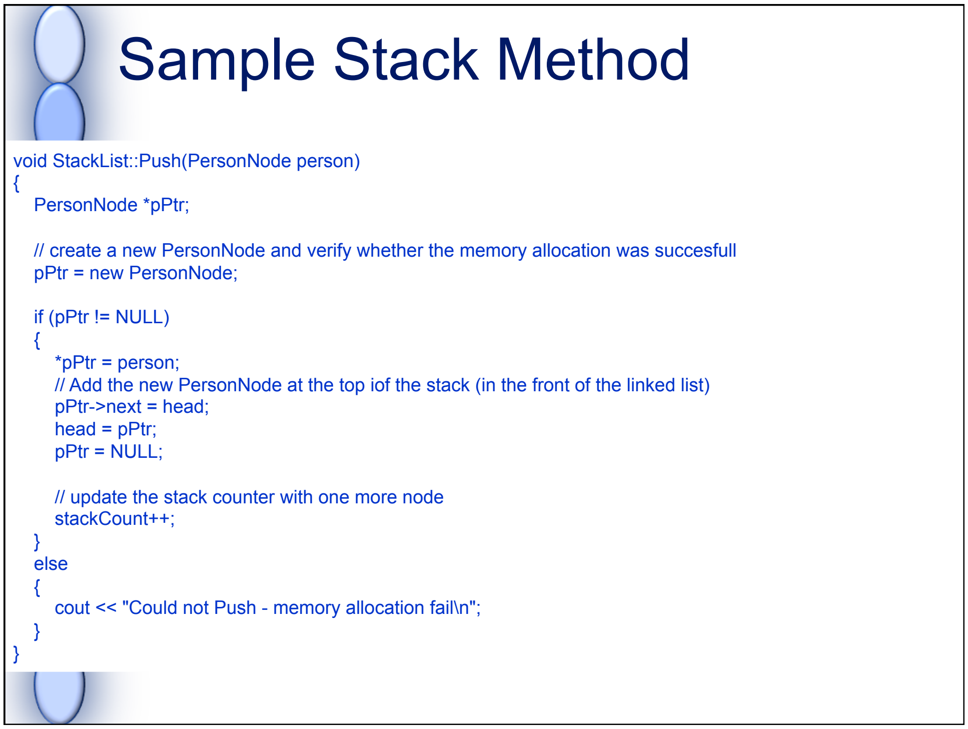


```
 struct PersonNode
 {
        string     name;     // store person's name
        char       gender;   // store person's gender
        int        age;      // store person's age
        PersonNode *next;    // linked list next pointer
 };

class StackList
{
public:
    /**********************************
     ***  CONSTRUCTOR / DESTRUCTOR ***
     **********************************/
    StackList ();
    ~StackList ();

    /******************
     ***  MUTATORS  ***
     ******************/
    void Push(PersonNode newPerson);     // create a PersonNode, add a
                                         // PersonNode in the stack, by adding
                                         // to the front of the linked List

    PersonNode Pop();                    // return the PersonNode in the top of
                                         // the stack, remove the PersonNode
                                         // from the stack, delete the PersonNode

    /******************
     *** ACCESSORS  ***
     ******************/
    bool       IsEmpty() const;   // check if stack is empty
    PersonNode Peek   () const;   // return the PersonNode at the top of the stack
    int        Size   () const;   // return the number of people in the stack

private:
    PersonNode *head;       // head pointer for stack
    int        stackCount;  // total number of persons in the stack
};
```



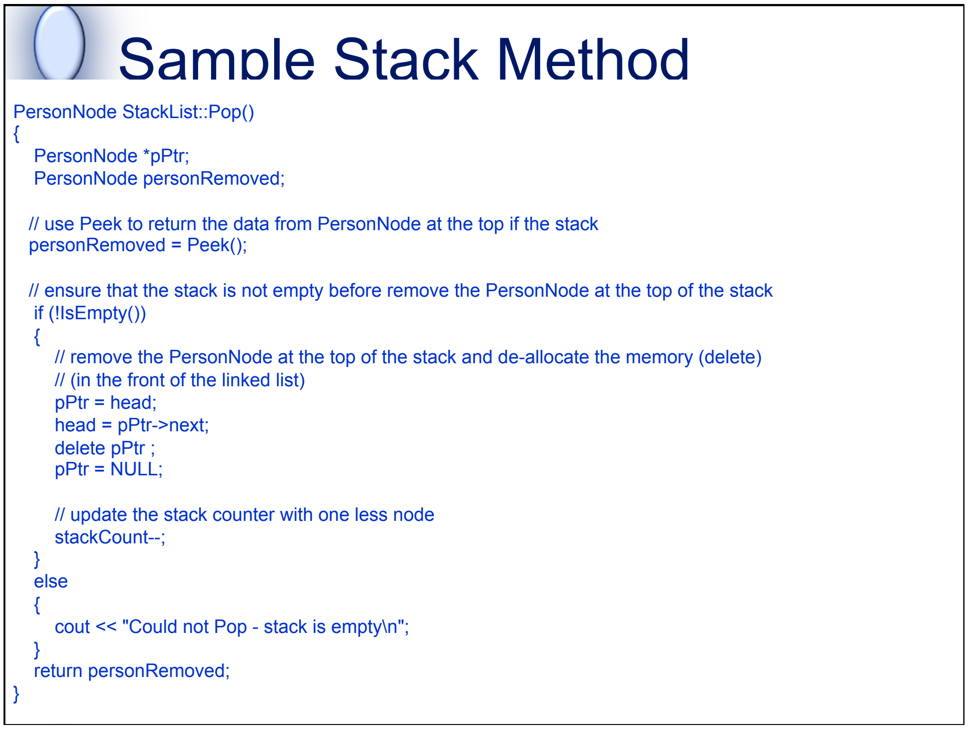
Sample Stack Method

```
StackList::StackList()          // constructor
{
   head         = NULL;         // initialize stack as empty
   stackCount = 0;              // initialize stack counter as zero;
}

StackList::~StackList()         // destructor
{
   PersonNode person;

   // ensure all PersonNode nodes in the stack are poped and memory de-allocated (delete)

   while (!IsEmpty)
   {
     person =  Pop();
   }
}

int StackList::Size() const
{
   // return the number of nodes in the stack - stackCount

   return stackCount;
}
```



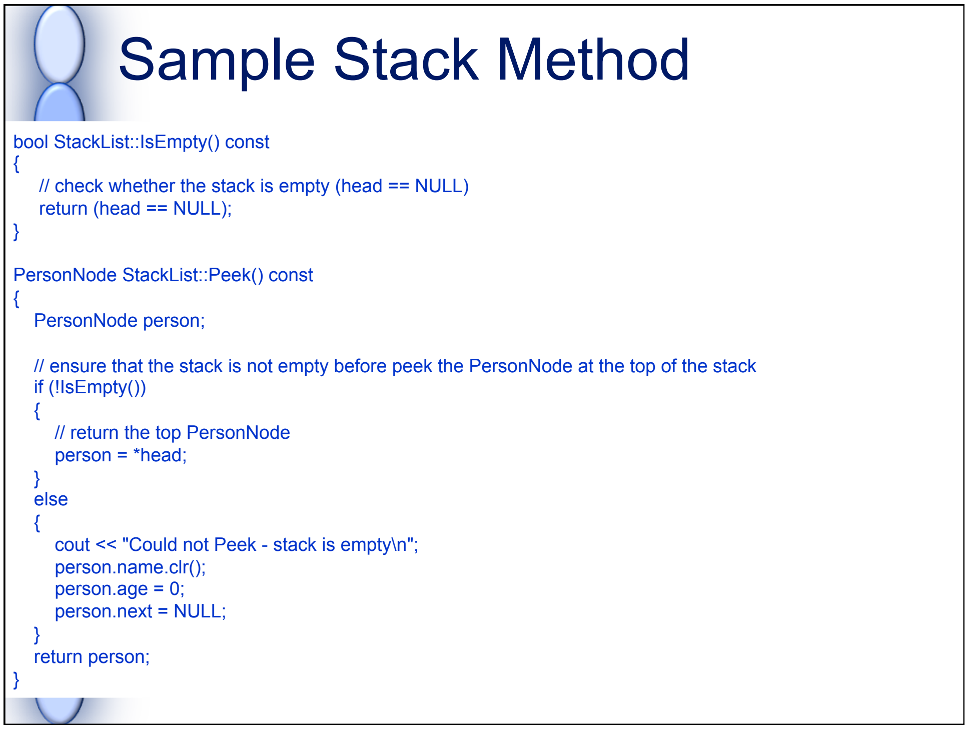
Sample Stack Method

```
void StackList::Push(PersonNode person)
{
   PersonNode *pPtr;

   // create a new PersonNode and verify whether the memory allocation was succesfull
   pPtr = new PersonNode;

   if (pPtr != NULL)
   {
      *pPtr = person;
      // Add the new PersonNode at the top iof the stack (in the front of the linked list)
      pPtr->next = head;
      head = pPtr;
      pPtr = NULL;

      // update the stack counter with one more node
      stackCount++;
   }
   else
   {
      cout << "Could not Push - memory allocation fail\n";
   }
}
```



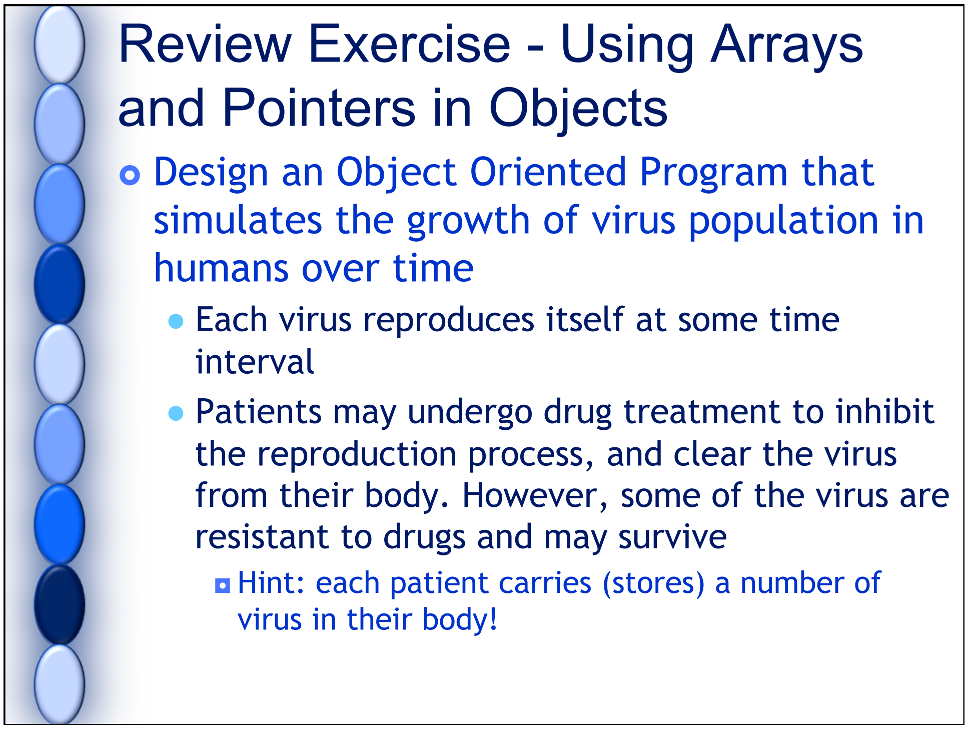
Sample Stack Method

```
PersonNode StackList::Pop()
{
   PersonNode *pPtr;
   PersonNode personRemoved;

  // use Peek to return the data from PersonNode at the top if the stack
  personRemoved = Peek();

  // ensure that the stack is not empty before remove the PersonNode at the top of the stack
   if (!IsEmpty())
   {
      // remove the PersonNode at the top of the stack and de-allocate the memory (delete)
      // (in the front of the linked list)
      pPtr = head;
      head = pPtr->next;
      delete pPtr ;
      pPtr = NULL;

      // update the stack counter with one less node
      stackCount--;
   }
   else
   {
      cout << "Could not Pop - stack is empty\n";
   }
   return personRemoved;
}
```



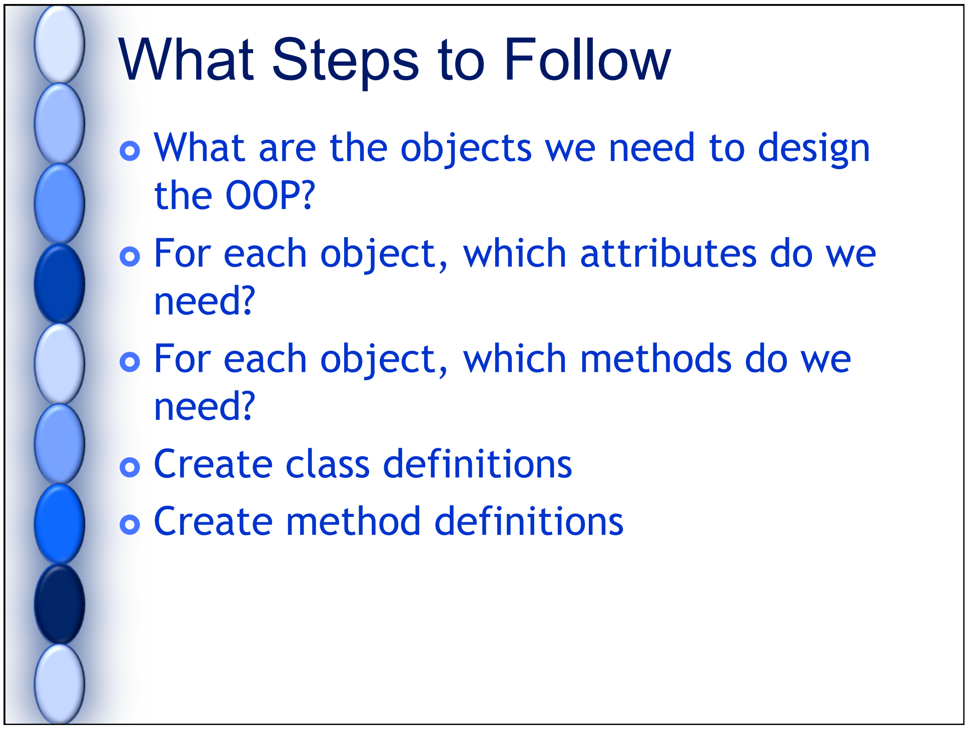
Sample Stack Method

```
bool StackList::IsEmpty() const
{
    // check whether the stack is empty (head == NULL)
    return (head == NULL);
}

PersonNode StackList::Peek() const
{
   PersonNode person;

   // ensure that the stack is not empty before peek the PersonNode at the top of the stack
   if (!IsEmpty())
   {
      // return the top PersonNode
      person = *head;
   }
   else
   {
      cout << "Could not Peek - stack is empty\n";
      person.name.clr();
      person.age = 0;
      person.next = NULL;
   }
   return person;
}
```



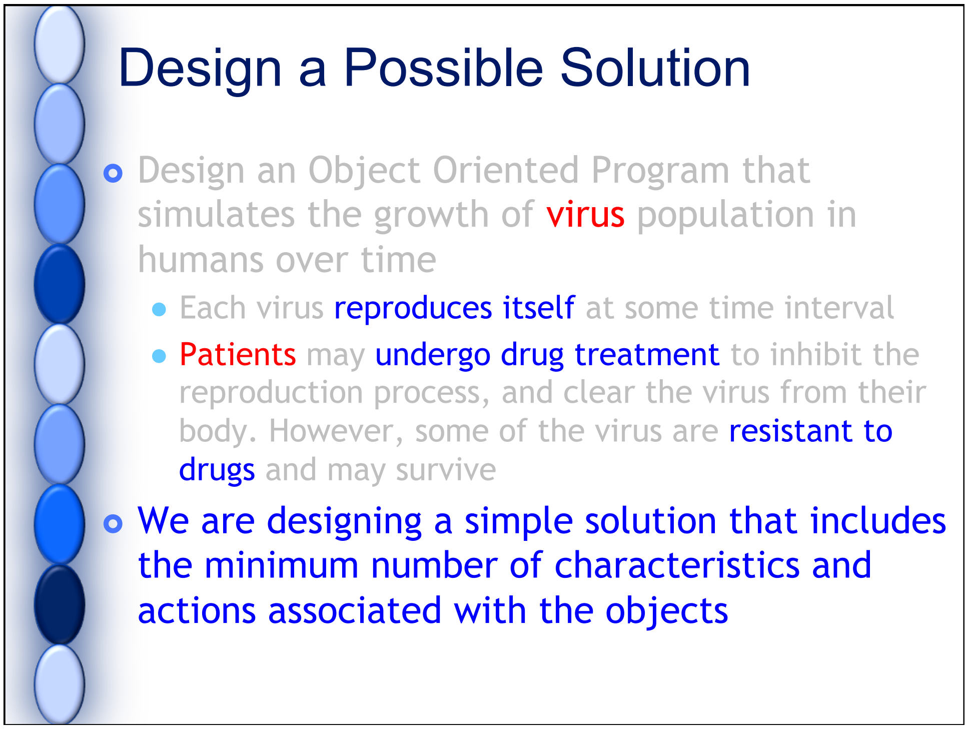
Review Exercise - Using Arrays and Pointers in Objects

- Design an Object Oriented Program that simulates the growth of virus population in humans over time
  - Each virus reproduces itself at some time interval
  - Patients may undergo drug treatment to inhibit the reproduction process, and clear the virus from their body. However, some of the virus are resistant to drugs and may survive
    - Hint: each patient carries (stores) a number of virus in their body!



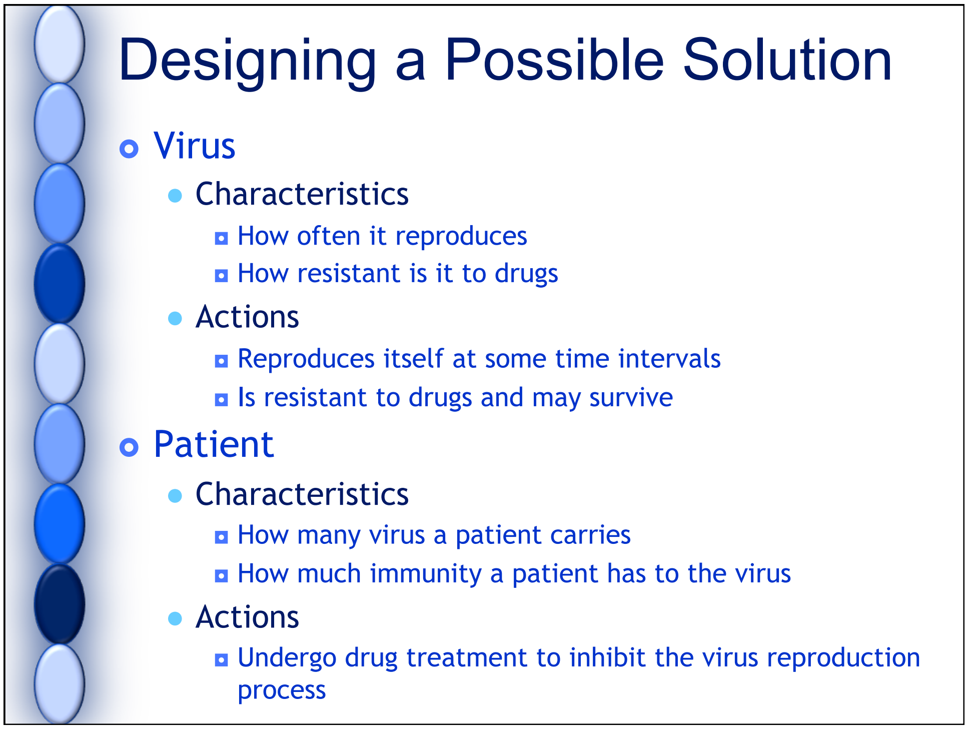
What Steps to Follow

- What are the objects we need to design the OOP?
- For each object, which attributes do we need?
- For each object, which methods do we need?
- Create class definitions
- Create method definitions



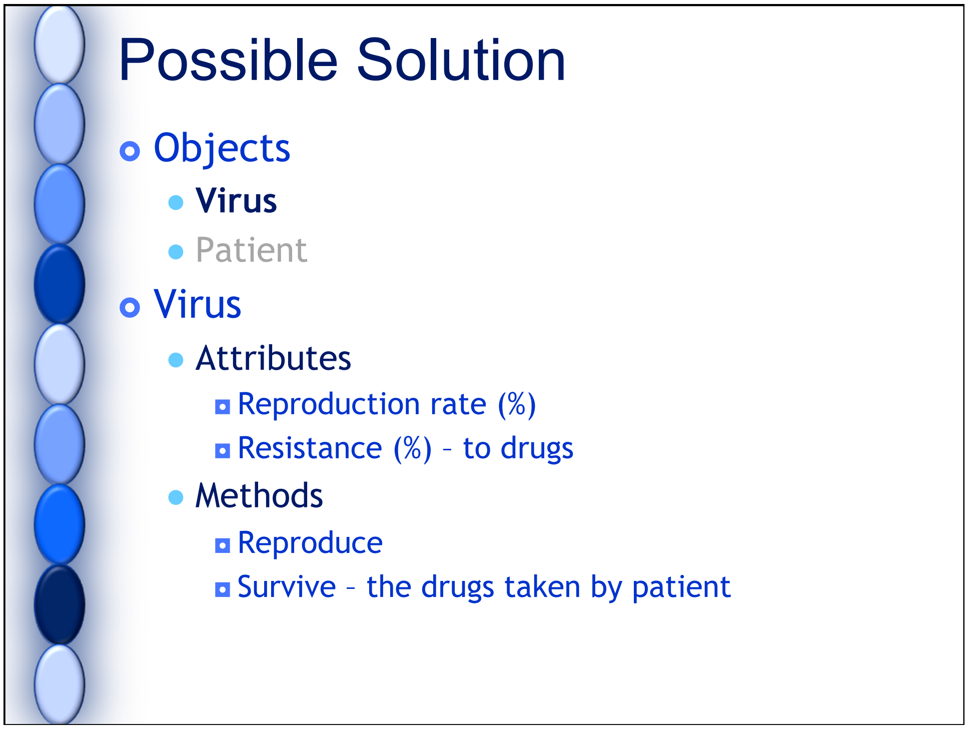
Design a Possible Solution

- Design an Object Oriented Program that simulates the growth of virus population in humans over time
  - Each virus reproduces itself at some time interval
  - Patients may undergo drug treatment to inhibit the reproduction process, and clear the virus from their body. However, some of the virus are resistant to drugs and may survive
- We are designing a simple solution that includes the minimum number of characteristics and actions associated with the objects



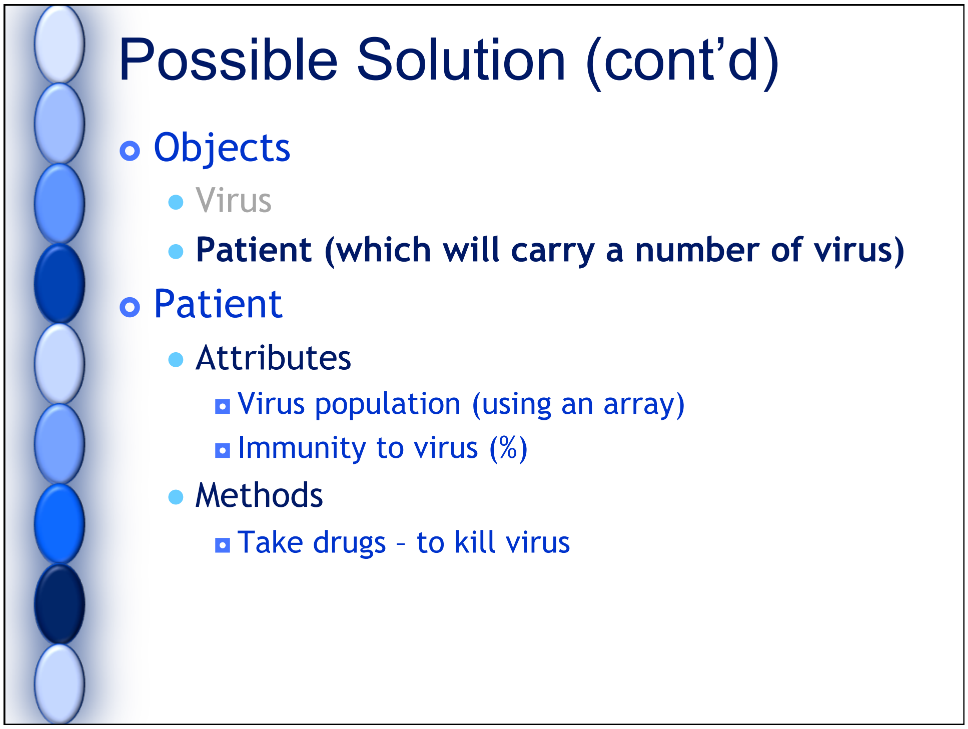
Designing a Possible Solution

- Virus
  - Characteristics
    - How often it reproduces
    - How resistant is it to drugs
  - Actions
    - Reproduces itself at some time intervals
    - Is resistant to drugs and may survive
- Patient
  - Characteristics
    - How many virus a patient carries
    - How much immunity a patient has to the virus
  - Actions
    - Undergo drug treatment to inhibit the virus reproduction process



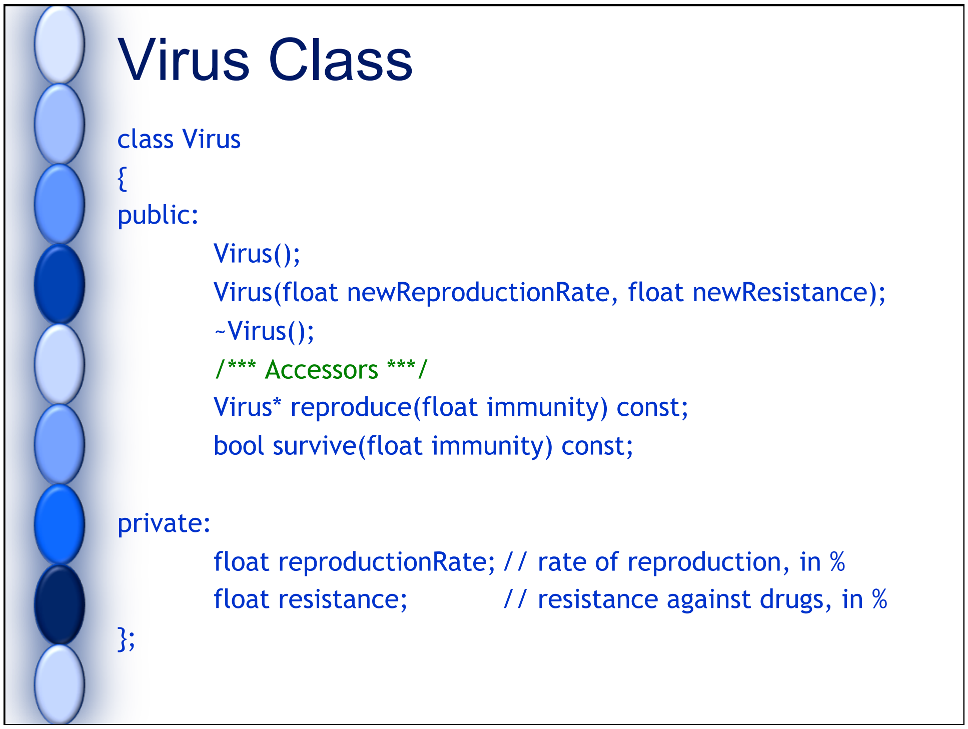
Possible Solution

- Objects
  - **Virus**
  - Patient
- Virus
  - Attributes
    - Reproduction rate (%)
    - Resistance (%) - to drugs
  - Methods
    - Reproduce
    - Survive - the drugs taken by patient



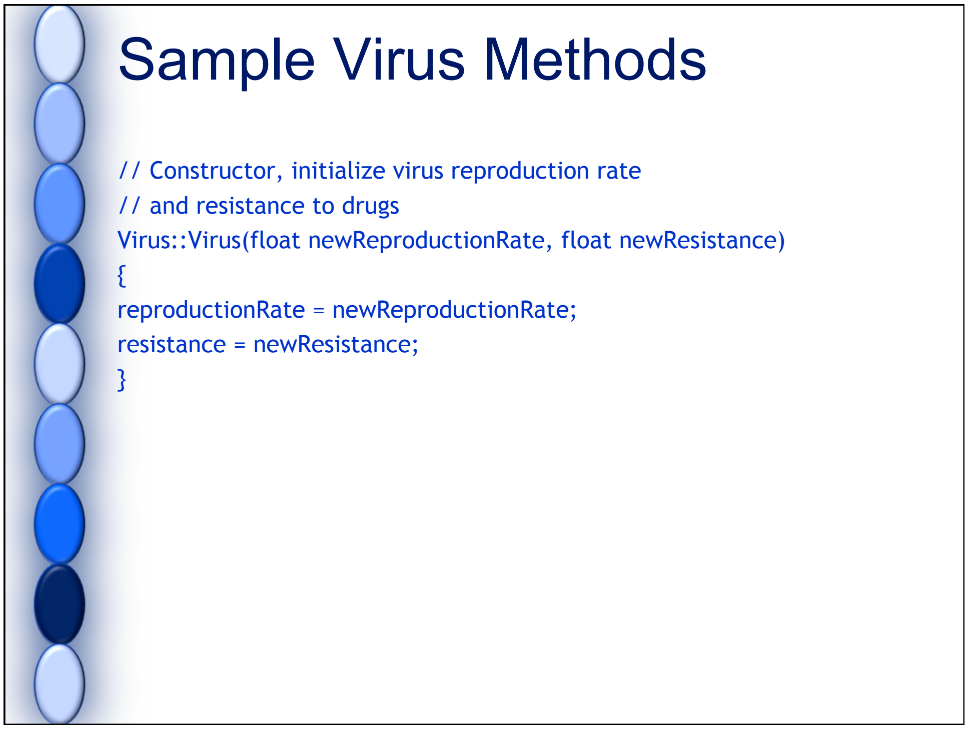
Possible Solution (cont'd)

- Objects
  - Virus
  - **Patient (which will carry a number of virus)**
- Patient
  - Attributes
    - Virus population (using an array)
    - Immunity to virus (%)
  - Methods
    - Take drugs - to kill virus



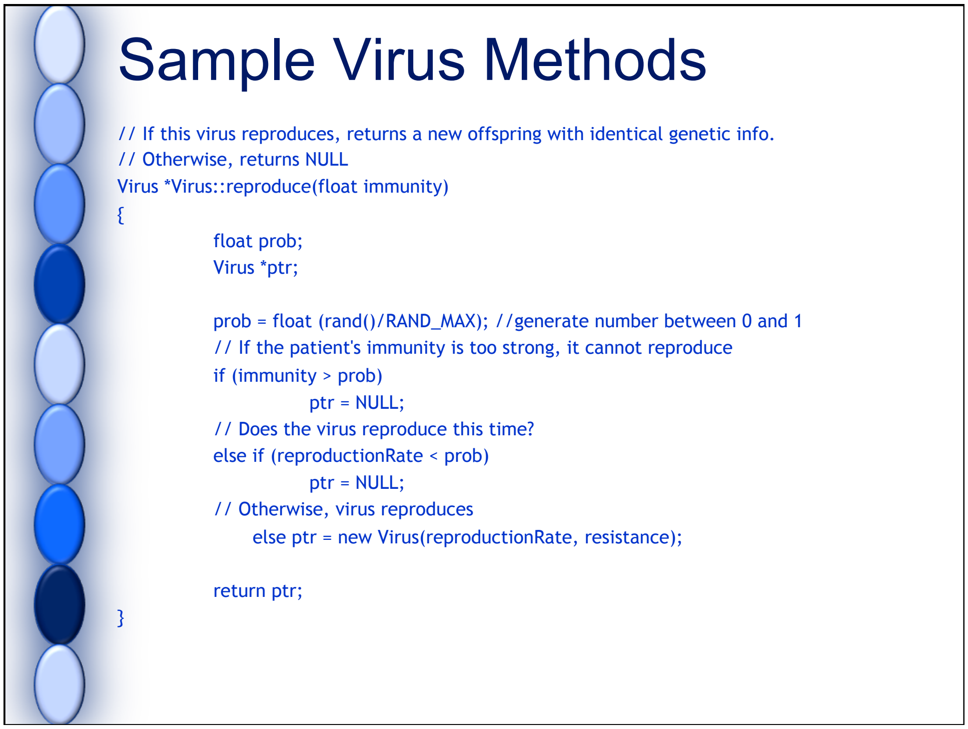
Virus Class

```
class Virus
{
public:
        Virus();
        Virus(float newReproductionRate, float newResistance);
        ~Virus();
        /*** Accessors ***/
        Virus* reproduce(float immunity) const;
        bool survive(float immunity) const;

private:
        float reproductionRate; // rate of reproduction, in %
        float resistance;       // resistance against drugs, in %
};
```



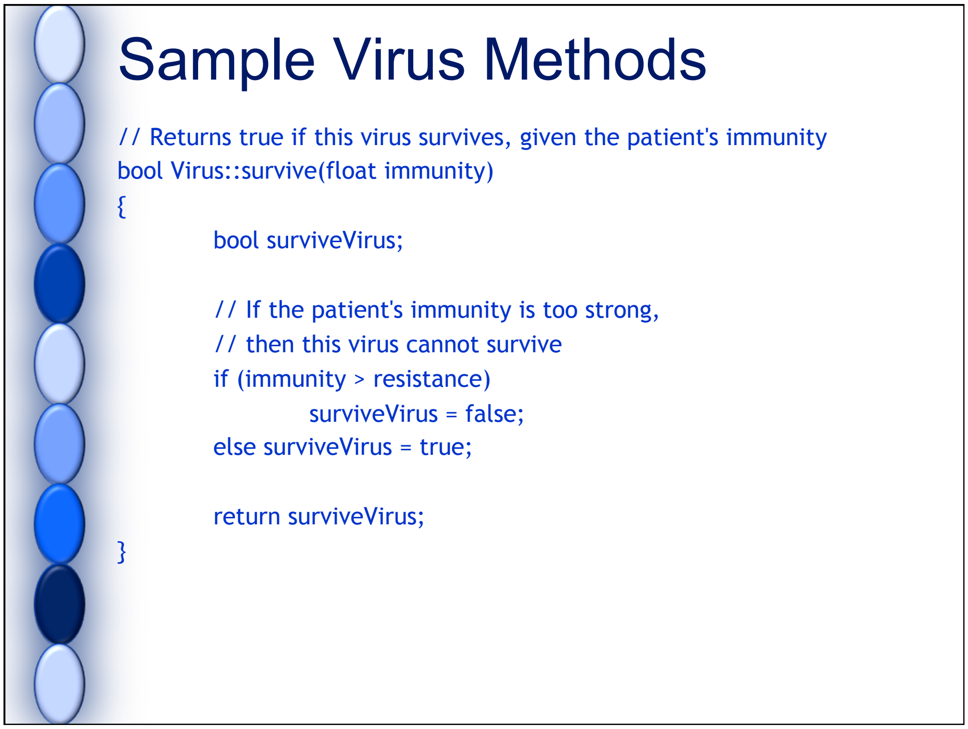
Sample Virus Methods

```
// Constructor, initialize virus reproduction rate
// and resistance to drugs
Virus::Virus(float newReproductionRate, float newResistance)
{
reproductionRate = newReproductionRate;
resistance = newResistance;
}
```



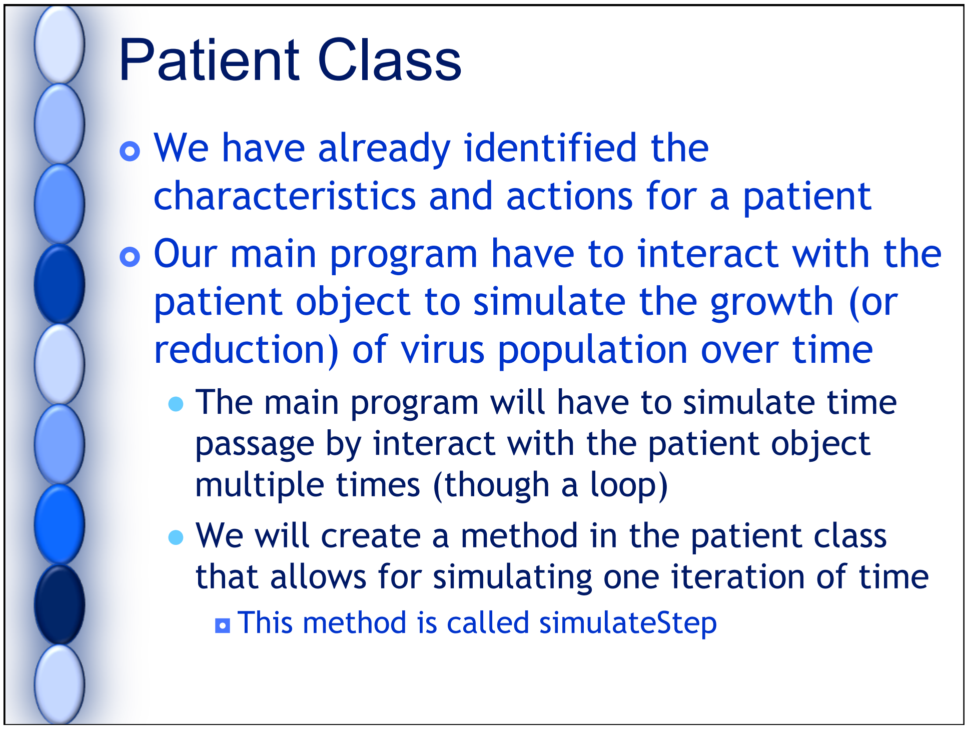
Sample Virus Methods

```
// If this virus reproduces, returns a new offspring with identical genetic info.
// Otherwise, returns NULL
Virus *Virus::reproduce(float immunity)
{
        float prob;
        Virus *ptr;

        prob = float (rand()/RAND_MAX); //generate number between 0 and 1
        // If the patient's immunity is too strong, it cannot reproduce
        if (immunity > prob)
                ptr = NULL;
        // Does the virus reproduce this time?
        else if (reproductionRate < prob)
                ptr = NULL;
        // Otherwise, virus reproduces
            else ptr = new Virus(reproductionRate, resistance);

        return ptr;
}
```



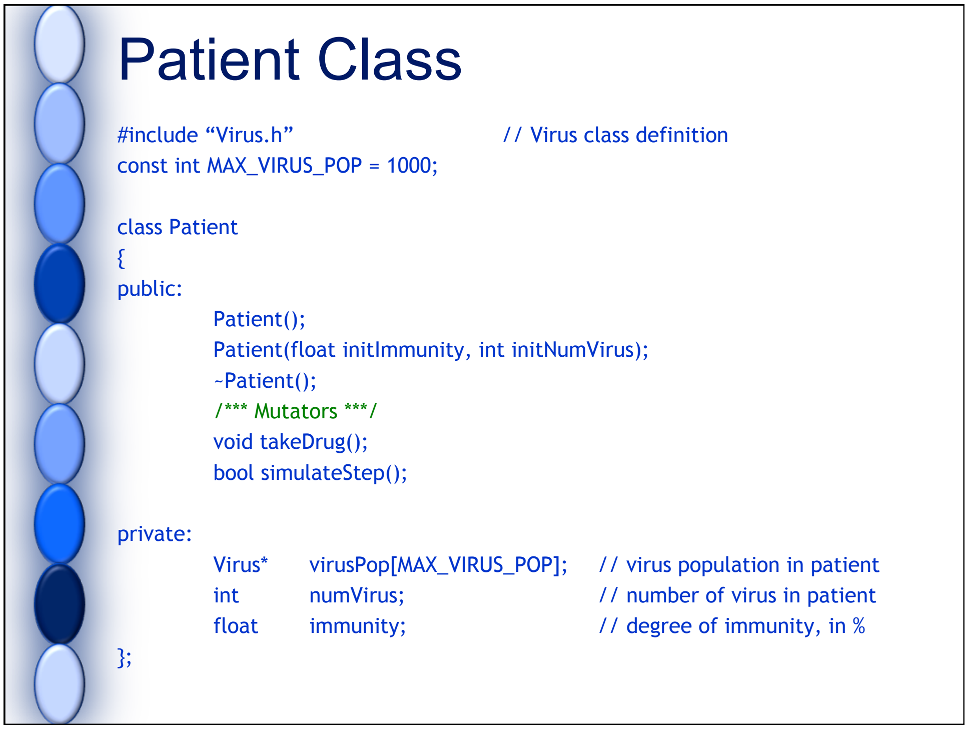
Sample Virus Methods

```
// Returns true if this virus survives, given the patient's immunity
bool Virus::survive(float immunity)
{
        bool surviveVirus;

        // If the patient's immunity is too strong,
        // then this virus cannot survive
        if (immunity > resistance)
                surviveVirus = false;
        else surviveVirus = true;

        return surviveVirus;
}
```



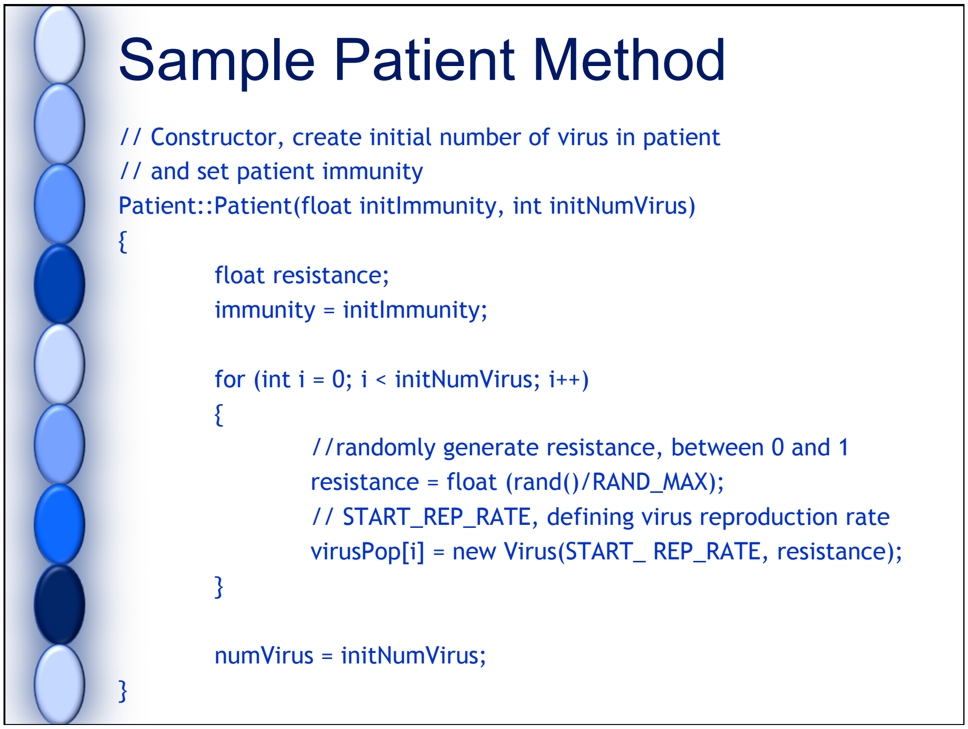
Patient Class

- We have already identified the characteristics and actions for a patient
- Our main program have to interact with the patient object to simulate the growth (or reduction) of virus population over time
  - The main program will have to simulate time passage by interact with the patient object multiple times (though a loop)
  - We will create a method in the patient class that allows for simulating one iteration of time
    - This method is called simulateStep



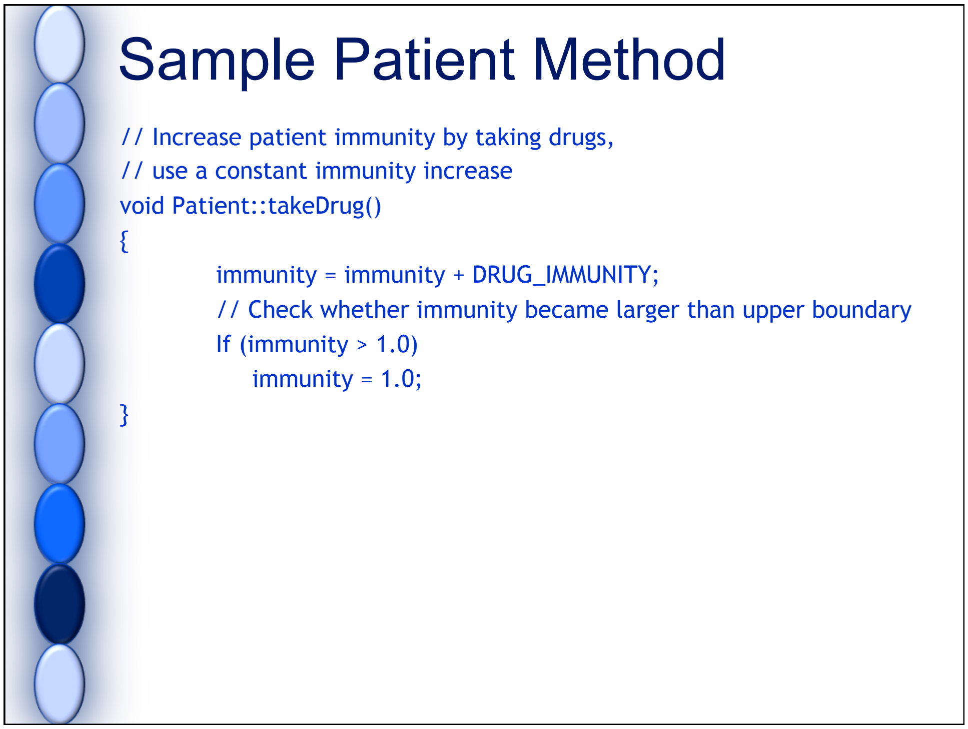
Patient Class

```
#include “Virus.h”                        // Virus class definition
const int MAX_VIRUS_POP = 1000;

class Patient
{
public:
        Patient();
        Patient(float initImmunity, int initNumVirus);
        ~Patient();
        /*** Mutators ***/
        void takeDrug();
        bool simulateStep();

private:
        Virus*     virusPop[MAX_VIRUS_POP];   // virus population in patient
        int        numVirus;                  // number of virus in patient
        float      immunity;                  // degree of immunity, in %
};
```



Sample Patient Method

```
// Constructor, create initial number of virus in patient
// and set patient immunity
Patient::Patient(float initImmunity, int initNumVirus)
{
        float resistance;
        immunity = initImmunity;

        for (int i = 0; i < initNumVirus; i++)
        {
                //randomly generate resistance, between 0 and 1
                resistance = float (rand()/RAND_MAX);
                // START_REP_RATE, defining virus reproduction rate
                virusPop[i] = new Virus(START_ REP_RATE, resistance);
        }

        numVirus = initNumVirus;
}
```



Sample Patient Method

```
// Increase patient immunity by taking drugs,
// use a constant immunity increase
void Patient::takeDrug()
{
        immunity = immunity + DRUG_IMMUNITY;
        // Check whether immunity became larger than upper boundary
        If (immunity > 1.0)
            immunity = 1.0;
}
```