VIII. Run-Time Arrays—Intro. to Pointers (§8.4 & 8.5)

A. Introduction to Pointers

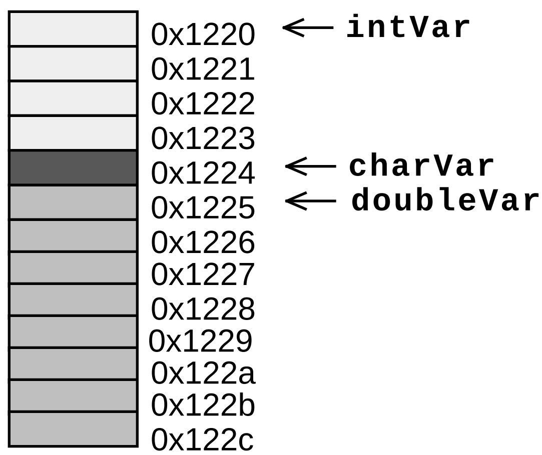
For declarations like

```
double doubleVar;
char charVar = 'A';
int intVar = 1234;
```

the compiler constructs the object being declared (`intVar`, `doubleVar`, and `charVar`), which means that it:

1. Allocates memory needed for values of that type
2. Associates the object's name with that memory
3. Initializes that memory

For example:



1. The Address-of Operator (&)

We have seen (Lab 1) that a variable's address can be determined by using the **address-of operator (&):**

&*variable* is the address of *variable*

Example: For the scenario described above:

Values of `&intVar`, `&charVar`, and `&doubleVar`

↓ ↓ ↓

0x1220, 0x1224, and 0x1225

2. Pointer Variables

a. To make addresses more useful, C++ provides *pointer variables***.**

Definition: A **pointer variable** (or simply **pointer**) is a variable whose value is a memory address.

b. Declarations:

```
Type * pointerVariable
```

declares a variable named `pointerVariable` that can store **the address of an object of type *Type***.

Example:

```
#include <iostream>
using namespace std;

int main()
{
  int i = 11, j = 22;
  double d = 3.3, e = 4.4;
                              // pointer variables that:
  int * iptr, * jptr;         //    store addresses of ints)
  double * dptr, * eptr;      //    store addresses of doubles)

  iptr = &i;                  // value of iptr is address of i
  jptr = &j;                  // value of jptr is address of j

  dptr = &d;                  // value of dptr is address of d
  eptr = &e;                  // value of eptr is address of e

  cout << "&i = " << (void*)iptr << endl
       << "&j = " << (void*)jptr << endl
       << "&d = " << (void*)dptr << endl
       << "&e = " << (void*)eptr << endl;
}
```

Output produced:

```
&i = 0x7fffb7f4
&j = 0x7fffb7f0
&d = 0x7fffb7e8
&e = 0x7fffb7e0
```

3. Dereferencing Operator

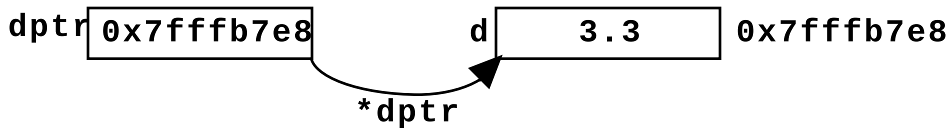
We have also seen that the dereferencing (or indirection) operator `*` can be used to access a value stored in a location. Thus for an expression of the form

```
*pointerVariable
```

the value produced is **not the address** stored in `pointerVariable`, but is instead the **value stored in memory at that address**.

Example:

Value of `dptr`: 0x7fffb7e8
Value of `*dptr`: 3.3



We say `dptr` **points to** that memory location (whose address is 0x7fffb7e8).

Suppose we replace the preceding output statements by:

```
cout << "i = " << *iptr << endl
     << "j = " << *jptr << endl
     << "d = " << *dptr << endl
     << "e = " << *eptr << endl;
```

Output produced will be:

```
i = 11
j = 22
d = 3.3
e = 4.4
```

4. A Note about Reference Parameters

Recall the C++ function to exchange the values of two `int` variables:

```
void Swap(int & A, int & B)
{
  int Temp = A; A = B; B = Temp;
}
```

The values of two `int` variables `x` and `y` can be exchanged with the call:

```
Swap(x,y);
```

The first C++ compilers were just preprocessors that read a C++ program, produced functionally equivalent C code, and ran it through the C compiler. But C has no reference parameters. How were they handled?

Translate the function to

```
void Swap(int * A, int * B)
{
  int Temp = *A; *A = *B; *B = Temp;
}
```

and the preceding call to

```
Swap(&x, &y);
```

This indicates how the call-by-reference parameter mechanism works:

→ A reference parameter is a variable containing the **address of its argument** (i.e., a **pointer variable**) and that is automatically **dereferenced** when used.

6. Anonymous Variables

a. Definition: A **variable** is a memory location.
A **named variable** has a name associated with its memory location, so that this memory location can be accessed conveniently.

An **anonymous variable** has no name associated with its memory location, but if the **address** of that memory location is stored in a **pointer variable**, then the variable can be **accessed indirectly using the pointer**.

b. Named variables are created using a normal variable declaration. For example, in the preceding example, the declaration

```
int j = 22;
```

i. constructed an integer (4-byte) variable at memory address 0x7fffb7f4 and initialized those 4 bytes to the value 22; and

ii. associated the name `j` with that address, so that all subsequent uses of `j` refer to address 0x7fffb7f4; the statement

```
cout << j << endl;
```

will display the 4-byte value (22) at address 0x7fffb7f4.

c. Anonymous variables are created using the **`new` operator**, whose form is:

```
new Type
```

When executed, this expression:

i. allocates a block of memory big enough for an object of type *Type*, and

ii. returns the starting address of that block of memory.

Example:

```
#include <iostream>
using namespace std;

int main()
{
  double * dptr,
         * eptr;

   dptr = new double;
   eptr = new double;

  cout << "Enter two numbers: ";
  cin >> *dptr >> *eptr;
  cout << *dptr << " + " << *eptr
       << " = " << *dptr + *eptr << endl;
}
```

Sample run:

```
Enter two numbers: 2.2 3.3
2.2 + 3.3 = 5.5
```

The program uses the `new` operator to allocate two anonymous variables whose addresses are stored in pointer variables `dptr` and `eptr`:

```
double * dptr, * eptr;

dptr = new double;
eptr = new double;
```

Note 1: We could have performed these allocations as initializations in the declarations of `dptr` and `eptr`:

```
double * dptr = new double,
       * eptr = new double;
```

Note 2: `new` must be used each time a memory allocation is needed. For example, in the assignment

```
dptr = eptr = new double;
```

`eptr = new double` allocates memory for a double value and assigns its address to `eptr`, but `dptr = eptr` simply assigns this same address to `dptr` (and does not allocate new memory.)

The program then inputs two numbers, storing them in these anonymous variables by dereferencing `dptr` and `eptr` in an input statement:

```
cout << "Enter two numbers: ";
cin >> *dptr >> *eptr;
```

It then outputs the two numbers and their sum:

```
cout << *dptr << " + " << *eptr
     << " = " << *dptr + *eptr << endl;
```

by dereferencing the pointer variables.

The expression `*dptr + *eptr` computes the sum of these anonymous variables. If we had wished to store this sum in a third anonymous variable, we could have written:

```
double * fptr = new double;

*fptr = *dptr + *eptr;
cout << *fptr << endl;
```

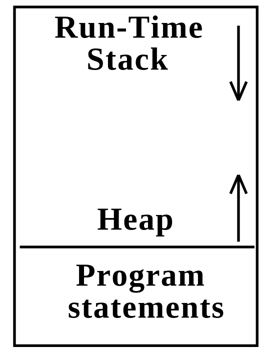
Note: It is an error to attempt to allocate the wrong type of memory block to a pointer variable; for example,

```
double dptr = new int;    // error
```

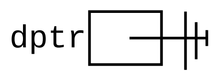
produces a compiler error.

7. Memory Allocation/Deallocation

`new` receives its memory allocation from a pool of available memory (called the *heap* or *free store*). It is usually located between a program and its run-time stack: The run-time stack grows each time a function is called, so it is possible for it to overrun the heap (if `main()` calls a function that calls a function that calls a function ...) It is also possible for the heap to overrun the run-time stack (if a program performs lots of `new` operations).



If a program executes a `new` operation and the heap has been exhausted, then **`new` returns the value 0** (called the **null address** or **null pointer**). It is common to picture a null pointer variable using the electrical engineering ground symbol:



It is a good idea to check whether a pointer variable has a null value before attempting to dereference it because *an attempt to dereference a null (or uninitialized or void) pointer variable produces a* ***segmentation fault***

```
double *dptr = new double;
if (dptr == 0)
{
  cerr << "\n*** No more memory!\n";
  exit(-1);
}
```

When many such checks must be made, an assertion is probably more convenient:

```
assert (dptr != 0);
```

The RTS grows each time a function is called, but it shrinks again when that function terminates. What is needed is an analogous method to reclaim memory allocated by `new`, to shrink the heap when an anonymous variable is no longer needed. Otherwise a **memory leak** results.

For this, C++ provides the **`delete` operation**:

```
delete pointerVariable
```

which **deallocates** the block of memory whose address is stored in `pointerVariable`, when it is no longer needed.

B. Run-Time-Allocated Arrays

Container classes like `Stack` and `Queue` that use arrays (as we know them) to store the elements have one obvious deficiency:

Their capacities are fixed at compile time.

This is because arrays as we have used them up to now have their capacities fixed at compile time. For example, the declaration

```
double a[50];
```

declares an array with exactly 50 elements.

This kind of array is adequate if a fixed-capacity array can be used to store all of the data sets being processed. However, this often is not true because the sizes of the data sets vary. In this case we must either:

— Make the array's capacity large enough to handle the biggest data set — an obvious waste of memory for smaller data sets.
— Change the capacity in the array's declaration in the source program/library and recompile.

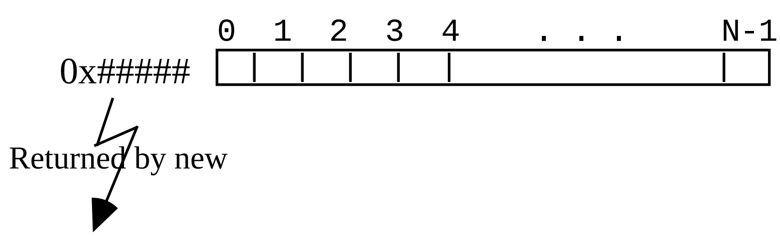
It would be nice if the user could specify the capacity of the array/stack/queue at **run time** and an array of that capacity would then be allocated and used. This is possible in C++.

1. Allocating an Array During Run-Time

The operator **`new`** can be used in an expression of the form

```
new Type[N]
```

where *`N`* is an integer expression, to allocate an **array with *N* elements, each of type *Type*;** it returns the **base address** of that array.



This allocation occurs when this expression is **executed**, that is, at **run-time, not at compile-time**. This means that the user can input a capacity, and the program can allocate an array with exactly that many elements!

The address returned by `new` must be assigned it to a pointer of type *`Type`*. Thus a declaration of a run-time-allocated array is simply a pointer declaration:

```
Type * arrayPtr;
```

Example

```
int numItems;
double dub[20];    // an ordinary compile-time array
double *dubPtr;    // a pointer to a (run-time) array

cout << "How many numbers do you have to process? ";
cin >> numItems;

dubPtr = new double[numItems];
```

Note: Recall that for an ordinary array like `dub`, the value of the array name `dub` is the base address of the array. So, in a subscript expression like

`dub[i]` (same as `operator[](dub, i)`)

the subscript operator actually takes two operands: the base address of the array and an integer index. since the pointer variable `dubPtr` also is the base address of an array, is can be used in the same manner as an array name:

`dubPtr[i]` (same as `operator[](dubPtr, i)`)

Example:

```
for (int i = 0; i < numItems; i++)
  cout << dubPtr[i] << endl;
```

2. Deallocating a Run-Time Array

We can use the **delete** operation in a statement of the form

`delete[]` ***`arrayPtr`*****`;`**

This returns the storage of the array pointed to by *`arrayPtr`* to the heap. This is important because **memory leaks involving arrays can result in considerable loss of memory** as in:

```
for(;;)
{
  int n;
  cout << "Size of array (0 to stop): ";
  cin >> n;
  if (n == 0) break;

  double * arrayPtr = new double[n];
  // process arrayPtr
     . . .
}
```

Each new allocation of memory to `arrayPtr` maroons the old memory block.

C. Run-Time-Allocation in Classes

Classes that use run-time allocated storage requirse some new members and modifications of others:

1. *Destructors*: To "tear down" the storage structure and deallocate its memory.
2. *Copy constructors*: To make copies of objects (e.g., value parameters)
3. *Assignment*: To assign one storage structure to another.

We will illustrate these using our `Stack` class.

1. Data Members

We will use a run-time allocated array so that the user can specify the capacity of the stack during run time. We simply change the declaration of the `myArray` member to a pointer and `STACK_CAPACITY` to a variable; to avoid confusion, we will use different names for the data members.

```
//***** RTStack.h *****
/*  -- Documentation as earlier    (: Saving space :)    --*/

#ifndef RTSTACK
#define RTSTACK

#include <iostream>
using namespace std;

template <class StackElement>
class Stack
{
/***** Function Members *****/
 public:
   . . .
/***** Data Members*****/
private:
  StackElement * myArrayPtr;  // run-time allocated array to store elements
  int myCapacity_,            // capacity of stack
      myTop_;                 // top of stack

};
#endif
```

2. The Class Constructor

We want to allow declarations such as

```
Stack<int> s1, s2(n);
```

to construct `s1` as a stack with some default capacity,
and construct `s2` as a stack with capacity `n`.

To permit both forms, we declare a constructor with a default argument:

```
/* --- Class constructor ---
   Precondition:  A stack has been defined.
   Receive:       Integer numElements > 0; (default = 128)
   Postcondition: The stack has been constructed as a stack with
                    capacity numElements.
-----------------------------------------------------------------*/
Stack(int numElements = 128);
```

This constructor must really construct something (and not just initialize data members):

```
//*** Definition of class constructor
template <class StackElement>
Stack<StackElement>::Stack(int numElements)
{
  assert (numElements > 0);   // check precondition
  myCapacity_ = numElements;  // set stack capacity
                              // allocate array of this capacity
  myArrayPtr = new StackElement[myCapacity_];

  if (myArrayPtr == 0)        // check if memory available
  {
    cerr << "*** Inadequate memory to allocate stack ***\n";
    exit(-1);
  }                           // or assert(myArrayPtr != 0);

  myTop_ = -1;
}
. . .
```

Now a program can include our `RTStack` header file and declare

```
cin >> num;
Stack<double> s1, s2(num);
```

`s1` will be constructed as a stack with capacity 128 and `s2` will be constructed as a stack with capacity `num`.

3. Other stack operations: empty, push, top, pop, output

The prototypes and definitions of `empty()`as well as the prototypes of `push()`, `top()`, `pop()`, and `operator<<()` are the same as before (except for some name changes). See pages 428-31

The definitions of `push()`, `top()`, `pop()`, and `operator<<()` require accessing the elements of the array data member. As we have noted, the subscript operator `[]` can be used in the same manner for run-time allocated arrays as for ordinary arrays, and thus (except for name changes), the definitions of these functions are the same as before; for example:

```
//*** Definition of push()
template <class StackElement>
void Stack<StackElement>::push(const StackElement & value)
{
  if (myTop_ < myCapacity_ - 1)
  {
    ++myTop_;
    myArrayPtr[myTop_] = value;
  }                  // or simply, myArrayPtr[++myTop_] = value;
  else
    cerr << "*** Stack is full -- can't add new value ***\n";
}
```

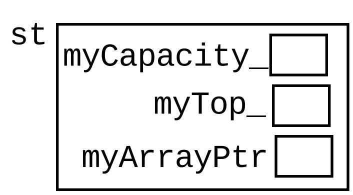
4. Class Destructor

For any class object `obj` we have used up to now, when `obj` is declared, the class constructor is called to initialize `obj`. When the lifetime of `obj` is over, its storage is reclaimed automatically because the location of the memory allocated is determined at compile-time.

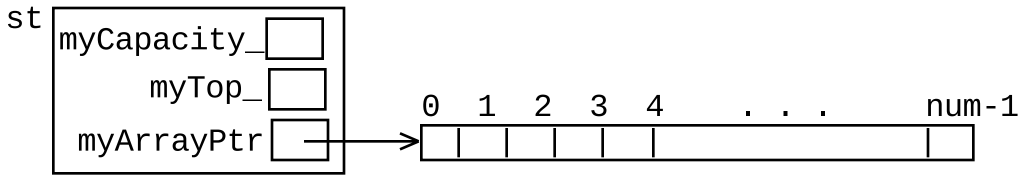
For objects created during run-time, however, a new problem arises. To illustrate, consider a declaration:

```
   . . .
Stack<double> st(num);
   . . .
```

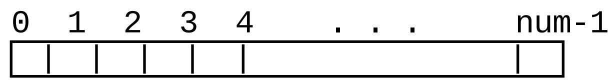
The compiler knows the data members `myCapacity_`, `myTop_`, and `myArrayPtr` of `st` so it can allocate memory for them:



The array to store stack elements is created by the constructor; so memory for it isn't allocated until run-time:



When the lifetime of `st` ends, the memory allocated to `myCapacity_`, `myTop_`, and `myArrayPtr` is automatically reclaimed, but not for the run-time allocated array:



We must add a **destructor** member function to the class to avoid this memory leak.

- Destructor's role: Deallocate memory allocated at run-time (the opposite of the constructor's role).

- At any point in a program where an object goes out of scope, the compiler inserts a call to this destructor. That is:

When an object's lifetime is over, its destructor is called first.

Form of destructor:

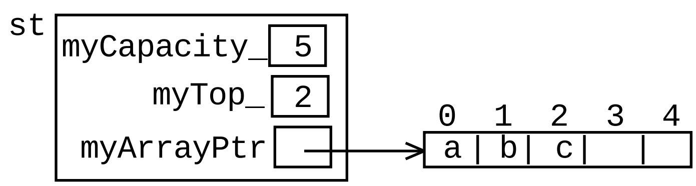
- Name is the class name preceded by a tilde (~).
- It has no arguments or return type

```
~ClassName()
```

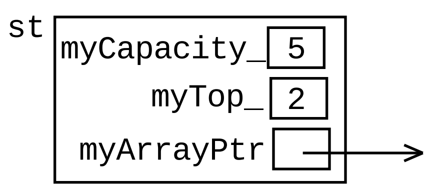
For our `Stack` class, we use the `delete` operation to deallocate the run-time array.

```
//***** RTStack.h *****
    ...
  /* --- Class destructor ---

     Precondition:  The lifetime of the Stack containing this
                      function should end.
     Postcondition: The run-time array in the Stack containing
                      this function has been deallocated.
  --------------------------------------------------------------*/

  ~Stack();

// Following class declaration
// Definition of destructor
template <class StackElement>
Stack<StackElement>::~Stack()
{
  delete[] myArrayPtr;
}
```

Suppose `st` is



When `st`'s lifetime is over, `st.~Stack()` will be called first, which produces



Memory allocated to `st` — `myCapacity_`, `myTop_`, and `myArrayPtr` — will then be reclaimed in the usual manner.

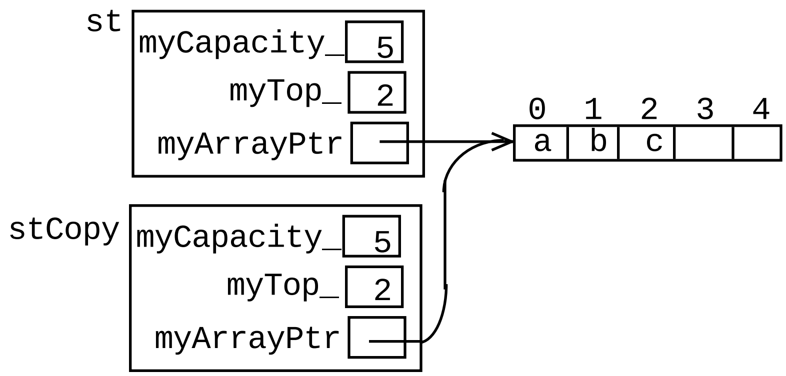
5. Copy constructor

Is needed whenever a copy of a class object must be built, which occurs:

- When a class object is passed as a value parameter
- When a function returns a class object
- If temporary storage of a class object is needed
- In initializations

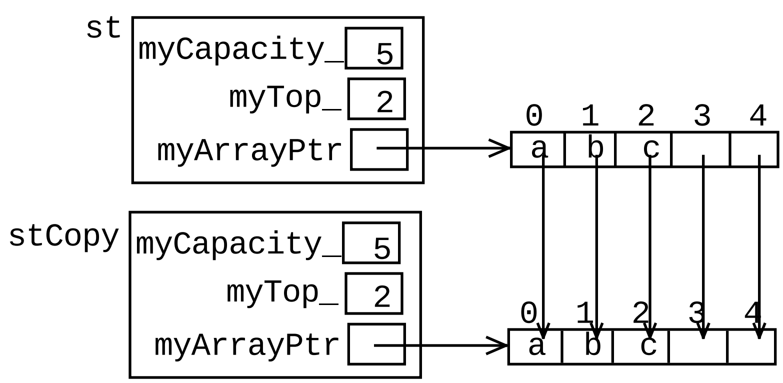
If a class has no copy constructor, the compiler uses a default copy constructor that does a byte-by-byte copy of the object. This has been adequate for classes up to now, but not for a class containing pointers to run-time allocated arrays (or other structures).

For example, a byte-by-byte copying of `st` to produce a copy `stCopy` gives



This is not correct, since copies of `myCapacity_`, `myTop_`, and `myArrayPtr` have been made, but not a copy of the run-time allocated array. Modifying `stCopy` will modify `st` also!

What is needed is to create a distinct copy of `st`, in which the array in `stCopy` has exactly the same elements as the array in `st`:



The copy constructor must be designed to do this.

Form of copy constructor:

- It is a constructor so it must be a function member, its name is the class name, and it has no return type.
- It needs a single parameter whose type is the class; this must be a reference parameter and should be const since it does not change this parameter or pass information back through it.

  (Otherwise it would be a value parameter, and since a value parameter is a copy of its argument, a call to the copy instructor will try and copy its argument, which calls the copy constructor, which will try and copy its argument, which calls the copy constructor . . .)

```
 //***** RTStack.h *****
       . . .
/* --- Copy Constructor ---
 * Precondition:  A copy of a stack is needed
 * Receive:       The stack to be copied (as a const
 *                  reference parameter)
 * Postcondition: A copy of original has been constructed.
 ************************************************************/
Stack(const Stack<StackElement> & original);
. . .

// end of class declaration

// Definition of copy constructor

template <class StackElement>
Stack<StackElement>::Stack(const Stack<StackElement> & original)
{
  myCapacity_ = original.myCapacity_;         // copy myCapacity_ member
  myArrayPtr = new StackElement[myCapacity_];   // allocate array in copy

  if (myArrayPtr == 0)                          // check if memory
  available
  {
    cout << "*** Inadequate memory to allocate stack ***\n";
    exit(-1);
  }

  for (int pos = 0; pos < myCapacity_; pos++) // copy array member
    myArrayPtr[pos] = original.myArrayPtr[pos];
  myTop_ = original.myTop_ ;                  // copy myTop_ member
}
```

6. Assignment

Assignment is another operation that requires special attention for classes containing pointers to run-time arrays (or other structures). Like the copy constructor, the default assignment operation does byte-by-byte copying. With it, the assignment statement

```
s2Copy = s2;
```

will give the same situation described earlier; the `myArrayPtr` data members of both `s2` and `s2Copy` would both point to the same anonymous array.

What is needed is to overload the assignment operator (`operator=`) so that it creates a distinct copy of the stack being assigned.

`operator=` must be a member function. So an assignment

```
stLeft = stRight;
```

will be translated by the compiler as

```
stLeft.operator=(stRight);
```

Prototype:

```
/* --- Assignment Operator ---
 *  Receive: Stack stRight (the right side of the assignment operator)
 *              object containing this member function
 *  Return (implicit parameter):  The Stack containing this
 *              function which will be a copy of stRight
 *   Return (function): A reference to the Stack containing
 *              this function
 **************************************************************/
Stack<StackElement> & operator=(const Stack<StackElement> & original);
```

The return type is a reference to a `Stack` since `operator=()` must return the object on the left side of the assignment and not a copy of it (to make chaining possible).

Definition:

It is quite similar to that for the copy constructor, but there are some differences:

1. The `Stack` on the left side of the assignment may already have a value.
   Must destroy it —deallocate the old so no memory leak and allocate a new one

2. Assignment must be concerned with self-assignments: `st = st;`
   Can't destroy the right old value in this case.

3. `operator=()` must return the `Stack` containing this function.

For this we use the following property of classes:

> Every member function of a class has access to a (hidden) pointer constant
> **`this`**
> whose value is the *address of the object containing this function*. The expression
> **`*this`**
> refers to *the object* itself.

We can now write the definition of `operator=()`:

```
//*** Definition of operator=
template <class StackElement>
Stack<StackElement> &
   Stack<StackElement>::operator=(const Stack<StackElement> & original)
{
  if (this != &original)                         // check that not st = st
  {
    delete[] myArrayPtr;                         // destroy previous array

    myArrayPtr = new StackElement[myCapacity_]; // allocate array in copy
    if (myArrayPtr == 0)                       // check if memory available
    {
      cerr << "*** Inadequate memory to allocate stack ***\n";
      exit(-1);
    }

    myCapacity_ = original.myCapacity_;         // copy myCapacity_ member

    for (int pos = 0; pos < myCapacity_; pos++) // copy array member
      myArrayPtr[pos] = original.myArrayPtr[pos];
    myTop_ = original.myTop_ ;                  // copy myTop_ member
  }

  return *this;                                 // return reference to
}                                               //   this object
```
==

```
//***** Test Driver ************************
#include <iostream>
using namespace std;
#include "RTStack.h"


Print (Stack<int> st)
{
  cout << st;
}


int main()
{
  int Size;
  cout << "Enter stack size: ";
  cin >> Size;

  Stack<int> S(Size);
  for (int i = 1; i <= 5; i++)
    S.push(i)

  Stack<int> T = S;
  cout << T << endl;
}
```

Sample Runs:

```
Enter stack capacity: 5
5
4
3
2
1
----------------------------------

Enter stack capacity: 3
*** Stack is full -- can't add new value ***
*** Stack is full -- can't add new value ***
3
2
1
----------------------------------

Enter stack capacity: 0
StackRT.cc:12: failed assertion `NumElements > 0'
Abort
```

Test driver with statements in the constructor, copy constructor, and destructor to trace when they

See Figure 8.7 on pp. 440-2

```
//***** Test Driver *****
#include <iostream>
using namespace std;

#include "StackRTemp1"

Print (Stack<int> st)
{
  cout << st;
}

int main()
{
  int numElements;
  cout << "Enter stack capacity: ";
  cin >> numElements;

  cout << "**A**\n";
  Stack<int> s(numElements);
  cout << "**B**\n";
  for (int i = 1; i <= 5; i++)
    {
      cout << "**C**\n";
      s.push(i);
    }
  cout << "**D**\n";
  Stack<int> t = s;
  cout << "**E**\n";
  Print(t);
  cout << "**F**\n";
  Stack<int> u;
  cout << "**G**\n";
  u = t;
  cout << "**H**\n";
  Print(u);
  cout << "**I**\n";
}
```

```
Enter stack capacity: 5
**A**
CONSTRUCTOR
**B**
**C**
**C**
**C**
**C**
**C**
**D**
COPY CONSTRUCTOR
**E**
COPY CONSTRUCTOR
5
4
3
2
1
DESTRUCTOR
**F**
CONSTRUCTOR
**G**
**H**
COPY CONSTRUCTOR
5
4
3
2
1
DESTRUCTOR
**I**
DESTRUCTOR
DESTRUCTOR
DESTRUCTOR
```

Part 2: LinkedLists and Other Linked Structures (Chap 8: §1-3, §6-8, Chap. 9)

D. Introduction to Lists (§8.1)

1. As an abstract data type, a **list** is a finite sequence (possibly empty) of elements with basic operations that vary from one application to another, but that commonly include:

Construction:	Usually constructs an empty list
Empty:	Check if list is empty
Traverse:	Go through the list or a part of it, accessing and processing the elements in order
Insert:	Add an item at any point in the list.
Delete:	Remove an item from the list at any point.

2. Array/Vector-Based Implementation of a List

Data Members:

Store the list items in consecutive array or `vector` locations:

```
a1,    a2,    a3 ,   . . .    an
 ↓      ↓      ↓               ↓
a[0] a[1] a[2] ... a[n-1] a[n] ... a[CAPACITY-1]
```

For an array, add a `mySize` member to store the length (`n`) of the list

Basic Operations

Construction: For array: Set `mySize` to 0; if run-time array, allocate memory for it
For `vector:` let its constructor do the work.

Empty: `mySize` == 0
For `vector`: Use its `empty()` operation

Traverse:

```
for (int i = 0; i < size; i++)
{  Process(a[i]);  }
```

or

```
i = 0;
while (i < size)
{  Process(a[i]);
   i++;
}
```

Insert: Insert 6 after 5 in 3, 5, 8, 9, 10, 12, 13, 15

3, 5, 6, 8, 9, 10, 12, 13, 15

Have to shift array elements to make room.

Delete: Delete 5 from preceding list:

3, 5, 6, 8, 9, 10, 12, 13, 15

3, 6, 8, 9, 10, 12, 13, 15

Have to shift array elements to close the gap.

E. Introduction to Linked Lists (§8.2)

The preceding implementation of lists is inefficient for *dynamic* lists (those that change frequently due to insertions and deletions), so we look for an alternative implementation . Minimal requirements: We must be able to:

1. Locate the first element.
2. Given the location of any list element, find its successor.
3. Determine if at the end of the list.

For the array/vector-based implementation:

1. At location 0
2. Successor of item at location i is at location i + 1
3. At location *size* – 1

The inefficiency is caused by #2; relaxing it by not requiring that list elements be stored in consecutive location leads us to linked lists.

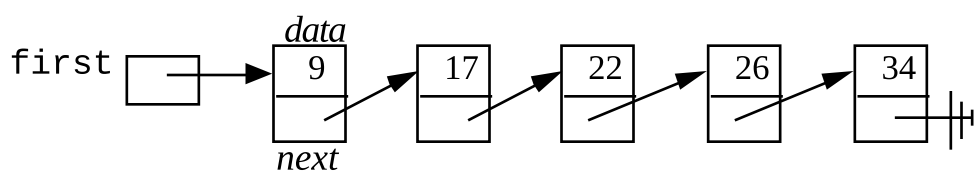
1. A **linked list** is an ordered collection of elements called **nodes** each of which has two parts:

   (1) **Data part**: Stores an element of the list;

   (2) **Next part**: Stores a link (pointer) to the location of the node containing the next list element. If there is no next element, then a special null value is used.

   Also, we must keep track of the location of the node storing the first list element, This will be the null value, if the list is empty.

   Example: A linked list storing 9, 17, 22, 26, 34:



2. Basic Operations:

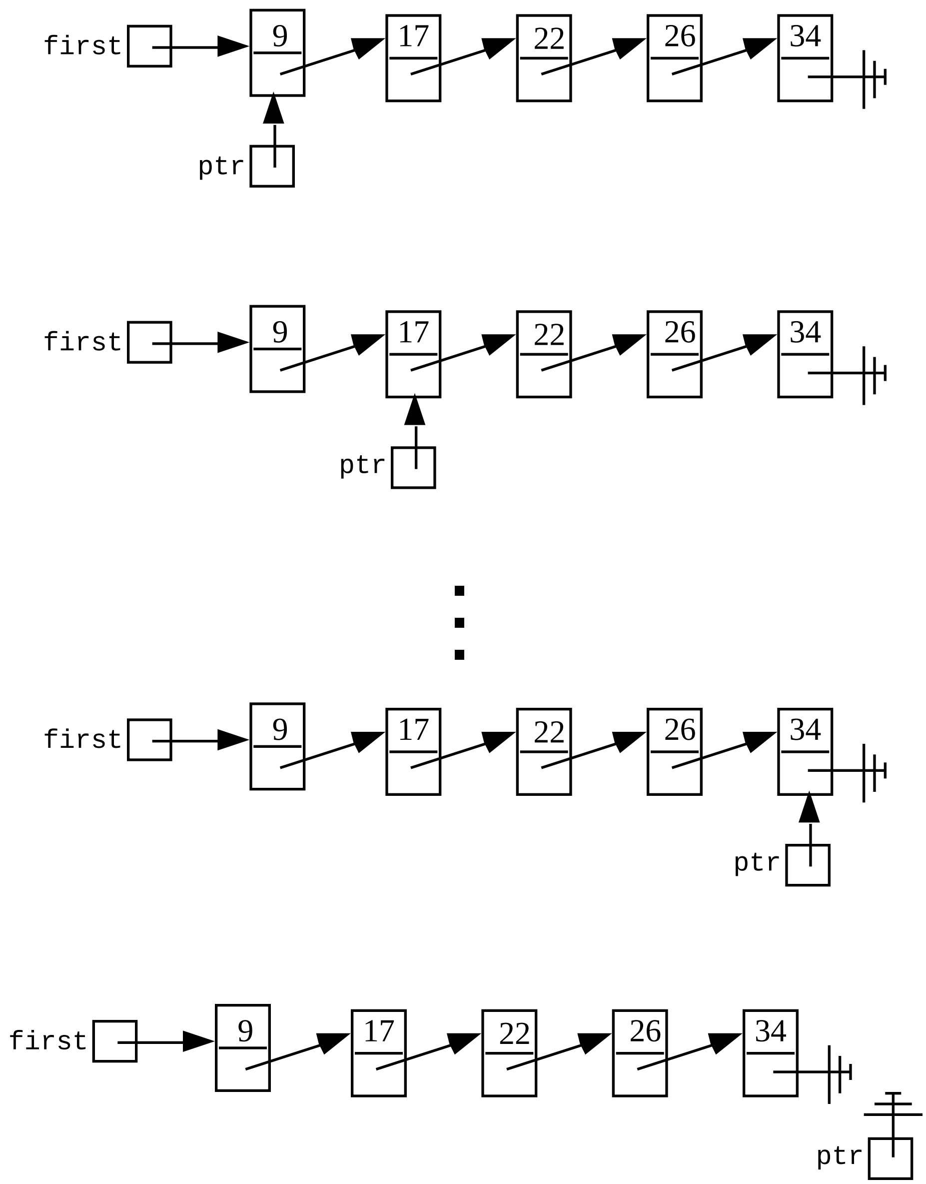
   Construction: `first = null_value;`

   Empty: `first == null_value?`

   Traverse:

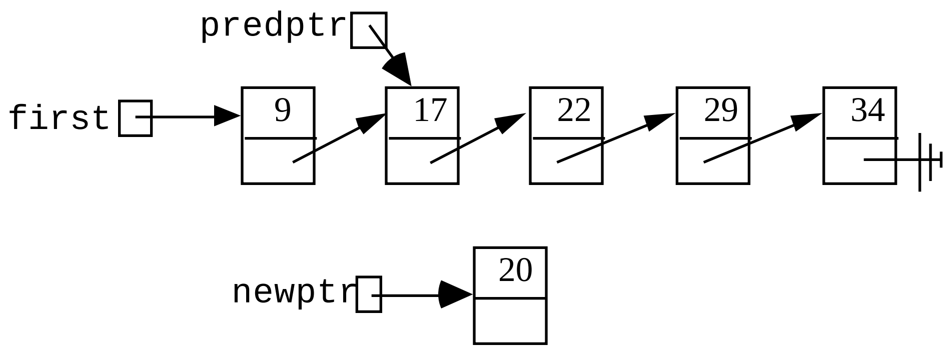
```
ptr = first;
while (ptr != null_value)
{
  Process data part of node pointed to by ptr;
  ptr = next part of node pointed to by ptr;
}
```

See pp. 391-2

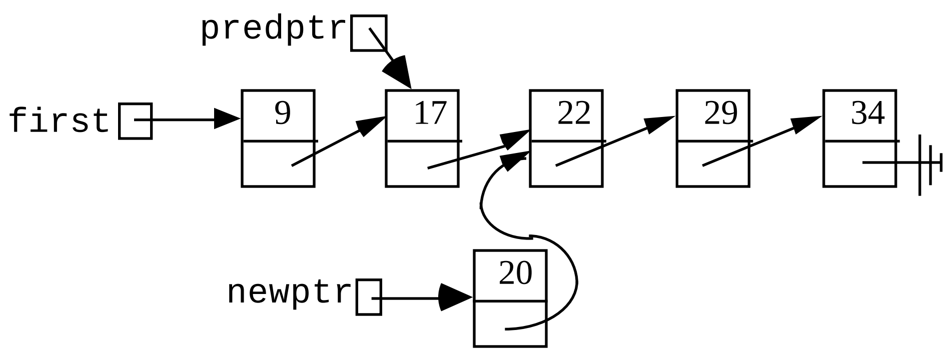

first
9
17
22
26
34
ptr
first
9
17
22
26
34
ptr
first
9
17
22
26
34
ptr
first
9
17
22
26
34
ptr

Insert: Insert 20 after 17 in the preceding linked list; suppose `predptr` points to the node containing 17.

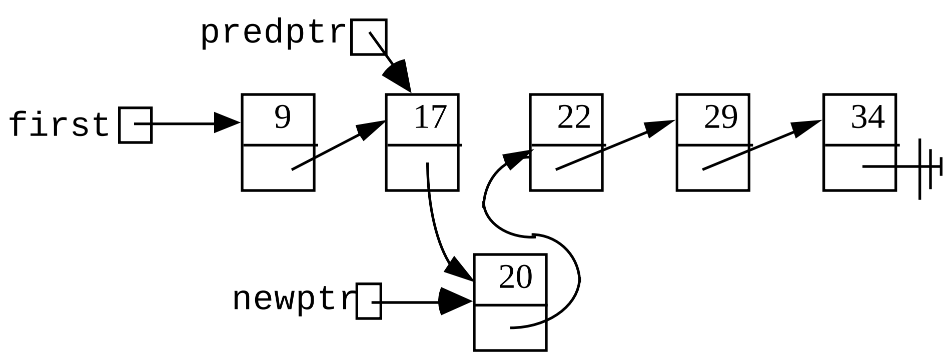
(1) Get a new node pointed to by new`ptr` and store 20 in it



(2) Set the next pointer of this new node equal to the next pointer in its predecessor, thus making it point to its successor.



(3) Reset the next pointer of its predecessor to point to this new node.



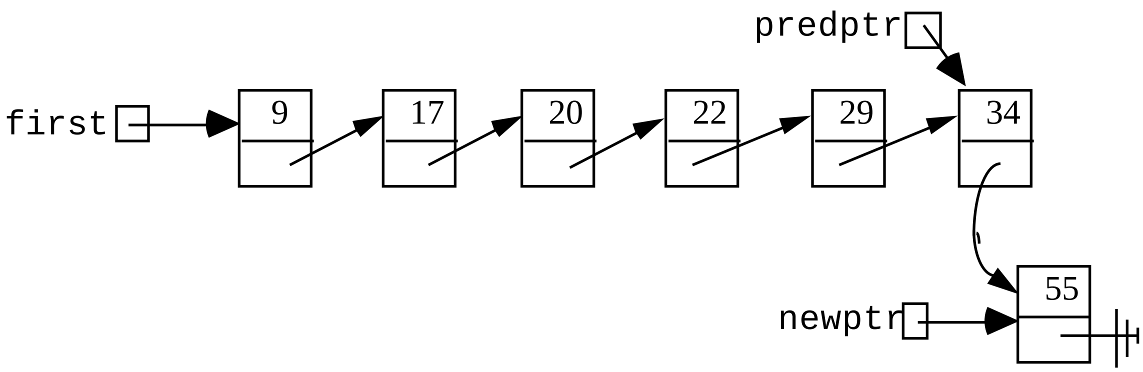
Note that this also works at the end of the list.

Example: Insert a node containing 55 at the end of the list.

(1) as before

(2) as before — sets next link to null pointer

(3) as before



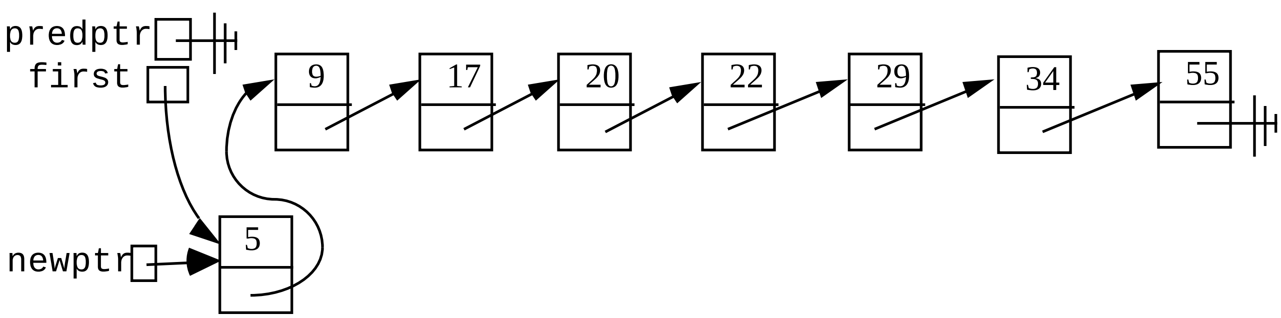
Inserting at the beginning of the list requires a modification of step 3:

Example: Insert a node containing 5 at the beginning of the list.

(1) as before

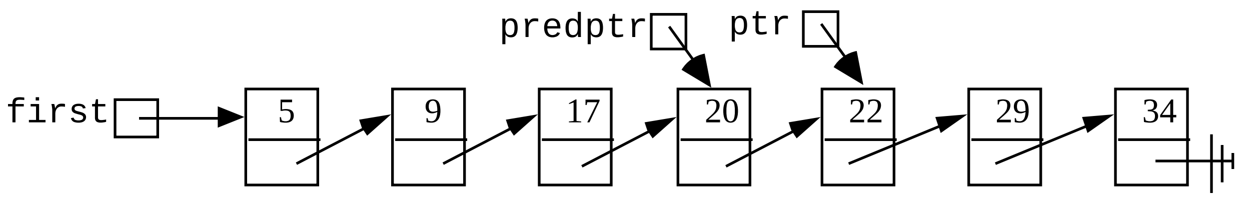
(2) sets next link to first node in the list

(3) set `first` to point to new node.

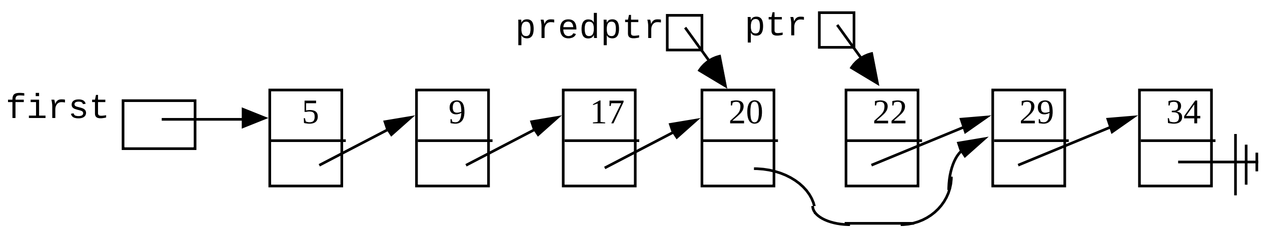


☞ <u>**Note: In all cases, no shifting of list elements is required !**</u>

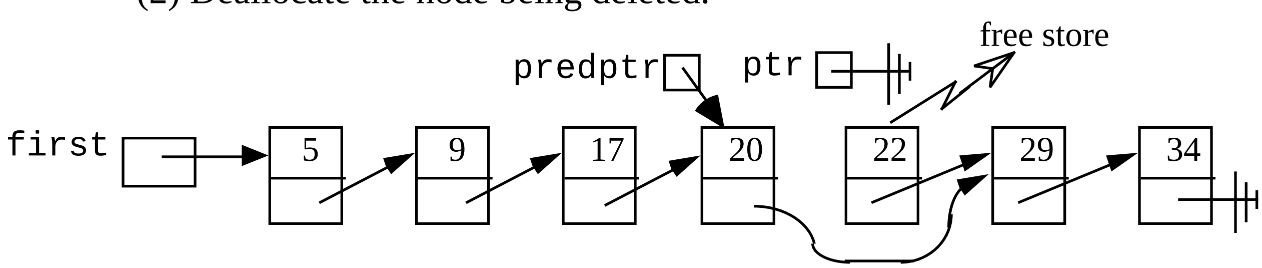
Delete: Delete node containing 22 from the following linked list; suppose `ptr` points to the node to be deleted and `predptr` points to its predecessor (the node containing 20):.



(1) Do a bypass operation: Set the next pointer in the predecessor to point to the successor of the node to be deleted

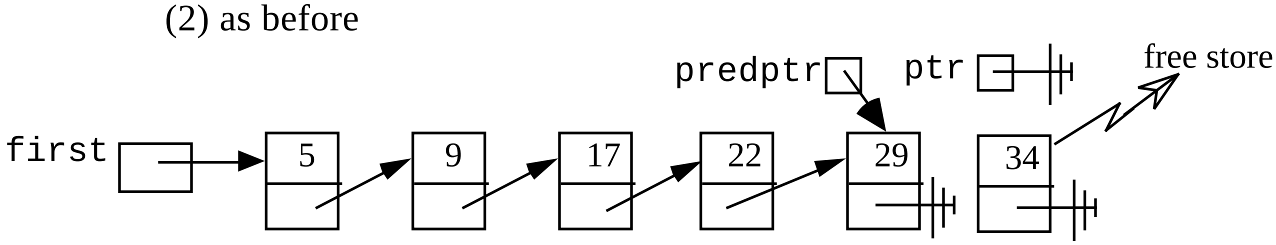


(2) Deallocate the node being deleted.



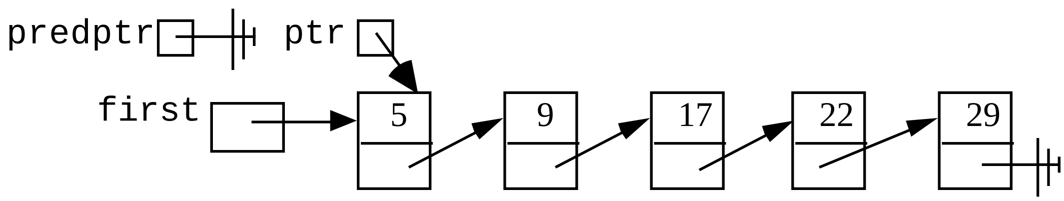
Note that this also works at the end of the list.
Example: Delete the node at the end of the list.

(1) as before — sets next link to null pointer
(2) as before

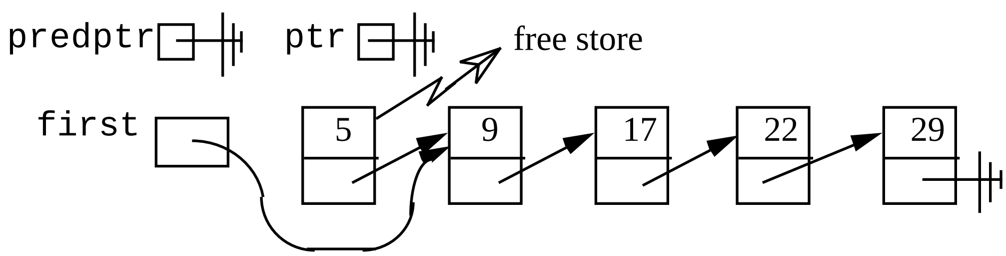


Deleting at the beginning of the list requires a modification of step 1:

Example: Delete 5 from the previous list



(1) reset `first`
(2) as before



☞ **Note: In all cases, no shifting of list elements is required !**

3. We gain a lot with linked lists. Do we lose anything?

We no longer have direct access to each element of the list;
we have direct access only to the first element.

List-processing algorithms that require fast access to each element cannot (usually) be done as efficiently with linked lists:

Example: Appending a value at the end of the list:

— Array-based method:

```
a[size++] = value;
```

or for a vector:

```
v.push_back(value);
```

— For a linked list:

Get a new node; set data part = `value` and next part = *`null_value`*
If list is empty
    Set `first` to point to new node.
else
    Traverse list to find last node
    Set next part of last node to point to new node.

Other examples: Many sorting algorithms need direct access
Binary search needs direct access

F. Implementing Linked Lists

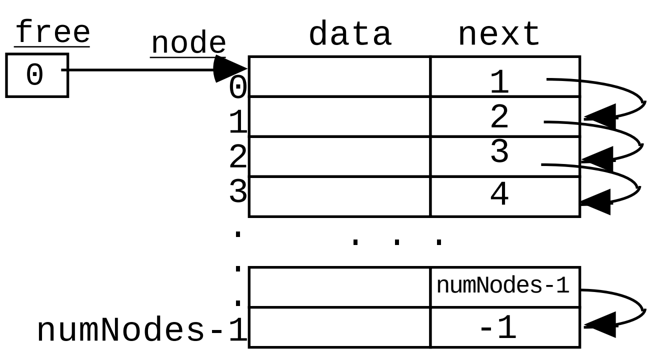
1. Linked lists can be implemented in many ways. For example, we could use arrays/vectors (Read §8.3)

For nodes:

```
 typedef int DataType;  // DataType is type of list elements
 typedef int Pointer;   // pointers are array indices
 struct NodeType
 {
   DataType data;
   Pointer next;
 };
```

For free store:

```
const int NULL_VALUE = -1;

const int numberOfNodes = 2048;
NodeType node[numberOfNodes];
Pointer free;          // points to a free node

// Initialize free store
// Each node points to the next one

for (int i = 0; i < numberOfNodes - 1; i++)
  node[i].next = i + 1;
node[numberOfNodes - 1].next = NULL_VALUE;
free = 0;
```




```
// Maintain free store as a stack
// New operation
   Pointer New()
   { Pointer p = free;
     if (free != NULL_VALUE)
       free = node[free].next;
     else
       cerr << "***Free store empty***\n";
     return p;
   }
```

```
// Delete operation
   void Delete(Pointer p)
   { node[p].next = free;
     free = p;
   }
```

For the linked list operations:

Use `node[p].data` to access the data part of node pointed to by `p`
Use `node[p].next` to access the next part of node pointed to by `p`

Example: Traversal

```
Pointer p = first;
while (p != NULL_VALUE)
{
  Process(node[p].data);
  p = node[p].next;
}
```

2. Implementing Linked Lists Using C++ Pointers and Classes (§8.6)

a. To Implement Nodes

```
class Node
{
 public:
   DataType data;
   Node * next;
};
```

Note: The definition of a `Node` is a *recursive (or self-referential) definition* because it uses the name `Node` in its definition: the `next` member is defined as a pointer to a `Node`.

b. How do we declare pointers, , assign them, access contents of nodes, etc.?

Declarations:

`Node * ptr;` or **`typedef Node * NodePointer;`**
`NodePointer ptr;`

Allocate and Deallocate:

`ptr = new Node;` **`delete ptr;`**

To access the `data` and `next` part of node:

`(*ptr).data` and **`(*ptr).next`**

or better, use the **`->` operator**

`ptr->data` and **`ptr->next`**

Why make data members public in class `Node`?

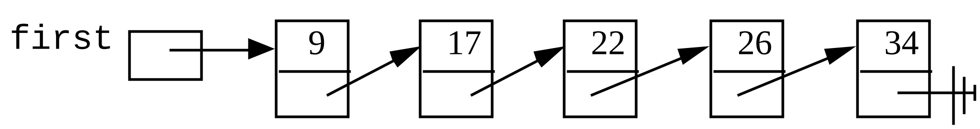
This class declaration will be placed inside another class declaration for `LinkedList`. The data members `data` and `next` of struct `Node` will be public inside the class and thus will accessible to the member and friend functions of the class, but they will be private outside the class.

```
#ifndef LINKEDLIST
#define LINKEDLIST

typedef int DataType;

class LinkedList
{
 private:
   class Node
   {
    public:
     DataType data;
     Node * next;
   };
  typedef Node * NodePointer;
  . . .
};
#endif
```

So why not just make `Node` a struct? We could, but it is common practice to use struct for C-style structs that contain no functions (and we will want to add a few to our `Node` class.)

b. Data Members for `LinkedList`s

Linked lists like

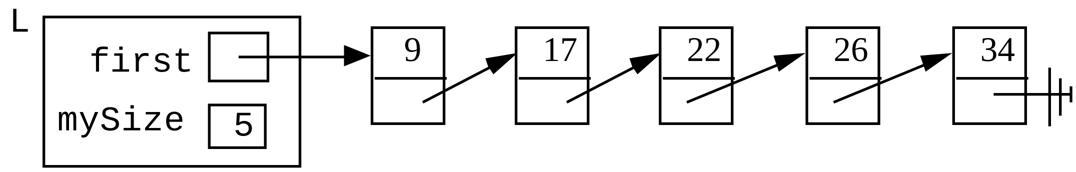


are characterized by:

(1) There is a pointer to the first node in the list.
(2) Each node contains a pointer to the next node in the list.
(3) The last node contains a null pointer.

We will call the kind of linked lists we've just considered *simple linked lists* to distinguish them from other variations we will consider shortly — circular, doubly-linked, lists with head nodes, etc..

For simple linked lists, only one data member is needed: a pointer to the first node. But, for convenience, another data member is usually added that keeps a count of the elements of the list:



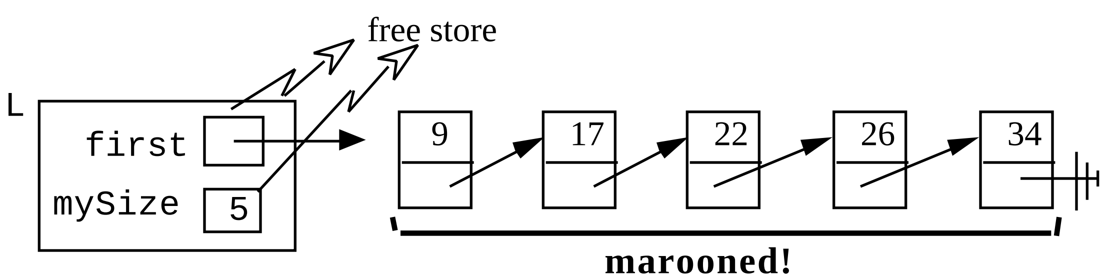
Otherwise we would have to traverse the list and count the elements each time we need to know the list's length.
(See p. 446)

1. Set `count` to 0.
2. Make `ptr` point at the first node.
3. While `ptr` is not null:
   a. Increment `count`.
   b. Make `ptr` point at the next node.
4. Return `count`.

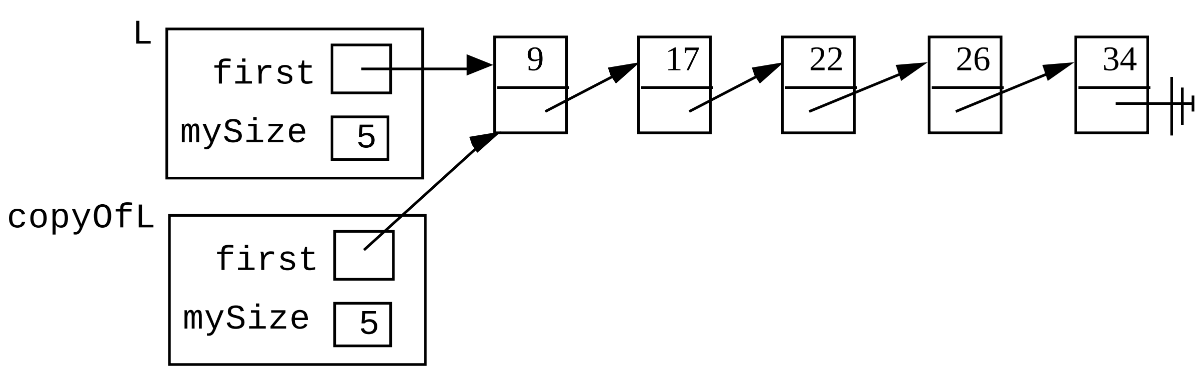
c. Function Members for `LinkedList`s

Constructor: Make `first` a null pointer and set `mySize` to 0.

Destructor: Why is one needed? For the same reason as for run-time arrays.
If we don't provide one, the default destructor used by the compiler for a linked list like that above will result in:

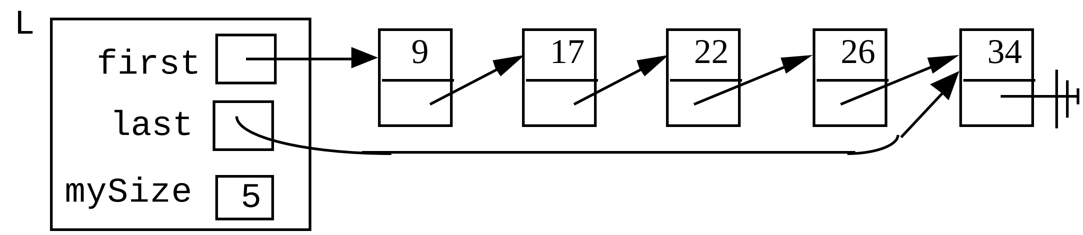


Copy constructor: Why is one needed? For the same reason as for run-time arrays.
If we don't provide one, the default copy constructor (which just does a byte-by-byte copy) used by the compiler for a linked list like `L` will produce:

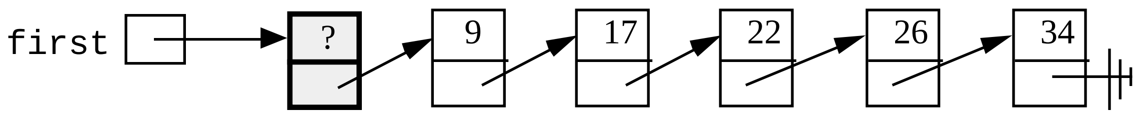


d. Other Kinds of Linked Lists (§9.1)

i. In some applications, it is convenient to keep access to both the first node and the last node in the list.

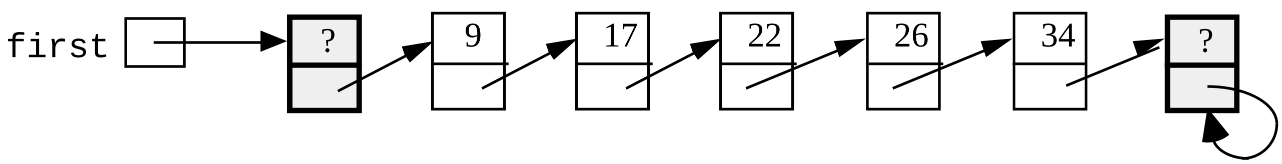


ii. Sometimes a **head node** is used so that **every node has a predecessor**, which thus eliminates special cases for inserting and deleting.



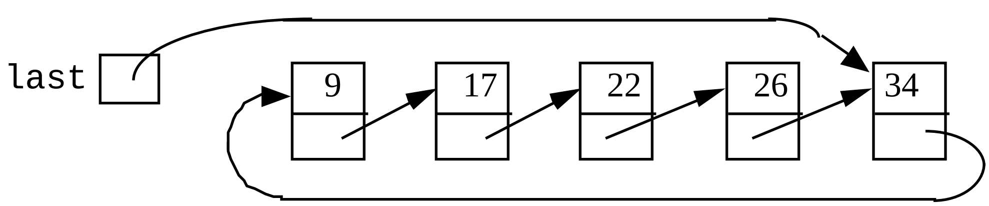
The data part of the head node might be used to store some information about the list, e.g., the number of values in the list.

iii. Sometimes a **trailer node** is also used so that **every node has a successor**.

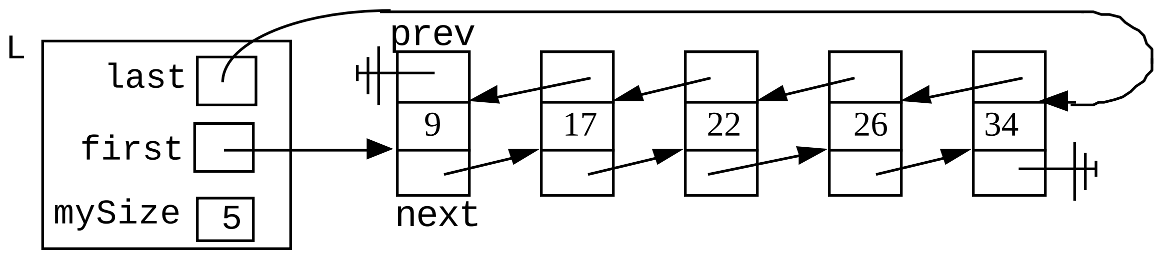


(Two or more lists can share the same trailer node.)

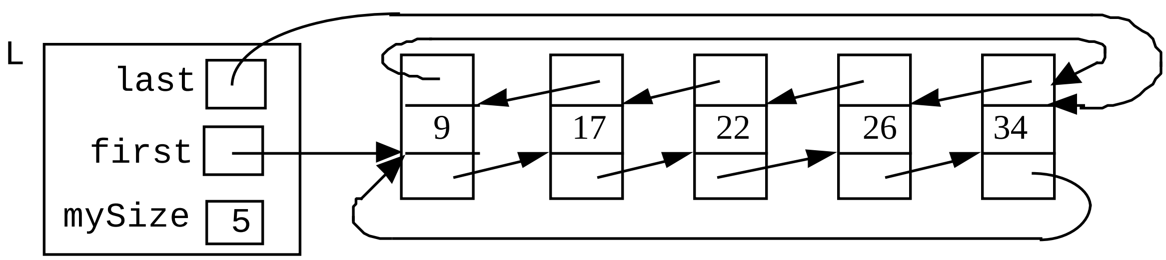
iv. In other applications (e.g., linked queues), a **circular linked list** is used; instead of the last node containing a NULL pointer, it contains a pointer to the first node in the list. For such lists, one can use a single pointer to the last node in the list, because then one has direct access to it and "almost-direct" access to the first node.



v. All of these lists, however, are uni-directional; we can only move from one node to the next. In many applications, bidirectional movement is necessary. In this case, each node has two pointers — one to its successor (null if there is none) and one to its precedessor (null if there is none.) Such a list is commonly called a **doubly-linked** (or **symmetrically-linked**) **list**.



vi. And of course, we could modify this doubly-linked list so that both lists are circular forming a **doubly-linked ring**.



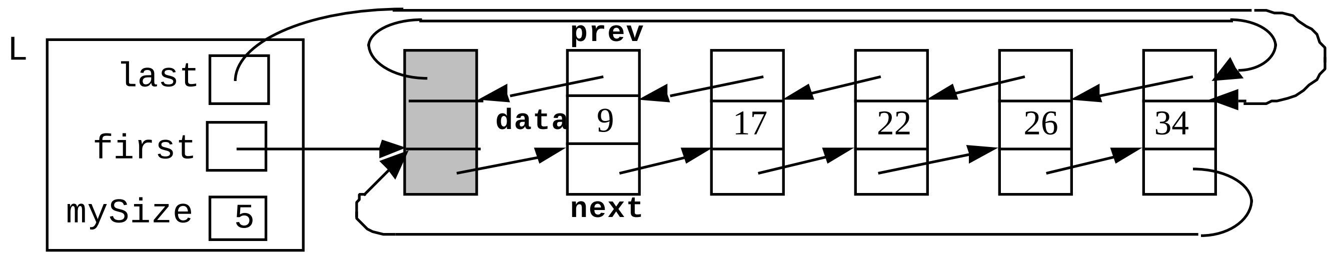
Add a head node and we have the implementation used in **STL's `list` class**.

G. The STL `list` Class Template

`list` is a sequential container that is optimized for insertion and erasure at arbitrary points in the sequence.

1. Implementation

   As a circular doubly-linked list with head node.



   Its node structure is:

```
struct list_node
{
  pointer next,
          prev;
  T data;
}
```

2. Allocation/Deallocation:

   On the surface, `list` looks quite simple. But it's allo/deallo-cation scheme is more complex than simply using `new` and `delete` operations. To reduce the inefficiency of using the heap manager for large numbers of allo/deallo-cations, it does it's own memory management.

   Basically, for each list of a certain type T:

   When a node is needed:

   1. If there is a node on the free list, allocate it.
      (This is maintained as a linked stack in exactly the way we described earlier.)
   2. If the free list is empty:
      a. Call the heap manager to allocate a block (called a *buffer*) of size (usually) 4K bytes.
      b. Carve it up into pieces of size required for a node of a `list<T>`.

   When a node is deallocated:

   Push it onto the free list.

   When ***all*** lists of this type T have been destroyed:
   Return all buffers to the heap.

3. Comparing `list` with other containers (p. 450)

Property	Array	`vector`	`deque`	`list`
Direct/random access (`[]`)	√+	√+	√	X
Sequential access	√+	√+	√	√+
Insert/delete at front	√−	√−	√+	√+
Insert/delete in middle	√−	√−	√−	√+
Insert/delete at end	√+	√+	√+	√+
Overhead	lowest	low	low/medium	high

As the table indicates, `list` does not support direct/random access and thus does not provide the subscript operator `[]`.

4. `list` iterators (p. 451)

`list`'s iterator is "weaker" than that for `vector`. (`vector`'s is called a *random access* iterator and `list`'s is a *bidirectional* iterator. They have the following operations in common:

- **`++`** Move iterator to the next element (like ptr = ptr-> next)
- **`--`** Move iterator to the preceding element (like ptr = ptr-> prev)
- `*` dereferencing operator: to access the value stored at the position to which an iterator points (like ptr-> data)
- = assignment: for same type iterators, `it1 = it2` sets `it1`'s position to same as `it2`'s
- == and `!=` for same type iterators, `it1 == it2` is true if `it1` and `it2` are both positioned at the same element

but bidirectional iterators *do not have*:

addition (+) and subtraction (-)
the corresponding shortcuts (+=, -=),
subscript (`[]`)

This means that algorithms such as `sort()` which require direct/random access cannot be used with `list`s.

Example: Construct a list containing first 4 even integers; then output the list.

```
list<int> l;

for (int i = 1; i <= 4; i++)
  l.push_back(2*i);

for (list<int>::iterator it = l.begin(); it != l.end(); it++)
  cout << *it << "  ";
cout << endl;
```

5. `list` member functions and operators (See Table 8.1)

Function Member	**Description**

Constructors
`list<T> l;`
`list<T> l(n);`
 	
`list<T> l(n, initVal);`
 	
`list<T> l(fPtr, lPtr);`
 	
Copy constructor
	
	Construct *l* as an empty `list<T>`
	Construct *l* as a `list<T>` to contain *n* elements (set to default value)
	Construct *l* as a `list<T>` to contain *n* copies of *initVal*
	Construct *l* as a `list<T>` to contain copies of elements in memory locations `fptr` up to `lptr` (pointers of type `T *`)

Denstructor `~list()`	Destroy contents, erasing all items.
l`.empty()` *l*`.size()`	Return `true` if and only if *l* contains no values Return the number of values *l* contains
l`.push_back(`*value*`);` *l*`.push_front(`*value*`);` *l*`.insert(`*pos*`, `*value*`)` *l*`.insert(`*pos*`, n, `*value*`);` *l*`.insert(`*pos*`, `*fPtr*`, `*lPtr*`);`	Append *value* at *l*'s end Insert *value* in front of *l*'s first element Insert *value* into *l* at iterator position *pos* and return an iterator pointing to the new element's position Insert *n* copies of *value* into *l* at iterator position *pos* Insert copies of all the elements in the range [*fPtr*, *lPtr*) at iterator position *pos*
l`.pop_back();` *l*`.pop_front();` *l*`.erase(`*pos*`);` *l*`.erase(`*pos1*`, `*pos2*`);` *l*`.remove(`*value*`);` *l*`.unique()`	Erase *l*'s last element Erase *l*'s first element Erase the value in *l* at iterator position *pos* Erase the values in *l* from iterator positions *pos1* to *pos2* Erase all elements in *l* that match *value*, using `==` to compare items. Replace all repeating sequences of a single element by a single occurrence of that element.
l`.front()` *l*`.back()`	Return a reference to *l*'s first element Return a reference to *l*'s last element
l`.begin()` *l*`.end()`	Return an iterator positioned to *l*'s first value Return an iterator positioned 1 element past *l*'s last value
l`.rbegin()` *l*`.rend()`	Return a reverse iterator positioned to *l*'s last value Return a reverse iterator positioned 1 element before *l*'s first value
l`.sort();` *l*`.reverse();`	Sort *l*'s elements (using `<`) Reverse the order of *l*'s elements

`l1.merge(l2);`	Remove all the elements in *l2* and merge them into *l1*; that is, move the elements of *l2* into *l1* and place them so that the final list of elements is sorted using <; (Assumes both *l2* and *l1* were sorted using <)
`l1.splice(pos, l2);`	Remove all the elements in *l2* and insert them into *l1* at iterator position *pos*
`l1.splice(to, l2, from);`	Remove the element in *l2* at iterator position *from* and insert it into *l1* at iterator position *to*
`l1.splice(pos, l2, first, last);`	Remove all the elements in *l2* at iterator positions [*first*, *last*)and insert them into *l1* at iterator position *pos*
`l1.swap(l2);`	Swap the contents of *l1* with *l2*

Operator	**Description**
`l1 = l2`	Assign to *l1* a copy of *l2*
`l1 == l2`	Return `true` if and only if *l1* contains the same items as *l2*, n the same order
`l1 < l2`	Return `true` if and only if *l1* is lexicographically less than *l2*

6. Sample program illustrating list operations (See Figure 8.8)

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

ostream & operator<<(ostream & out, list<int> l)
{
  for (list<int>::iterator i = l.begin(); i != l.end(); i++)
    out << *i << "  ";
  return out;
}

int main()
{
  list<int> l, l1(4, 111), l2(6);

  cout << "l:  " << l << "  size = " << l.size() << endl;
  cout << "l1: " << l1 << "  size = " << l1.size() << endl;
  cout << "l2: " << l2 << "  size = " << l2.size() << endl;

  // construct l3 from an array
  int b[] = {2, 22, 222,2222};
  list<int> l3(b, b+4);
  cout << "l3: " << l3 << endl;

  // assignment
  cout << "\nAssignments l = l3 and  l2 = l3:" << endl;
  l = l3;
  l2 = l3;
  cout << "l = " << l << "  size = " << l.size() << endl;
  cout << "l2 = " << l2 << "  size = " << l2.size() << endl;

  cout << "\nInserts in l1:\n";
  list<int>::iterator i;
  i = l1.begin();
  i++; i++;
  l1.insert(i, 66666);
  cout << l1 << endl;

  l1.insert(i,3, 555);
  cout << l1 << endl;

  l1.insert(i, b, b+3);
  cout << l1 << endl;

  l1.push_back(888);
  l1.push_front(111);
  cout << l1 << endl;
```

```
  cout << "\nErases in l1:\n";
  i = find(l1.begin(), l1.end(), 66666);  // find is an algorithm
  if (i != l1.end())
    {
      cout << "66666 found -- will erase it\n";
      l1.erase(i);
    }
  else
    cout << "66666 not found\n";
  cout << l1 << endl;

  i = l1.begin(); i++;
  list<int>::iterator j = l1.end();
  --j; --j; i = --j; i --; i--;
  l1.erase(i,j);
  cout << l1 << endl;

  l1.pop_back();
  l1.pop_front();
  cout << l1 << endl;

  cout << "\nReverse l3:\n";
  l3.reverse();
  cout << l3 << endl;

  cout << "\nSort l1:\n";
  l1.sort();
  cout << l1 << endl;

  cout << "\nMerge l1 and l3:\n";
  l1.merge(l3);
  cout << "l1: " << l1 << endl;
  cout << "l3: " << l3 << endl;

  cout << "\nSplice l2 into l at second position:\n";
  i=l.begin(); i++;
  l.splice(i, l2);
  cout << "l: " << l << endl;
  cout << "l2: " << l2 << endl;

  cout << "\nRemove 22s from l:\n";
  l.remove(22);
  cout << l << endl;

  cout << "\nUnique applied to l1:\n";
  l1.unique();
  cout << l1 << endl;

}
```

Output:

```
l:    size = 0
l1: 111  111  111  111    size = 4
l2: 0  0  0  0  0  0    size = 6
l3: 2  22  222  2222

Assignments l = l3 and  l2 = l3:
l = 2  22  222  2222    size = 4
l2 = 2  22  222  2222    size = 4

Inserts in l1:
111  111  66666  111  111
111  111  66666  555  555  555  111  111
111  111  66666  555  555  555  2  22  222  111  111
111  111  111  66666  555  555  555  2  22  222  111  111  888

Erases in l1:
66666 found -- will erase it
111  111  111  555  555  555  2  22  222  111  111  888
111  111  111  555  555  555  2  111  111  888
111  111  555  555  555  2  111  111

Reverse l3:
2222  222  22  2

Sort l1:
2  111  111  111  111  555  555  555

Merge l1 and l3:
l1: 2  111  111  111  111  555  555  555  2222  222  22  2
l3:

Splice l2 into l at second position:
l: 2  2  22  222  2222  22  222  2222
l2:

Remove 22s from l:
2  2  222  2222  222  2222

Unique applied to l1:
2  111  555  2222  222  22  2
```