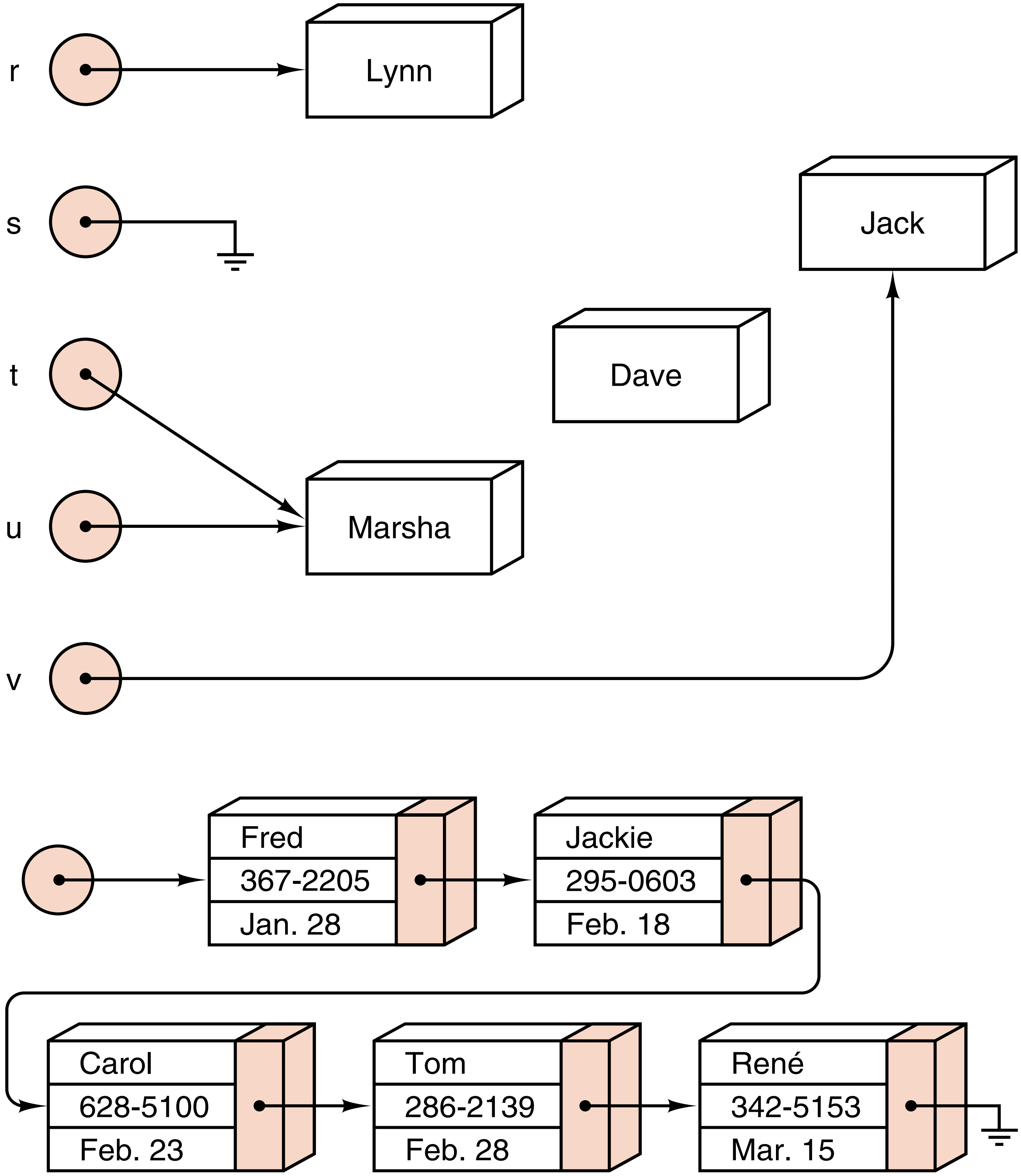
Chapter 4

LINKED STACKS AND QUEUES

1. Pointers and Linked Structures
2. Linked Stacks
3. Linked Stacks with Safeguards
4. Linked Queues
5. Application: Polynomial Arithmetic
6. Abstract Data Types and Implementations

Pointers and a Linked List



r

s

t

u

v

Lynn

Jack

Dave

Marsha

Fred
367-2205
Jan. 28

Jackie
295-0603
Feb. 18

Carol
628-5100
Feb. 23

Tom
286-2139
Feb. 28

René
342-5153
Mar. 15

Key Terms

Overflow:

Running out of space.

Pointer:

An object, often a variable, that stores the location (that is the machine address) of some other object, typically of a structure containing data that we wish to manipulate. (Also sometimes called a ***link*** or a ***reference***)

Linked list:

A list in which each entry contains a pointer giving the location of the next entry.

Contiguous:

Next to each other, touching, adjoining; used in contrast to *linked*.

Automatic object:

An object that exists as long as the block of program declaring it is active; referenced by giving it a name when writing the program.

Dynamic object:

An object that is created (and perhaps destroyed) while the program is running; accessed indirectly via pointers.

Pointers in C++

Notation:

C++ uses an asterisk * to denote a pointer. If Item is a type, then a pointer to such an Item object has the type Item *. For example,

```
Item *item_ptr;
```

declares item_ptr as a pointer variable to an Item object.

Creating dynamic objects:

item_ptr = **new** Item;   creates a new dynamic object of type Item and assigns its location to the pointer variable item_ptr.

The dynamic objects that we create are kept in an area of computer memory called the ***free store*** (or the *heap*).

Deleting dynamic objects:

delete item_ptr;   disposes of the dynamic object to which item_ptr points and returns the space it occupies to the free store so it can be used again.

After this **delete** statement is executed, the pointer variable item_ptr is undefined and so should not be used until it is assigned a new value.

Following pointers:

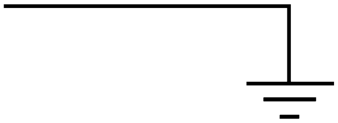
*item_ptr denotes the object to which item_ptr points. The action of taking *item_ptr is called "dereferencing the pointer *item_ptr."

NULL pointers:

If a pointer variable item_ptr has no dynamic object to which it currently refers, then it should be given the special value

```
item_ptr = NULL;
```

In diagrams we reserve the electrical ground symbol



for NULL pointers.

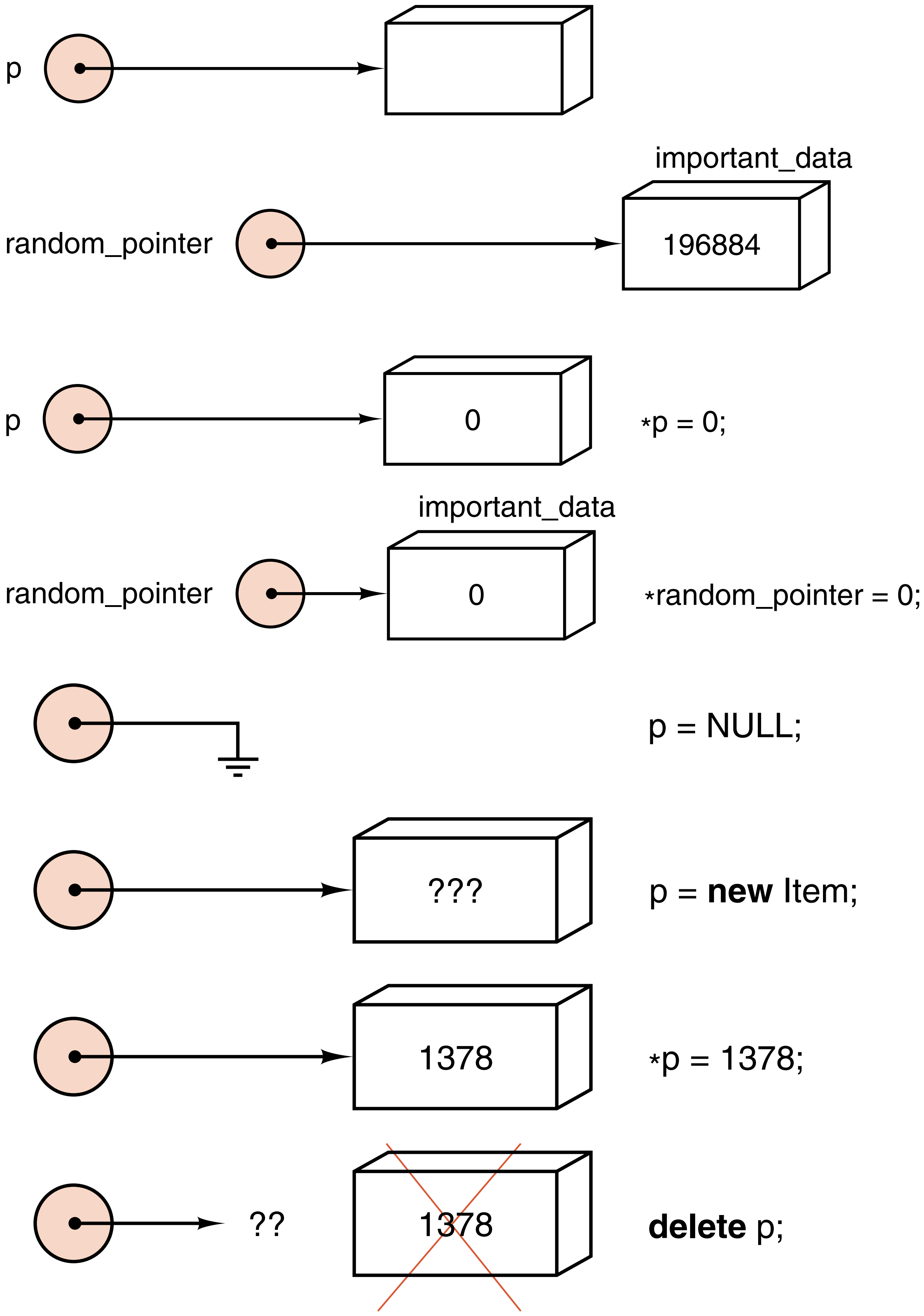
The value NULL is used as a constant for all pointer types and is generic in that the same value can be assigned to a variable of any pointer type.

Undefined pointers versus NULL pointers:

item_ptr == NULL means that item_ptr currently points to no dynamic object. If the value of item_ptr is undefined, then item_ptr might point to any random location in memory.

Programming Precept

Uninitialized or random pointer objects should always be reset to NULL.
After deletion, a pointer object should be reset to NULL.



*p = 0;

*random_pointer = 0;

p = NULL;

p = **new** Item;

*p = 1378;

delete p;

Dynamically Allocated Arrays

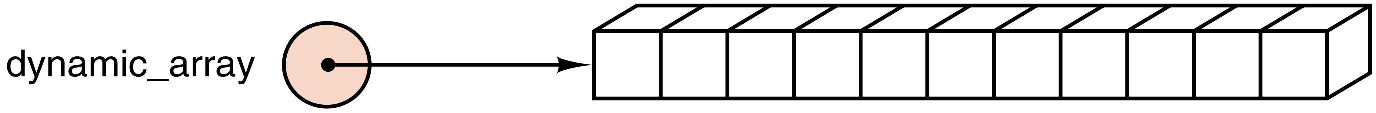
The declaration item_array = **new** Item [array_size]; creates a dynamic array of Item objects, indexed from 0 up to array_size − 1.

Consider, for example:

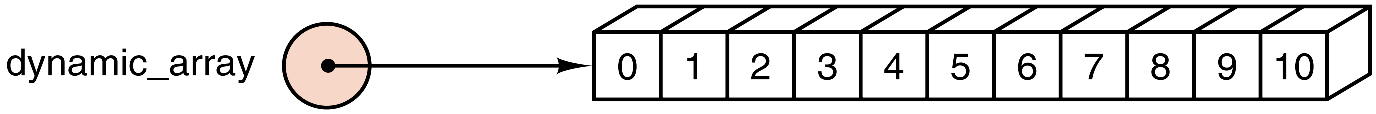
```
int size, *dynamic_array, i;
cout ≪ "Enter an array size: " ≪ flush;
cin ≫ size;
dynamic_array = new int [size];
for (i = 0; i < size; i++) dynamic_array [i] = i;
```

The result is illustrated as:

dynamic_array = **new int** [size];

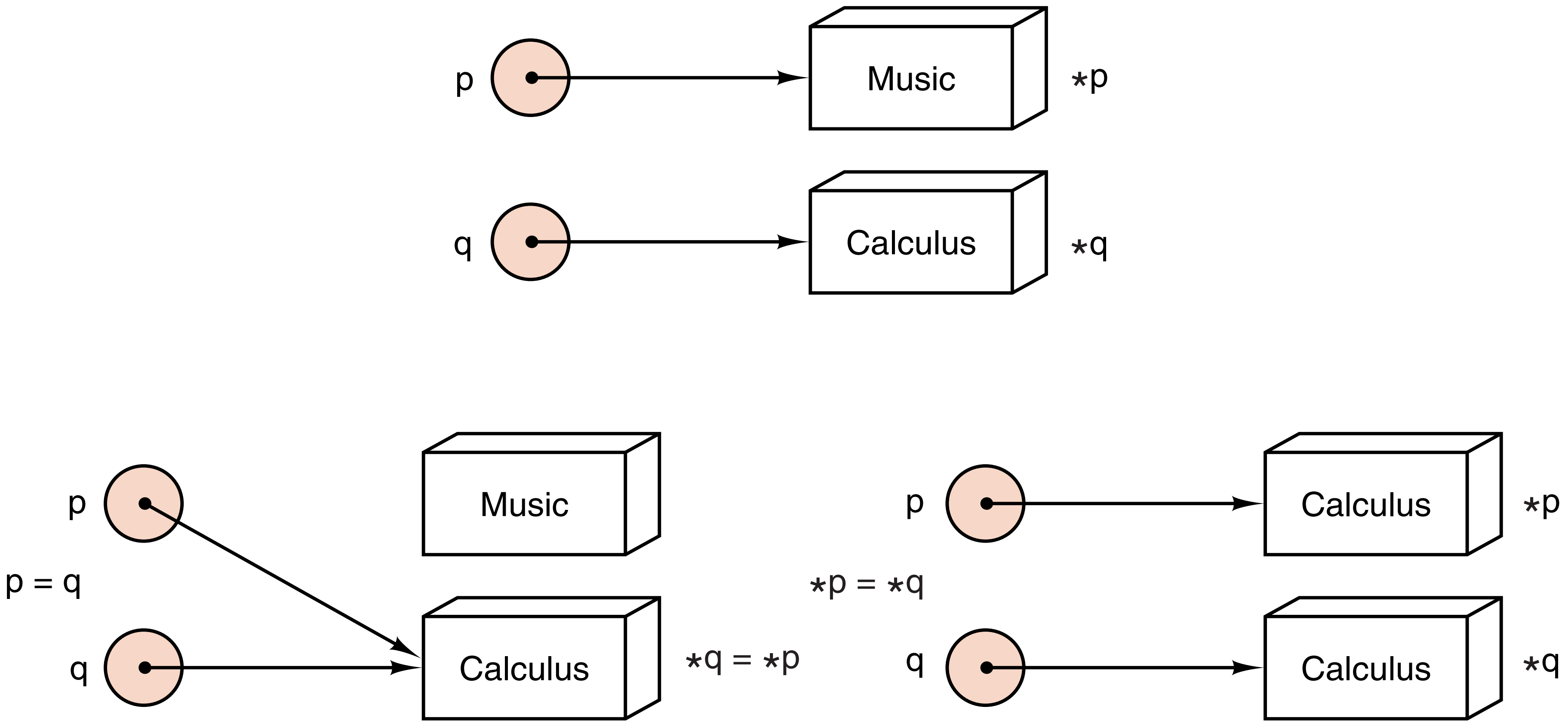


for (i=0; i<size; i++) dynamic_array[i] = i;



The statement **delete** [ ]dynamic_array; returns the storage in dynamic_array to the free store.

If i is an integer value and p is a pointer to an Item, then p + i is an expression of type Item *. The value of p + i gives the memory address offset from p by i Item objects. That is, the expression p + i actually yields the address p + n × i, where n is the number of bytes of storage occupied by a simple object of type Item.


p
Music
*p
q
Calculus
*q
p
Music
p = q
q
Calculus
*q = *p
p
Calculus
*p
*p = *q
q
Calculus
*q

Addresses of automatic objects:

If x is a variable of type Item, then &x is a value of type Item * that gives the address of x. In this case, a declaration and assignment such as Item *ptr = &x would establish a pointer, ptr, to the object x.

Address of an array:

The address of the initial element of an array is found by using the array's name without any attached [ ] operators. For example, given a declaration Item x[20] the assignment

Item *ptr = x

sets up a pointer ptr to the initial element of the array x.

Observe that an assignment expression ptr = &(x[0]) could also be used to find this address.

Pointers to structures:

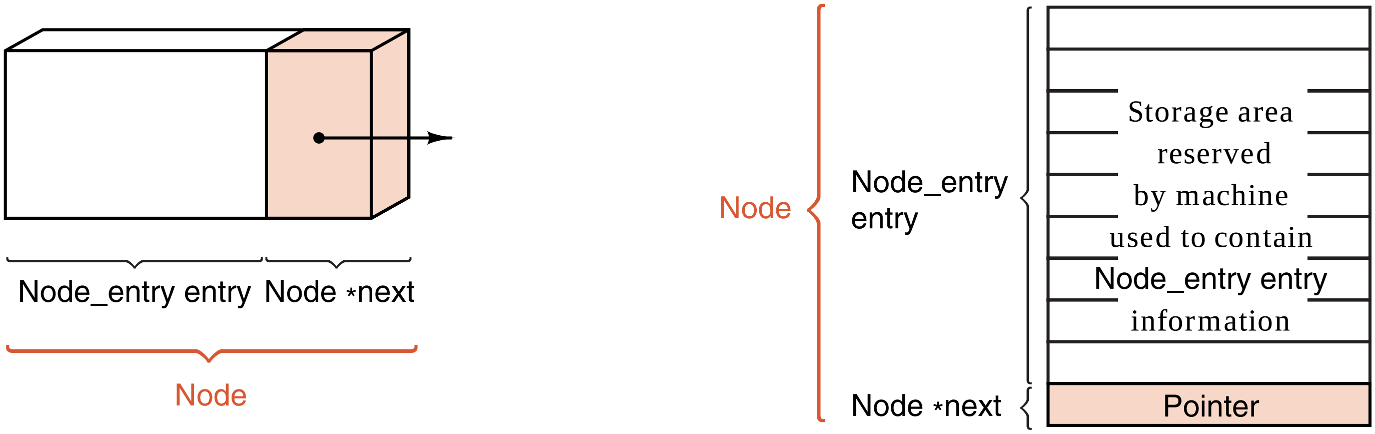
If p is a pointer to a structure object that has a data member called the_data, then we could access this data member with the expression (*p).the_data, but C++ provides the operator -> as a shorthand, so we can replace the expression (*p).the_data by the equivalent, but more convenient, expression p->the_data.

The Basics of Linked Structures

A linked structure is made up of nodes, each containing both the information that is to be stored as an entry of the structure and a pointer telling where to find the next node in the structure. We shall refer to these nodes making up a linked structure as the ***nodes*** of the structure, and the pointers we often call ***links***. Since the link in each node tells where to find the next node of the structure, we shall use the name next to designate this link.

We shall use a **struct** rather than a class to implement nodes.

```
struct Node {
//    data members
   Node_entry entry;
   Node *next;
//    constructors
   Node( );
   Node(Node_entry item, Node *add_on = NULL);
};
```


Node_entry entry
Node *next
Node
Node
Node_entry entry
Storage area reserved by machine used to contain Node_entry entry information
Node *next
Pointer

(a) Structure of a Node   (b) Machine storage representation of a Node

Node Constructors

First form:

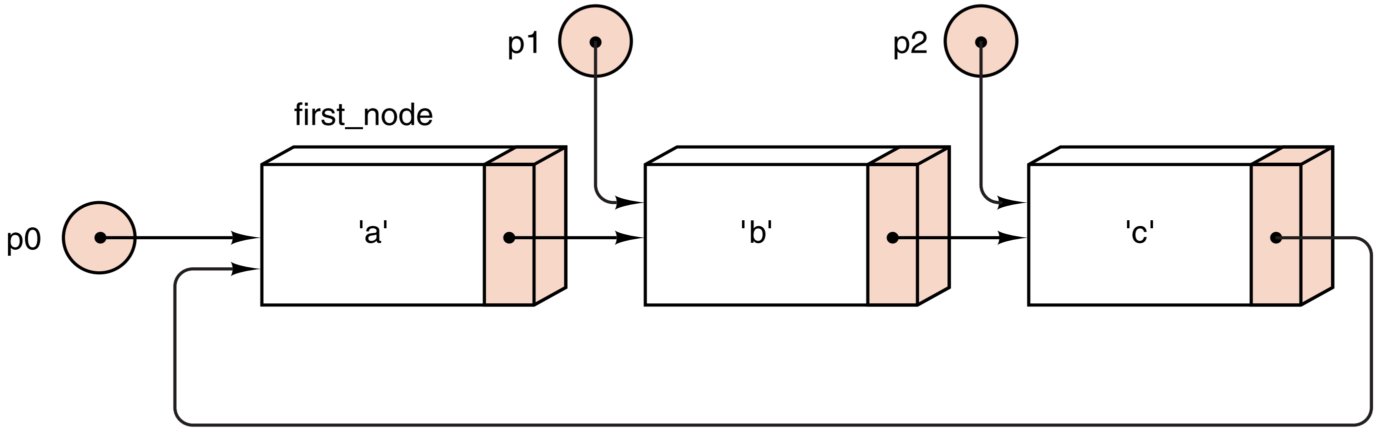
```
Node :: Node( )
{
   next = NULL;
}
```

The second form accepts two parameters for initializing the data members.

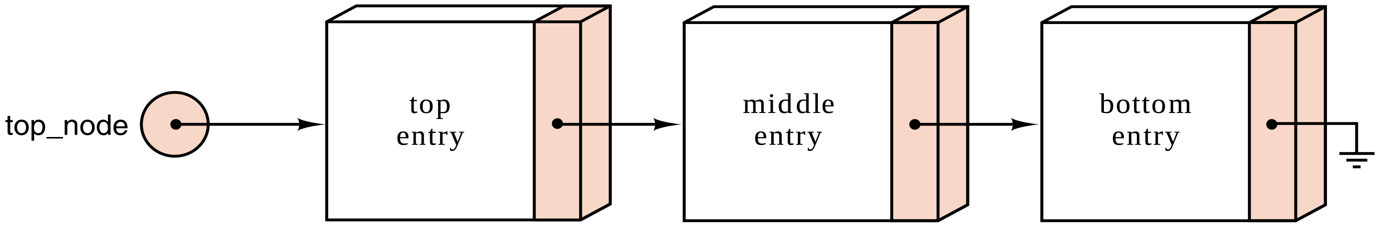
```
Node :: Node(Node_entry item, Node *add_on)
{
   entry = item;
   next = add_on;
}
```

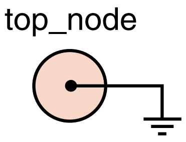
Example:

```
   Node first_node('a');      //   Node first_node stores data 'a'.
   Node *p0 = &first_node;    //   p0 points to first_Node.
   Node *p1 = new Node('b');  //   A second node storing 'b' is created.
   p0->next = p1;             //   The second Node is linked after first_node.
   Node *p2 = new Node('c', p0);  //   A third Node storing 'c' is created.
      //   The third Node links back to the first node, *p0.
   p1->next = p2;             //   The third Node is linked after the second Node.
```

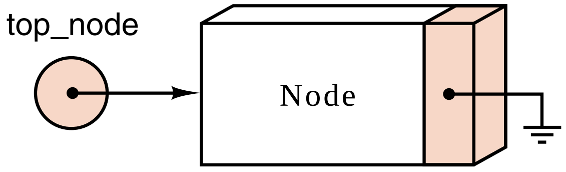


Linked Stacks

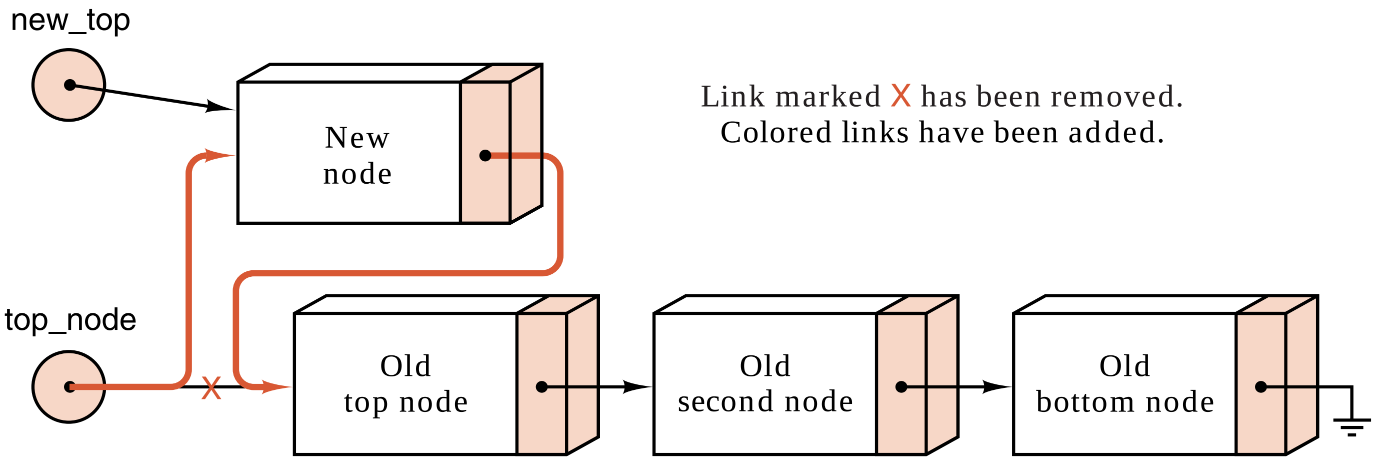




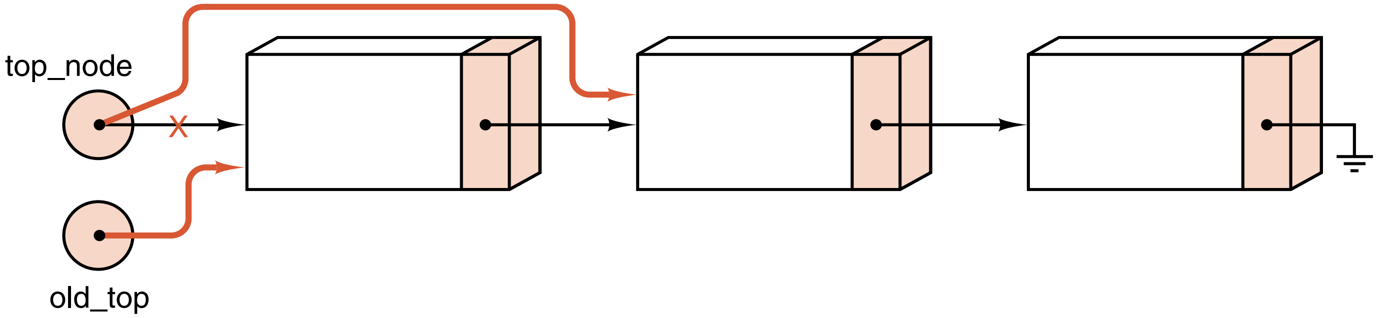
Empty stack



Stack of size 1



Link marked X has been removed.
Colored links have been added.



Class Declaration for Linked Stack

```
class Stack {
public:
   Stack( );
   bool empty( ) const;
   Error_code push(const Stack_entry &item);
   Error_code pop( );
   Error_code top(Stack_entry &item) const;
protected:
   Node *top_node;
};
```

Benefits of Class Implementation

- Maintain encapsulation: If we do not use a class to contain our stack, we lose the ability to set up methods for the stack.
- Maintain the logical distinction between the stack itself, made up of all of its entries (each in a node), and the top of the stack, which is a pointer to a single node.
- Maintain consistency with other data structures and other implementations, where structures are needed to collect several methods and pieces of information.
- Help with debugging by allowing the compiler to perform better type checking.

Pushing a Linked Stack

```
Error_code Stack :: push(const Stack_entry &item)
/* Post: Stack_entry item is added to the top of the Stack; returns success or
         returns a code of overflow if dynamic memory is exhausted. */

{
   Node *new_top = new Node(item, top_node);
   if (new_top == NULL) return overflow;
   top_node = new_top;
   return success;
}
```

Popping a Linked Stack

```
Error_code Stack :: pop( )
/* Post: The top of the Stack is removed. If the Stack is empty the method returns
         underflow; otherwise it returns success. */

{
   Node *old_top = top_node;
   if (top_node == NULL) return underflow;
   top_node = old_top->next;
   delete old_top;
   return success;
}
```

Linked Stacks with Safeguards

Client code can apply the methods of linked stacks in ways that lead to the accumulation of garbage or that break the encapsulation of Stack objects.

C++ provides three devices (additional **class** methods) to alleviate these problems:

- destructors,
- copy constructors
- overloaded assignment operators

These new methods replace compiler generated default behavior and are often called silently (that is, without explicit action by a client).

Problem Example

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

Suppose that the linked Stack implementation is used. As soon as the object small goes out of scope, the data stored in small becomes garbage. Over the course of a million iterations of the loop, a lot of garbage will accumulate. The loop would have executed without any problem with a contiguous Stack implementation, where all allocated space for member data is released every time a Stack object goes out of scope.

The Destructor

Definition:

A ***destructor*** is a special method in a class that is automatically executed on objects of the class immediately before they go out of scope.

The client does not need to call a destructor explicitly and does not even need to know it is present. Destructors are often used to delete dynamically allocated objects that would otherwise become garbage.

Declaration:

The destructor must be declared as a class method without return type and without parameters. Its name is given by adding a ∼ prefix to the corresponding class name. Hence, the prototype for a Stack destructor is:

```
Stack :: ∼Stack( );
```

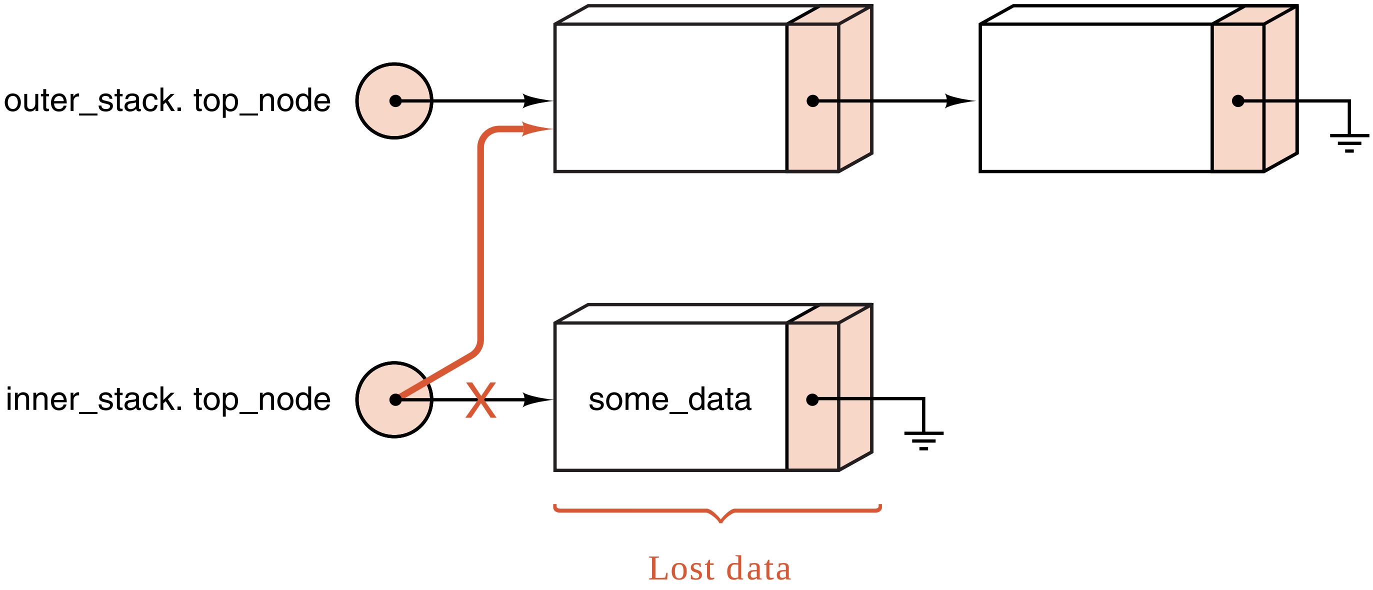
```
Stack :: ∼Stack( )            //    Destructor
/* Post: The Stack is cleared. */
{
   while (!empty( ))
      pop( );
}
```

Policy

Every linked structure should be equipped with a destructor to clear its objects before they go out of scope.

Dangers in Assignment

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



Misbehaviors:

- Lost data space.
- Two stacks have shared nodes.
- The destructor on inner_stack deletes outer_stack.
- Such a deletion leaves the pointer outer_stack.top_node addressing what a random memory location.

Overloading the Assignment Operator

In C++, we implement special methods, known as ***overloaded assignment operators*** to redefine the effect of assignment. Whenever the C++ compiler translates an assignment expression of the form x = y, it first checks whether the class of x has an overloaded assignment operator.

Prototype:

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

This declares a Stack method called **operator** = , the overloaded assignment operator, that can be invoked with

x.**operator** = (y);   or   x = y;

By looking at the type(s) of its operands, the C++ compiler can tell that it should use the overloaded operator rather than the usual assignment.

Implementation outline:

- Make a copy of the data stacked in the calling parameter.
- Clear out any data already in the Stack object being assigned to.
- Move the newly copied data to the Stack object.

Overloaded Assignment of Linked Stacks

```
void Stack :: operator = (const Stack &original) //    Overload assignment
/* Post: The Stack is reset as a copy of Stack original. */
{
   Node *new_top, *new_copy, *original_node = original.top_node;
   if (original_node == NULL) new_top = NULL;
   else {                                   //    Duplicate the linked nodes
      new_copy = new_top = new Node(original_node->entry);
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node(original_node->entry);
         new_copy = new_copy->next;
      }
   }
   while (!empty())                         //    Clean out old Stack entries
      pop();
   top_node = new_top;                      //    and replace them with new entries.
}
```

The Copy Constructor

Problem example:

```
void destroy_the_stack (Stack copy)
{
}

int main( )
{
  Stack vital_data;
  destroy_the_stack(vital_data);
}
```

In this code, a copy of the Stack vital_data is passed to the function. The Stack copy shares its nodes with the Stack vital_data, and therefore when a Stack destructor is applied to copy, at the end of the function, vital_data is also destroyed.

Solution:

If we include a *copy constructor* as a member of our Stack class, our copy constructor will be invoked whenever the compiler needs to copy Stack objects. We can thus ensure that Stack objects are copied using value semantics.

For any class, a standard way to declare a ***copy constructor*** is as a constructor with one argument that is declared as a constant reference to an object of the class.

Hence, a Stack copy constructor would normally have the following prototype:

```
Stack :: Stack(const Stack &original);
```

Implementation outline:

1. Deal with the case of copying an empty Stack.
2. Copy the first node.
3. Run a loop to copy all of the other nodes.

Linked-stack copy constructor:

```
Stack :: Stack(const Stack &original)   //   copy constructor
/* Post: The Stack is initialized as a copy of Stack original. */
{
   Node *new_copy, *original_node = original.top_node;
   if (original_node == NULL) top_node = NULL;
   else {                              //   Duplicate the linked nodes.
      top_node = new_copy = new Node(original_node->entry);
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node(original_node->entry);
         new_copy = new_copy->next;
      }
   }
}
```

Policy

For every linked class, include a copy constructor, or
warn clients that objects are copied with reference semantics.

Modified Linked-Stack Specification

```
class Stack {
public:
//    Standard Stack methods
   Stack( );
   bool empty( ) const;
   Error_code push(const Stack_entry &item);
   Error_code pop( );
   Error_code top(Stack_entry &item) const;
//    Safety features for linked structures
    ~Stack( );
   Stack(const Stack &original);
   void operator = (const Stack &original);
protected:
   Node *top_node;
};
```

Linked Queues

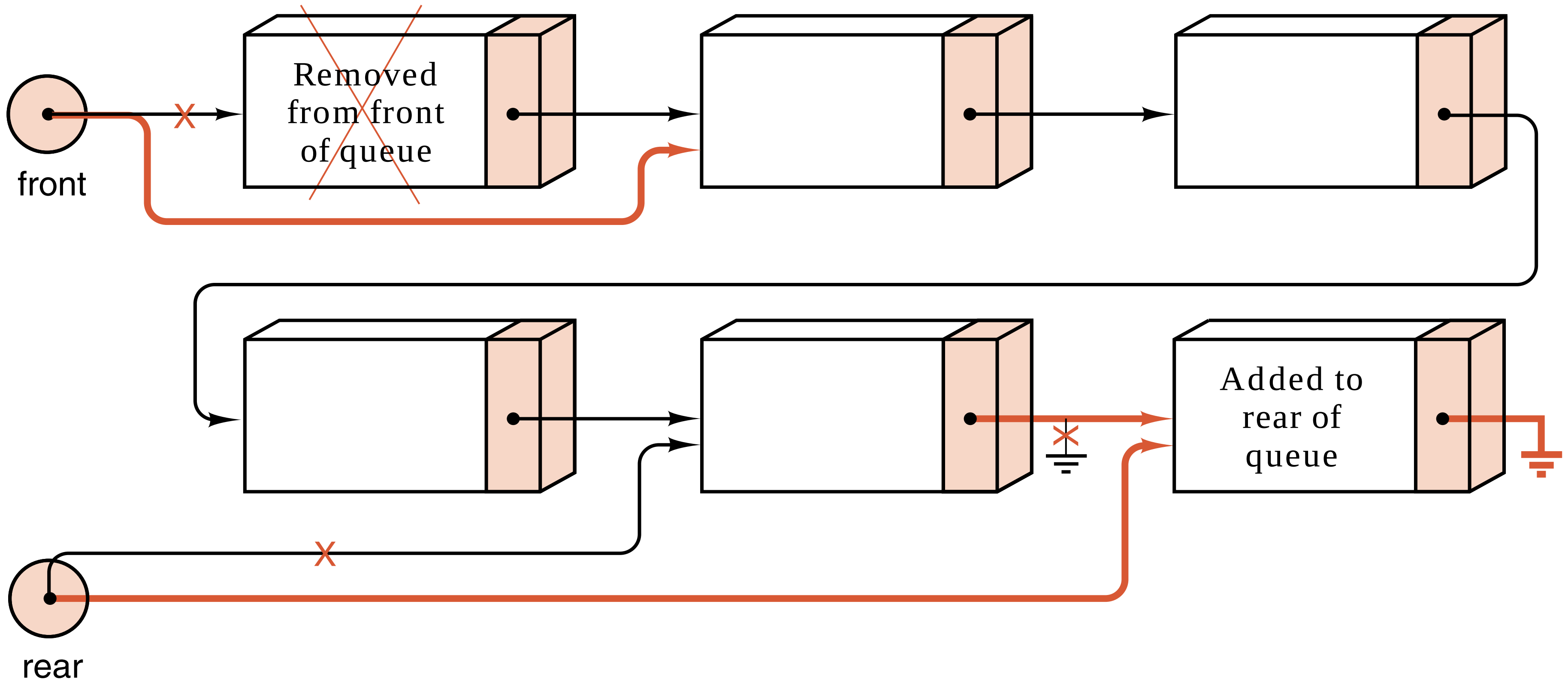
Class declaration, linked queues:

```
class Queue {
public:
//    standard Queue methods
   Queue( );
   bool empty( ) const;
   Error_code append(const Queue_entry &item);
   Error_code serve( );
   Error_code retrieve(Queue_entry &item) const;
//    safety features for linked structures
   ~Queue( );
   Queue(const Queue &original);
   void operator = (const Queue &original);
protected:
   Node *front, *rear;
};
```

Constructor:

```
Queue :: Queue( )
/* Post:  The Queue is initialized to be empty.  */
{
   front = rear = NULL;
}
```

Operations on a linked queue



Linked Queue Methods

Append an entry:

```
Error_code Queue :: append(const Queue_entry &item)
/* Post: Add item to the rear of the Queue and return a code of success or
         return a code of overflow if dynamic memory is exhausted. */
{
   Node *new_rear = new Node(item);
   if (new_rear == NULL) return overflow;
   if (rear == NULL) front = rear = new_rear;
   else {
      rear->next = new_rear;
      rear = new_rear;
   }
   return success;
}
```

Serve an entry:

```
Error_code Queue :: serve( )
/* Post: The front of the Queue is removed. If the Queue is empty, return an
         Error_code of underflow. */
{
   if (front == NULL) return underflow;
   Node *old_front = front;
   front = old_front->next;
   if (front == NULL) rear = NULL;
   delete old_front;
   return success;
}
```

Extended Linked Queues

Class definition:

```
class Extended_queue: public Queue {
public:
   bool full( ) const;
   int size( ) const;
   void clear( );
   Error_code serve_and_retrieve(Queue_entry &item);
};
```

There is no need to supply explicit methods for the copy constructor, the overloaded assignment operator, or the destructor, since the compiler calls the corresponding method of the base Queue object.

Size:

```
int Extended_queue :: size( ) const
/* Post: Return the number of entries in the Extended_queue. */
{
   Node *window = front;
   int count = 0;
   while (window != NULL) {
      window = window->next;
      count++;
   }
   return count;
}
```

Application: Polynomial Arithmetic

- We develop a program that simulates a calculator that does addition, subtraction, multiplication, division, and other operations for polynomials rather than numbers.
- We model a *reverse Polish* calculator whose operands (polynomials) are entered *before* the operation is specified. The operands are pushed onto a stack. When an operation is performed, it pops its operands from the stack and pushes its result back onto the stack.
- We reuse the conventions of Section 2.3: ? denotes pushing an operand onto the stack, + , −, * , / represent arithmetic operations, and = means printing the top of the stack (but not popping it off).

The Main Program

```
int main( )
/* Post: The program has executed simple polynomial arithmetic commands entered by
         the user.
   Uses: The classes Stack and Polynomial and the functions introduction, in-
         structions, do_command, and get_command. */

{
   Stack stored_polynomials;
   introduction( );
   instructions( );
   while (do_command(get_command( ), stored_polynomials));
}
```

Performing Commands

```
bool do_command(char command, Stack &stored_polynomials)
/* Pre:  The first parameter specifies a valid calculator command.
   Post: The command specified by the first parameter has been applied to the Stack
         of Polynomial objects given by the second parameter. A result of true is
         returned unless command == 'q'.
   Uses: The classes Stack and Polynomial. */

{
   Polynomial p, q, r;

   switch (command) {
   case '?':
      p.read( );
      if (stored_polynomials.push(p) == overflow)
         cout << "Warning: Stack full, lost polynomial" << endl;
      break;

   case '=':
      if (stored_polynomials.empty( ))
         cout << "Stack empty" << endl;
      else {
         stored_polynomials.top(p);
         p.print( );
      }
      break;
```

```
  case ′+′:
     if (stored_polynomials.empty( ))
        cout ≪ "Stack empty" ≪ endl;
     else {
        stored_polynomials.top(p);
        stored_polynomials.pop( );
        if (stored_polynomials.empty( )) {
           cout ≪ "Stack has just one polynomial" ≪ endl;
           stored_polynomials.push(p);
        }

        else {
           stored_polynomials.top(q);
           stored_polynomials.pop( );
           r.equals_sum(q, p);
           if (stored_polynomials.push(r) ==  overflow)
              cout ≪ "Warning: Stack full, lost polynomial" ≪ endl;
        }
     }
     break;

  //    Add options for further user commands.

  case ′q′:
     cout ≪ "Calculation finished." ≪ endl;
     return false;
  }
  return true;
}
```

Stubs and Testing

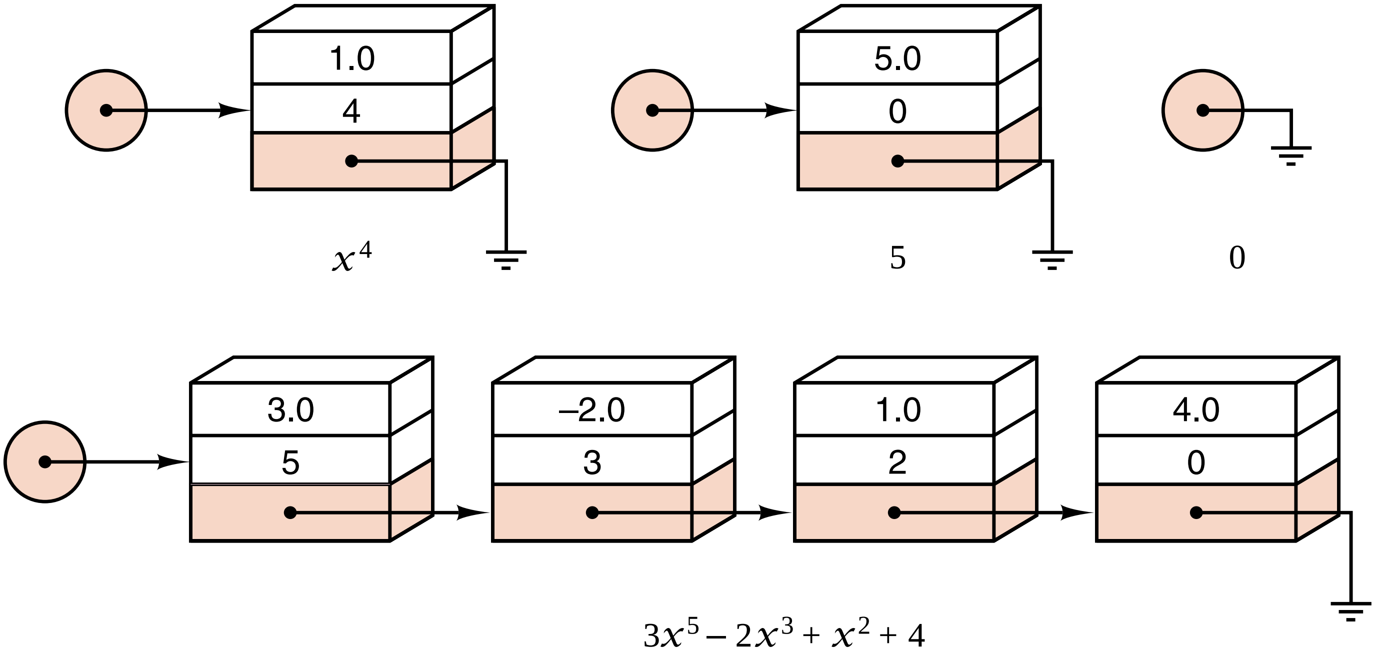
- Let us pause to compile the program, debug it, and test it to make sure that what has been done so far is correct.
- To compile the program, we must supply stubs for all the missing elements. The only missing part is the class Polynomial.
- We have not yet even decided how to store polynomial objects.
- For testing, we run the program as an ordinary reverse Polish calculator operating on real numbers.
- We need a stub class declaration that uses real numbers in place of polynomials:

```
class Polynomial {
public:
   void read( );
   void print( );
   void equals_sum(Polynomial p, Polynomial q);
   void equals_difference(Polynomial p, Polynomial q);
   void equals_product(Polynomial p, Polynomial q);
   Error_code equals_quotient(Polynomial p, Polynomial q);
private:
   double value;
};

void Polynomial :: equals_sum(Polynomial p, Polynomial q)
{
   value = p.value + q.value;
}
```

- Producing a skeleton program ensures that the stack and utility packages are properly integrated into the program.

Data Structures for Polynomials



1. Each node represents one term of a polynomial and is a structure containing a coefficient, an exponent, and a pointer to the next term of the polynomial.

2. The terms of every polynomial are stored in the order of decreasing exponent within the linked queue, and no two terms have the same exponent.

3. Terms with zero coefficient are not stored in the polynomial.

4. The polynomial that is identically 0 is represented by an empty queue.

Writing a Polynomial

The following print method reflects the customary but quite special conventions for writing polynomials. Our method suppresses any initial + sign, any coefficients and exponents with value 1, and any reference to x^0.

```
void Polynomial :: print( ) const
/* Post: The Polynomial is printed to cout. */
{
   Node *print_node = front;
   bool first_term = true;

   while (print_node != NULL) {
      Term &print_term = print_node->entry;
      if (first_term) {                    //   In this case, suppress printing an initial '+'.
         first_term = false;
         if (print_term.coefficient < 0) cout ≪ "− ";
      }

      else if (print_term.coefficient < 0) cout ≪ " − ";
      else cout ≪ " + ";

      double r = (print_term.coefficient >= 0)
                    ? print_term.coefficient : −(print_term.coefficient);
      if (r != 1) cout ≪ r;
      if (print_term.degree > 1) cout ≪ " Xˆ" ≪ print_term.degree;
      if (print_term.degree == 1) cout ≪ " X";
      if (r == 1 && print_term.degree == 0) cout ≪ " 1";
      print_node = print_node->next;
   }
   if (first_term)
      cout ≪ "0";                          //   Print 0 for an empty Polynomial.
   cout ≪ endl;
}
```

Reading a Polynomial

```
void Polynomial :: read( )
/* Post: The Polynomial is read from cin. */
{ clear( );
  double coefficient;
  int last_exponent, exponent;
  bool first_term = true;
  cout ≪ "Enter the coefficients and exponents for the polynomial, "
       ≪ "one pair per line." ≪ endl
       ≪ "Exponents must be in descending order." ≪ endl
       ≪ "Enter a coefficient of 0 or an exponent of 0 to terminate."
       ≪ endl;
  do {
    cout ≪ "coefficient? " ≪ flush;
    cin  ≫ coefficient;
    if (coefficient != 0.0) {
      cout ≪ "exponent? " ≪ flush;
      cin  ≫ exponent;
      if ((!first_term && exponent >= last_exponent) || exponent < 0) {
        exponent = 0;
        cout ≪ "Bad exponent:"
             ≪ "Polynomial terminates without its last term." ≪ endl;
      }
      else {
        Term new_term(exponent, coefficient);
        append(new_term);
        first_term = false;
      }
      last_exponent = exponent;
    }
  } while (coefficient != 0.0 && exponent != 0);
}
```

Addition of Polynomials

```
void Polynomial :: equals_sum(Polynomial p, Polynomial q)
/* Post: The Polynomial object is reset as the sum of the two parameters. */

{
   clear( );
   while (!p.empty( ) || !q.empty( )) {
      Term p_term, q_term;
      if (p.degree( ) > q.degree( )) {
         p.serve_and_retrieve(p_term);
         append(p_term);
      }

      else if (q.degree( ) > p.degree( )) {
         q.serve_and_retrieve(q_term);
         append(q_term);
      }

      else {
         p.serve_and_retrieve(p_term);
         q.serve_and_retrieve(q_term);
         if (p_term.coefficient + q_term.coefficient != 0) {
            Term answer_term(p_term.degree,
                                  p_term.coefficient + q_term.coefficient);
            append(answer_term);
         }
      }
   }
}
```

Group Project Responsibilities

1. Allocation of tasks

2. Determining capabilities and specifications

3. Timetable

4. Stubs, drivers, and testing

5. Modifications, extensions, and revisions

6. Coordination and supervision

7. Documentation and reporting

Abstract Queues

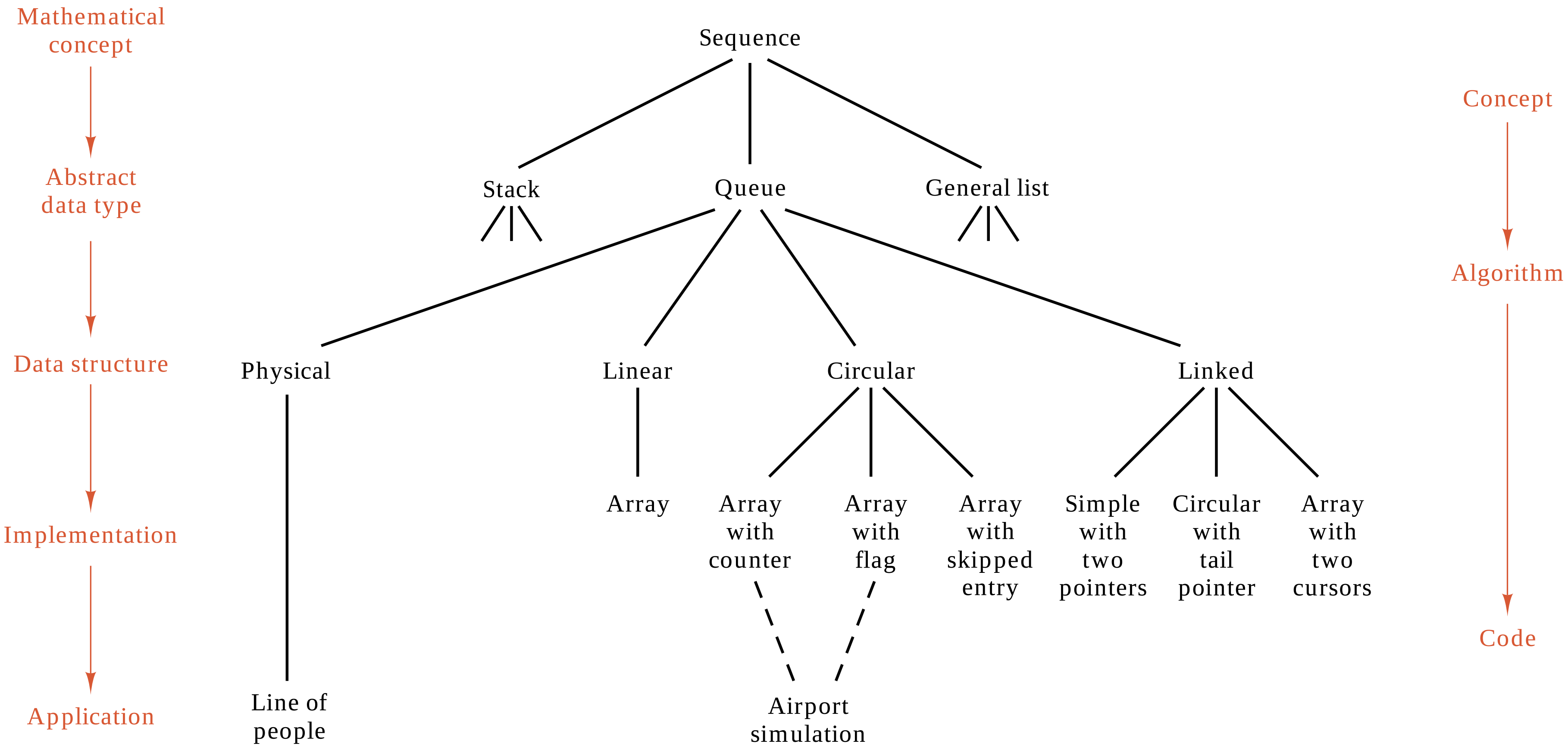
Definition A ***queue*** of elements of type T is a finite sequence of elements of T together with the following operations:

1. *Create* the queue, leaving it empty.
2. Test whether the queue is *Empty*.
3. *Append* a new entry onto the rear of the queue, provided the queue is not full.
4. *Serve* (and remove) the entry from the front of the queue, provided the queue is not empty.
5. *Retrieve* the front entry off the queue, provided the queue is not empty.

Definition An ***extended queue*** of elements of type T is a queue of elements of T together with the following additional operations:

4. Determine whether the queue is *full* or not.
5. Find the *size* of the queue.
6. *Serve and retrieve* the front entry in the queue, provided the queue is not empty.
7. *Clear* the queue to make it empty.

Refinement of a queue



Mathematical concept → Abstract data type → Data structure → Implementation → Application

Sequence

Stack

Queue

General list

Physical

Linear

Circular

Linked

Array

Array with counter

Array with flag

Array with skipped entry

Simple with two pointers

Circular with tail pointer

Array with two cursors

Line of people

Airport simulation

Concept → Algorithm → Code

Pointers and Pitfalls

1. Before choosing implementations, be sure that all the data structures and their associated operations are fully specified on the abstract level.
2. To help in the choice between linked and contiguous implementations, consider the necessary operations on the data structure. Linked structures are more flexible in regard to insertions, deletions, and rearrangement; contiguous structures are sometimes faster.
3. Contiguous structures usually require less computer memory, computer time, and programming effort when the items in the structure are small and the algorithms are simple. When the structure holds large records, linked structures usually save space, time, and often programming effort.
4. Dynamic memory and pointers allow a program to adapt automatically to a wide range of application sizes and provide flexibility in space allocation among different data structures. Automatic memory is sometimes more efficient for applications whose size can be completely specified in advance.
5. Before reassigning a pointer, make sure that the object that it references will not become garbage.
6. Set uninitialized pointers to `NULL`.
7. Linked data structures should be implemented with destructors, copy constructors, and overloaded assignment operators.
8. Use private inheritance to model an "is implemented with" relationship between classes.
9. Draw "before" and "after" diagrams of the appropriate part of a linked structure, showing the relevant pointers and the way in which they should be changed. If they might help, also draw diagrams showing intermediate stages of the process.

10. To determine in what order values should be placed in the pointer fields to carry out the various changes, it is usually better first to assign the values to previously undefined pointers, then to those with value NULL, and finally to the remaining pointers. After one pointer variable has been copied to another, the first is free to be reassigned to its new location.

11. Be sure that no links are left undefined at the conclusion of a method of a linked structure, either as links in new nodes that have never been assigned or links in old nodes that have become dangling, that is, that point to nodes that no longer are used. Such links should either be reassigned to nodes still in use or set to the value NULL.

12. Verify that your algorithm works correctly for an empty structure and for a structure with only one node.

13. Avoid the use of constructions such as (p->next)->next, even though they are syntactically correct. A single object should involve only a single pointer dereferencing. Constructions with repeated dereferencing usually indicate that the algorithms can be improved by rethinking what pointer variables should be declared in the algorithm, introducing new ones if necessary.