C++ Dynamic Memory Allocation

The amount of memory required for program variables cannot always be predicted in advance. Therefore, many programming languages, including C++, allow running programs to request additional storage space. The allocation of memory at run-time is called dynamic **memory allocation**. One type of dynamic memory allocation is used to create **dynamic arrays**, i.e. arrays whose size can be determined at run-time. Dynamic arrays are useful in situations where the programmer cannot predict in advance how large an array might be needed. A simple call to the **new operator** will request memory allocation from the operating system and will create the array.

The **new operator** will request a certain amount of memory from the operating system and then if memory is available assign this memory location, address, to a pointer. A pointer variable will then hold the address of this requested or dynamic memory. But what happens if there is not enough memory available for this request? Depending on the C++ compiler you are using and what header files may or may not be included; one of two things will happen if the operator **new** fails:

1) The operator **new** will return a **0** or **NULL**, which is assigned to the pointer variable.
2) It **throws** a **bad allocation exception**, which can be handled with a **try-catch** structure. This try-catch structure is very similar to Visual Basic try-catch, which is similar to an if-else structure.

This handout will cover only the first option, which if it is not the default of the compiler, it can be made to work this way. In Visual C++ .NET this is done by having "`#include <new>`" at the top and when ever the operator **new** is used, use a "`nothrow`", example "`new(nothrow)`".

Following is code segment for creating an array at run-time.

```
unsigned ArraySize;

cout << "How big an array would you like? ";
cin >> ArraySize;

int *A;

A = new(nothrow) int[ArraySize];      //try to create new array

if ( A != NULL )                      //was array created?
  cout << "The array is created!";
else                                    //if not, handle error
{
  cerr << "Error trying to allocate array of length "
         << ArraySize << "\nExiting...\n";
  return 1;
}
```

As a more realistic example, consider a function prototyped as follows:

```
void NeedCopy (char A[]);
```

that needs to declare as a local variable an array the same size as A, so that a copy can be made for temporary use in NeedCopy. There simply is no way to predict in advance the size of the char array A, and hence no way to determine the size needed for the local array in the function.

However, one can compute the length of A and then make a local array dynamically with the call to **new**, as follows:

```
int Length = strlen(A);          // Find length of array A
char *CopyA = new(nothrow) char[Length + 1];    //create array and allow
                                                //1 extra space for NULL
strcpy(CopyA, A);                //Make a copy of A
// Work with CopyA
....
delete[] CopyA;                  //Don't create memory leak!
                                 //reallocate memory to heap.
```

It is very important to call the **delete operator** before exiting a function such as NeedCopy that uses dynamic memory. If you don't, the memory allocated for CopyA becomes unavailable for future use and a **memory leak** results. Memory leaks in operating system software are deadly and lead to system crashes.

Arrays Have Shortcomings

Arrays are not appropriate for certain kinds of programming needs. Consider the following.

1) Although arrays are convenient method for storing information, it is often impossible for a programmer to determine in advance (even at run-time) just how big an array should be. It would be helpful if at run-time we could create small chunks of storage on an "as needed" basis and add it to what we already have.

2) In situations where insertions and deletions of data are required, arrays are often inefficient. For example, deleting the first element of an array of length 1000 requires 999 data movements! Insertions also often lead to a great deal of data movement.

3) Arrays are not always appropriate data structures for modeling "real world" situations. For example, a tree is often a good storage model for solving problems and arrays are not ideal for modeling many kinds of trees.

Linked Data Structures Provide Solutions

One good solution to the array problems given above is to use **linked data structures**, including trees and linked lists. Linked data structures have the following advantages:

- They can be built at run-time and extra storage can be added as needed – the programmer need not anticipate the amount of storage required.

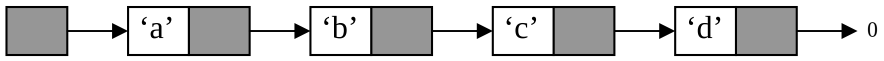
- Insertion and deletion of items into linked structures is very efficient and often much faster.

- Linked data structures are useful for modeling some real world situations.

The remainder of this handout will focus on linked lists, a versatile type of linked data structure.

A High Level View of Linked Lists

To master the notions involved when working with linked lists, you must be able to work with them at both a *high level* (abstract) and a *low level*. High level work involves drawing diagrams that facilitate the construction of algorithms. Low level work involves the (sometimes difficult) process of using C++ and writing code that implements the algorithm. Let's first take a high level view of linked lists. To represent a linked list that contains the characters 'a' through 'd', we might draw a diagram that follows below.



This diagram attempts to convey the idea that the linked list above contains the characters 'a'..'d'. Each of the rectangles is called a **node** (or cell). The node containing 'a' is called the **head node**. The node containing the 'd' is called the **tail node**. The shaded portion of the node contains a pointer variable often called the **link**. The arrow is used to indicate that the link points at the next node, i.e. to indicate that the link contains the address of the next node. The white portion of the node contains data. In the linked list above, the data consists of a single character, but the data portion can contain virtually anything.

Diagrams, such as the one above, give an abstract view of linked list which can be most helpful when one is trying to construct algorithms to manipulate linked lists. However, at some point, one needs to implement algorithms in code, so we need to look at the C++ declarations required to set up linked structures in a C++ program.

Setting Up C++ Declarations to Support Linked List

There is more than one way to do the required C++ declarations. The method below is probably the easiest.

```
struct Node      // Definition that sets up a struct named “Node”
{                // with one char field and one field containing
  char Ch;       // a pointer to another struct of type Node.
  Node* Link;
};               // semicolon needed

Node* List;      // Declares a pointer to a var of type Node.
```

Note that the above definition is a recursive, defining a *Node* to be a struct, one of whose fields points to another struct of type *Node*. We'll soon see how the type definition of *Node* above can be used to implement a linked list in C++.

Creating Linked Lists in C++

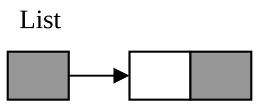
In general, the following steps are required when building a linked structure:

1) Create a new node.
2) Fill the node with appropriate data.
3) Connect the node to previously built linked structure.

The first two steps are quite easy, but the third requires care. The **new** operator is used to create (at run-time) a new node of a linked structure. Assuming the declarations above, the call "`List = new(nothrow) Node;`" does *two* things.

1) It creates a new chunk of storage of type *Node* (or in general whatever type that `List` is supposed to point at).
2) It aims `List` at this chunk of storage.

After "`List = new(nothrow) Node;`", the situation is as pictured.



C++ Naming Rules

1) Using the usual rule for dereferencing pointers, the new struct pointed at by `List` has the name, "`*List`".
2) The data portion of the new `Node` is therefore "`(*List).Ch`" and the link portion is "`(*List).Link`". The parentheses are needed because "*" has lower precedence then ".".
3) These are not good names, but C++ provides some better ones. The names, "`List->Ch`" and "`List->Link`" can be used to reference the `Ch` and `Link` fields of `*List`.

Points to Remember

- Strange or even disastrous run-time behavior will result if your code writes to memory pointed at by unitialized pointers!
- It is necessary to initialize the last pointer of a linked list, which in this case, is `List->Link`.
- The special value "`NULL`" is often used to initialize unused pointer variables. `NULL` is defined in stdio.h and has value 0. When this is true we also say that the link points at `NULL`.
- The short C++ program below creates a linked list with a single node, which contains 'a'. The tail node points at `NULL`.

```
struct Node
{
  char Ch;
  Node *Link;
};

int main()
{
  Node *List;

  List = new(nothrow) Node;    //create new node and aim List at it
  if ( List == NULL )          //error allocating memory
    return 1;

  List->Ch = 'a';              //put an 'a' in Ch field of List
  List->Link = NULL;           //Initialize link field
  return 0;
}
```

A Program to Build a Linked List with Three Nodes

The program below builds a linked list with just 3 nodes, then displays the data in the list. The code is not elegant, but provides a simple example for those trying to learn about linked list. No error check for enough memory was left out on purpose.

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

int main(void)
{
  struct Node
  {
    char Ch;
    Node *Link;
  };

  Node *List, *Last;         //List and Last are pointers to a Node

  List = new(nothrow) Node; //Create new struct and aim List at it
  List->Ch = 'a';           //Put 'a' in the first node.

  Last = List;              //We'll use Last to build rest of list

  Last->Link = new(nothrow) Node;  //Create new node at tail
  Last->Link->Ch = 'b';            //Fill in Ch field

  Last = Last->Link;               //Move Last forward one node

  Last->Link = new(nothrow) Node;  //Add the final node
  Last->Link->Ch = 'c';            //fill in Ch field

  Last->Link->Link = NULL;         //pointer constant marks end of list

  cout << "\nHere is the data in the linked list: \"";

  while (List != NULL)
  {
    cout << List->Ch;        //Display data
    List = List->Link;       //Point at next node
  }

  cout << "\"\n";

  return 0;
}
```

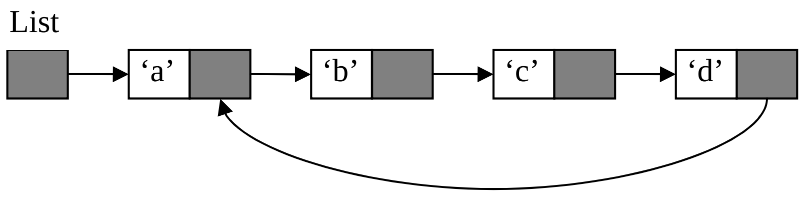
A Program to Build a Linked List with Any Number of Nodes

The program below builds a linked list holding characters typed by the user. The tail pointer of the list holds NULL.

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

int main(void)
{
  struct Node
  {
    char Ch;
    Node *Link;
  };

  Node *List = NULL,   //List is pointer to first node in the list
       *Last;          //Last points to last node of list
  char Ch;

  cout << "This program will build a linked list of characters.\n"
          "Type as many characters as you wish -- then a return.\n\n==>";

  cin.get(Ch);
  if (Ch == '\n')           //User wishes to exit immediately
    return 0;

  List = new(nothrow) Node; //Build first node as special case
  if (List == NULL)         //Unlikely, but possible memory error
    return 1;

  List->Ch = Ch;           //Store first character in node.
  List->Link = NULL;       //Make sure tail pointer is initialized
  Last = List;             //Last always points to end of list

  cin.get(Ch);
  while (Ch != '\n')
  {
    Last->Link = new(nothrow) Node;  //Create new node at tail

     if (Last->Link == NULL)          //error trying to allocate memory
       return 1;
     else
     {
       Last = Last->Link;             //Keep Last aimed at the last node
         Last->Ch = Ch;                //Store Char just read in linked list
       Last->Link = NULL;             //Don't leave unitialized pointers
     }
     cin.get(Ch);                     //Get the next char, exit if newline
  }

  cout << "\nHere is the list: \"";
  while (List != NULL)
  {
    cout << List->Ch;        //Display data
    List = List->Link;       //Point at next node
  }
  cout << "\"\n";
  return 0;  }
```

Circular Linked List

There are many variations of the linked lists. One useful variation is the **circular linked list**. A circular linked list has its tail node link pointing to the first node of the list, as shown below.

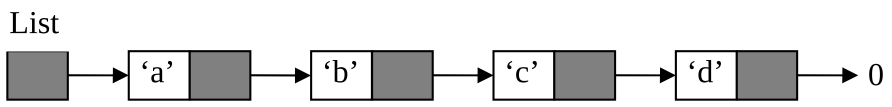


Dummy Nodes

Dummy nodes are nodes that contain no useful data, but are sometimes used to mark the head and/or the tail of a linked list. The use of dummy nodes generally leads to simpler algorithms. A **dummy head node** is simply a node at the start of a list that contains no data. The purpose of a dummy head node is to simplify algorithms. **A dummy tail node** is a node at the end of the list that contains no data.

Deleting Nodes

When using dynamic storage, it's often necessary to deallocate the storage. As an example, the code below deletes the first node of the linked list below.



```
if (List != NULL )       // Then there's something to delete
{
   Node *Temp = List;    // Need temporary pointer to aim at node

   List = List->Link;    // Aim List at second node

   delete (Temp);        // Return storage to free up memory
}
```

After the code above has been executed, the list looks like this.