Short Notes on Dynamic Memory Allocation, Pointer and Data Structure

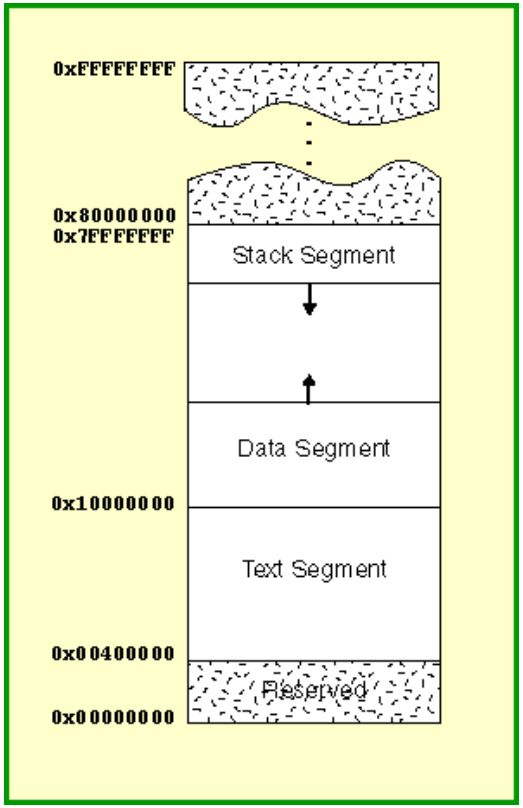
Dynamic Memory Allocation in C/C++

Motivation

```
 /*   a[100]  vs. *b or *c     */
Func(int array_size)
{
    double  k, a[100], *b, *c;
    b = (double *) malloc(array_size * sizeof(double));  /* allocation in C*/
    c = new double[array_size];  /* allocation in C++ */
}
```

- The size of the problem often can not be determined at “compile time”.
- Dynamic memory allocation is to allocate memory at “run time”.
- Dynamically allocated memory must be referred to by pointers.

Stack vs Heap



When a program is loaded into memory:

- Machine code is loaded into **text** segment
- **Stack** segment allocate memory for automatic variables within functions
- **Heap** segment is for dynamic memory allocation

Pointers

- A variable can be viewed as a specific block of memory in the computer memory which can be accessed by the identifier (the name of the variable).
  - int *k*; /* the compiler sets aside 4 bytes of memory (on a PC) to hold the value of the integer. It also sets up a symbol table. In that table it adds the symbol *k* and the relative address in memory where those 4 bytes were set aside. */
  - *k* = 8; /*at run time when this statement is executed, the value 8 will be placed in that memory location reserved for the storage of the value of k. */
- With k, there are two associated values. One is the value of the integer, 8, stored. The other is the "value" or address of the memory location.
- The variable for holding an address is a pointer variable.
  - int *ptr; /*we also give pointer a type which refers to the type of data stored at the address that we will store in the pointer*/

- ptr = &k; /* & operator retrieves the address of k */
- *ptr = 7; /* dereferencing operator "*" copies 7 to the address pointed to by ptr */

- Pointers and arrays
  - int a[100], *ptr_a;
  - ptr_a = &(a[0]); /* or ptr_a = a; */
  - ptr_a++; /*or ptr_a += 1; */ // ptr_a points to the next integer, a[1];

Memory Allocation/Free Functions in C/C++

C:

- void *malloc(size_t number_of_bytes)
  -- allocate a contiguous portion of memory
  -- it returns a pointer of type void * that is the beginning place in memory of allocated portion of size number_of_bytes.
- void free(void * ptr);
  -- A block of memory previously allocated using a call to malloc, calloc or realloc is deallocated, making it available again for further allocations.

C++:

- "new" operator
  -- pointer = new type
  -- pointer = new type [number_of_elements]
  -- It returns a pointer to the beginning of the new block of memory allocated.
- "delete" operator
  -- delete pointer;
  -- delete [] pointer;

Example 1

```
Func() /* C++ version */
{
        double *ptr;
        ptr = new double;
        *ptr = -2.5;
}
Func_C()  /* C version */
{
         double *ptr;
         ptr = (double *) malloc(sizeof(double));
         ….
}
```

- **Illustration**

Name	Type	Contents	Address
ptr	double pointer	0x3D3B38	0x22FB66

Memory heap (free storage we can use)	
…	
0x3D3B38	-2.5
0x3D3B39	

Example 2

```
Func() /* C++ version */
{
        double *ptr, a[100];
        ptr = new double[10];   /* in C, use: ptr = (double *)malloc(sizeof(double)*10); */
        for(int i = 0; i < 10; i++)
            ptr[i] = -1.0*i;
        a[0] = *ptr;
        a[1] = *(ptr+1);   a[2] = *(ptr+2);
}
```

- **Illustration**

Name	Type	Contents	Address
ptr	double array pointer	0x3D3B38	0x22FB66

Memory heap (free storage we can use)	
…	
0x3D3B38	0.0
0x3D3B39	-1.0
…	

Example 3

- Static array of dynamically allocated vectors

```
Func() /* allocate a contiguous memory which we can use for 20 ×30 matrix */
{
      double *matrix[20];
      int   i, j;
      for(i = 0; i < 20; i++)
         matrix[i] = (double *) malloc(sizeof(double)*30);

      for(i = 0; i < 20; i++)
      {
           for(j = 0; j < 30; j++)
                matrix[i][j] =  (double)rand()/RAND_MAX;
      }

}
```

Example 4

- Dynamic array of dynamically allocated vectors

```
Func() /* allocate a contiguous memory which we can use for 20 ×30 matrix */
{
      double **matrix;
      int   i, j;

      matrix = (double **) malloc(20*sizeof(double*));
      for(i = 0; i < 20; i++)
         matrix[i] = (double *) malloc(sizeof(double)*30);

      for(i = 0; i < 20; i++)
      {
           for(j = 0; j < 30; j++)
                matrix[i][j] =  (double)rand()/RAND_MAX;
      }
}
```

Example 5

- Another way to allocate dynamic array of dynamically allocated vectors

```
Func() /* allocate a contiguous memory which we can use for 20 ×30 matrix */
{
      double **matrix;
      int   i, j;

      matrix = (double **) malloc(20*sizeof(double*));
      matrix[0] = (double*)malloc(20*30*sizeof(double));

      for(i = 1; i < 20; i++)
         matrix[i] = matrix[i-1]+30;

      for(i = 0; i < 20; i++)
      {
           for(j = 0; j < 30; j++)
                matrix[i][j] =  (double)rand()/RAND_MAX;
      }
}
```

Release Dynamic Memory

```
Func()
{
    int *ptr, *p;
    ptr = new int[100];
    p = new int;
    delete[] ptr;
    delete  p;
}
```

Functions and passing arguments

- Pass by value

```
1.    #include<iostream>
2.    void foo(int);

3.    using namespace std;
4.    void foo(int y)
5.    {
6.         y = y+1;
7.         cout << "y + 1 = " << y << endl;
8.    }
9.
10.   int main()
11.   {
12.       foo(5); // first call
13.
14.       int x = 6;
15.       foo(x); // second call
16.       foo(x+1); // third call
17.
18.       return 0;
19.   }
```

When foo() is called, variable y is created, and the value of 5, 6 or 7 is copied into y. Variable y is then destroyed when foo() ends.

- Pass by address (or pointer)

```
1.  #include<iostream>
2.  void foo2(int*);
3.  using namespace std;

4.  void foo2(int *pValue)
5.  {
6.     *pValue = 6;
7.  }
8.
9.  int main()
10. {
11.    int nValue = 5;
12.
13.    cout << "nValue = " << nValue << endl;
14.    foo2(&nValue);
15.    cout << "nValue = " << nValue << endl;
16.    return 0;
17. }
```

Passing by address means passing the address of the argument variable. The function parameter must be a pointer. The function can then dereference the pointer to access or change the value being pointed to.

1. It allows us to have the function change the value of the argument.
2. Because a copy of the argument is not made, it is fast, even when used with large structures or classes.
3. Multiple values can be returned from a function.

- Pass by reference

```
1.  #include<iostream>
2.  void foo3(int&);
3.  using namespace std;

4.   void foo3(int &y) // y is now a reference
5.   {
6.      cout << "y = " << y << endl;
7.      y = 6;
8.      cout << "y = " << y << endl;
9.   } // y is destroyed here
10.
11.  int main()
12.  {
13.     int x = 5;
14.     cout << "x = " << x << endl;
15.     foo3(x);
16.     cout << "x = " << x << endl;
17.     return 0;
18.  }
```

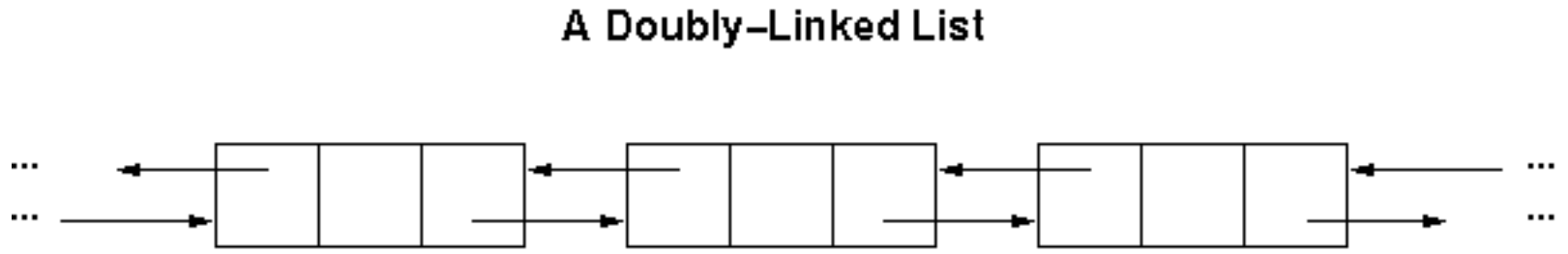
Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument.

```
1.  #include <iostream>
2.  int nFive = 5;
3.  int nSix = 6;
4.  void SetToSix(int *pTempPtr);
5.  using namespace std;
6.
7.  int main()
8.  {
9.     int *pPtr = &nFive;
10.    cout << *pPtr;
11.
12.    SetToSix(pPtr);
13.    cout << *pPtr;
14.    return 0;
15. }
16.
17. // pTempPtr copies the value of pPtr!
18. void SetToSix(int *pTempPtr)
19. {
20.     pTempPtr = &nSix;
21.
22.    cout << *pTempPtr;
23. }
```

Implementing Doubly-Linked Lists

- Overall Structure of Doubly-Linked Lists

  A list element contains the data plus pointers to the next and previous list items.

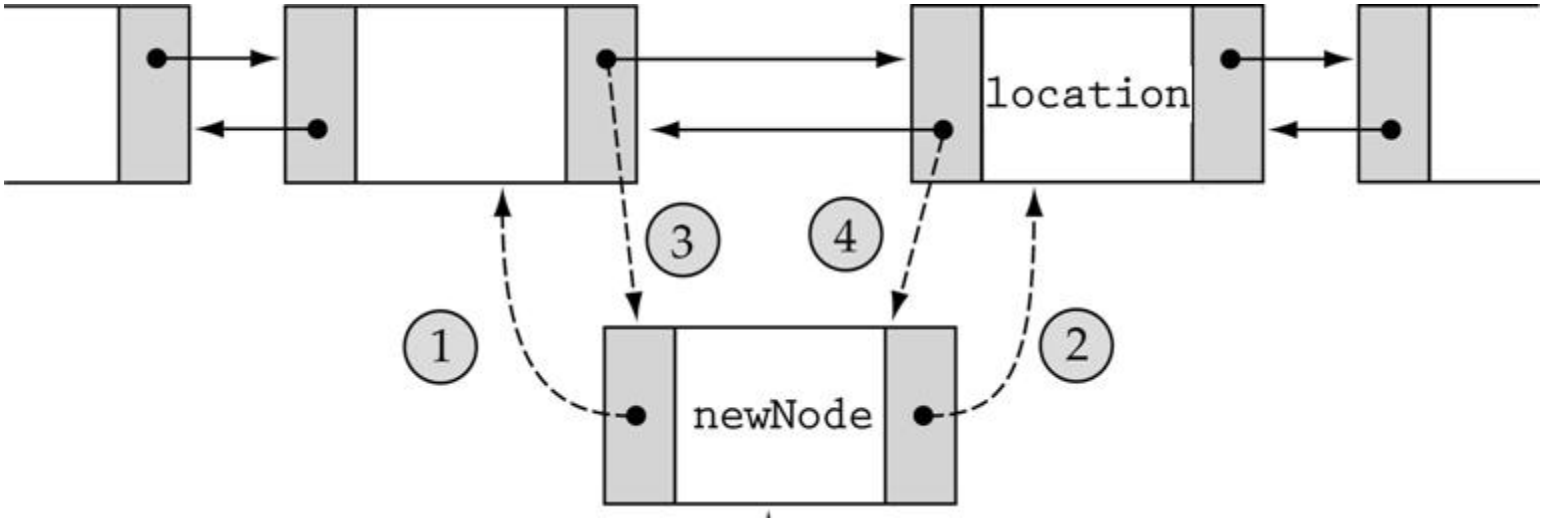


A generic doubly linked list node:

```
typedef struct node {
 int data;
 struct node* next;  // that points to the next node in the list
 struct node* prev; // that points to the previous node in the list.
} node;
```

node* head = (node*) malloc(sizeof(node));

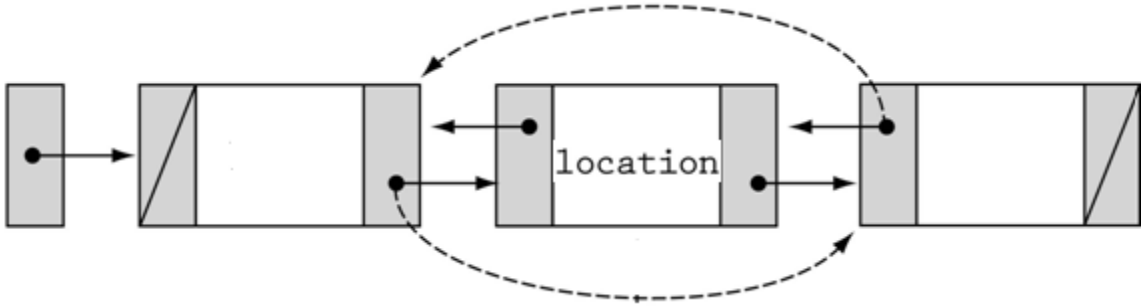
- Inserting to a Doubly Linked List



Following codes are needed:

1. newNode->prev = location->prev;
2. newNode->next = location;
3. location->prev->next=newNode;
4. location->prev = newNode;

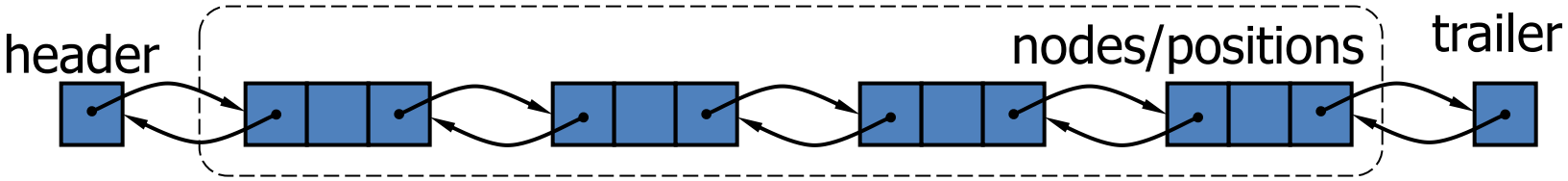
- Deleting “location” node from a Doubly Linked List



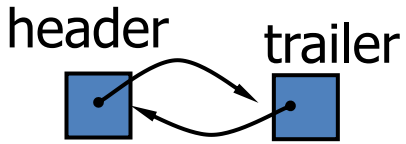
node* temp;

1. temp = location->prev;
2. temp->next =location->next;
3. (temp->next)->prev = temp;
4. free(location);

- Special trailer and header nodes and initiating doubly linked list



1. To simplify programming, two special nodes have been added at both ends of the doubly-linked list.
2. Head and tail are dummy nodes, and do not store any data elements.
3. Head: it has a null-prev reference (link).
4. Tail: it has a null-next reference (link).



Initialization:
node header, trailer;
1. header.next = &trailer;
2. trailer.prev = &header;

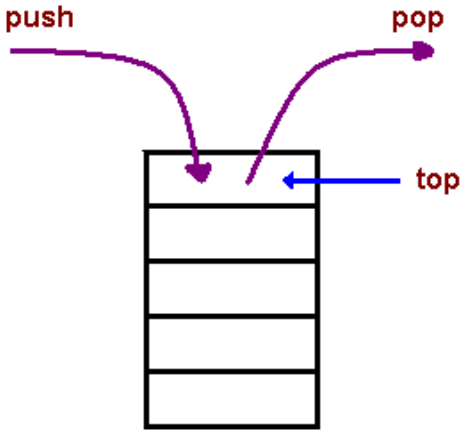
- Insertion into a Doubly-Linked List from the End

AddLast algorithm – to add a new node as the last of list:

```
addLast( node *T,  node *trailer)
{
        T->prev = trailer->prev;
        trailer->prev->next = T;
        trailer->prev = T;
        trailer->prev->next = trailer;
}
```

Stacks

- A stack is a container of objects that are inserted and removed according to the last-in first-out (**LIFO**) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack.




```
template <class T>
class stack {
    T*    v;
    T*    p;
    int    sz;

public:
   stack (int s)       {v = p = new T[sz = s];}
   ~stack()            {delete[] v;}
   void push (T a)  { *p = a;  p++;}
   T pop()             {return *--p;}
   int  size()     const {return p-v;}
};

stack <char> sc(200);    // stack of characters
```

Remark:
The template <class T> prefix specifies that a template is being declared and that an argument T of type type will be used in the declaration. After its introduction, T is used exactly like other type names. The scope of T extends to the end of the declaration that template <class T> prefixes.

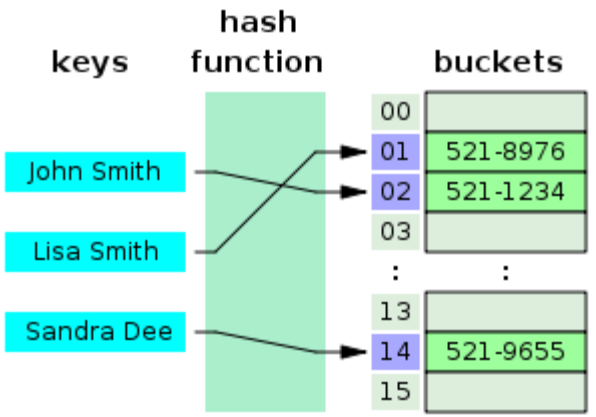
Non template version of stack of characteristics

```
class stack_char {
    char*    v;
    char*    p;
    int    sz;

public:
   stack_char (int s)       {v = p = new char[sz = s];}
   ~stack_char()            {delete[] v;}
   void push (char a)  { *p = a;  p++;}
   char pop()              {return *--p;}
   int  size()     const {return p-v;}
};

stack_char sc(200);    // stack of characters
```

Hash Table

- A hash is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.

  See also http://xlinux.nist.gov/dads/HTML/hashtab.html



Hashing: Given a key, the algorithm computes an index that suggests where the entry can be found.

index = f(key, array_size);

C++ Template

- C++ templates (or parameterized types) enable users to define a family of functions or classes that can operate on different types of information. See also http://www.cplusplus.com/doc/oldtutorial/templates/

```
// min for ints
int min( int a, int b ) {
  return ( a < b ) ? a : b;
}

// min for longs
long min( long a, long b ) {
  return ( a < b ) ? a : b;
}

// min for chars
char min( char a, char b ) {
  return ( a < b ) ? a : b;
}
```

```
//a single function template implementation
template <class T> T min( T a, T b ) {
  return ( a < b ) ? a : b;
}

int main()
{
   min<double>(2, 3.0);
}
```