Lecture 10.1

Stack data structure

Dynamic memory allocation in C

- Just like in C++ we allocate memory dynamically with the **new** operator, in C there is a function called **malloc** used to allocate memory.
- Its prototype is:

  void* malloc (size_t nbytes);

Dynamic memory allocation in C

- **malloc** returns a void pointer to the allocated buffer.
- This pointer must be *cast* (converted) into the proper type to access the data to be stored in the buffer.
- On failure, malloc returns a null pointer. The return from malloc should be tested for error as shown below.

```
char *cpt;
...
if ( ( cpt = (char *) malloc ( 25 ) ) == NULL )
{
    printf ( “Error on malloc\n” );
}
```


Remember sizeof ( )?

```
cout<< "The size of an int is “ << sizeof ( int )<<endl;
cout<< "The size of a float is “ << sizeof ( float ) <<endl;
cout<< "The size of a char is “ << sizeof ( char ) <<endl;
cout<< "The size of a double is "<< sizeof ( double ) <<endl;
```

```
The size of an int    is  4
The size of a float   is  4
The size of a char    is  1
The size of a double  is  8
```

Dynamic memory allocation in C

- To free the memory, just like using **delete** in C++, in C you call the **free** function

  void **free** (void *pt);

- It is not necessary to cast the pointer argument back to void. The compiler handles this conversion.
- Example of use:

  free cpt;

Another Example

C++ code	C code
Node * pNode = new Node;	if (**Node * pNode =** **(Node*) malloc (sizeof (Node)))** == NULL) printf (“Error on malloc”);
pNode = pHead …	pNode = pHead …
.	.
.	.
.	.
delete pNode	free (pNode);

What is a Stack?

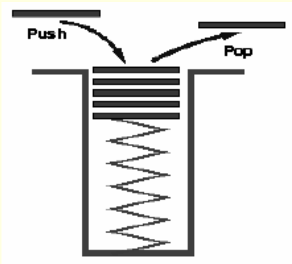
- A stack is another way to store data.
- It is generally implemented with only two principle operations.

  push // put in

  pop // take out

How does a stack work?

- A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.
- Stacks form Last- In-First- Out (LIFO) queues and have many applications.



LIFO?

- LIFO = Last In, First Out.
  The last element in is the first element out.
- With lists we can implement different orders:
  FIFO = First In, First Out.
  FILO = First In, Last Out
- Stacks have many applications from the parsing of algebraic expressions to keeping track of variables and return address values for function calls

About the stack implementation

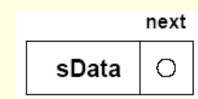
- A linked list implementation of a stack is possible
  (adding and deleting from the head of a linked list produces exactly the LIFO semantics of a stack)
- However the most common applications for stacks have a space restraint so that using an array implementation is a natural and efficient one

About dynamic allocation

- In most operating systems, allocation and de-allocation of memory is a relatively expensive operation, there is a penalty for the flexibility of linked list implementations.

Stack Implementation with linked lists

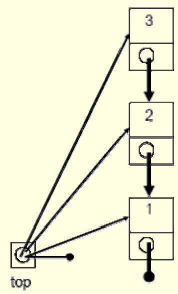
```
typedef struct stack_node{
  int sData;
  stack_node* next;
}sNode;
```



- sNode* top = NULL;

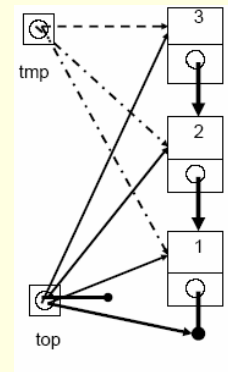
Push

```
void push ( int data ) {
   sNode* nNode = ( sNode * ) malloc ( sizeof ( sNode ) );
   nNode->sData = data;
   if ( top == NULL)
   {
        top = nNode;
        nNode- >next = NULL;
   }
   else
   {
        nNode->next = top;
        top = nNode;
   } }
```



Pop

```
bool pop ( int &data )
{
   if ( top == NULL)
        return false;
   else
   {
        sNode* tmp = top;
        data = top->sData;
        top = top->next;
        free (tmp);
        return true;
   }
}
```



Printing the Stack

```
void printStack ( ) {
   sNode* tmp = top;
   while ( tmp != NULL)
   {
        cout << "\t“ << tmp- > sData << endl;
        tmp = tmp- > next;
   }
}
```


The output

- The output will **always** be in inverse order as the input.
- If we push the numbers:
  - 1 2 3 4 5 6 7
- In that order, we will pop:
  - 7 6 5 4 3 2 1

Calling the methods

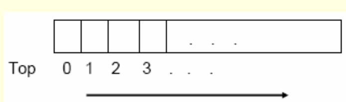
```
int main ( )
{
   int dData;
   for ( int i = 0; I <= 5; i++ )
        push ( i );
   cout << "Printing Stack \n";
   printStack ( ) ;
   for ( int i = 5; I >= 0; i-- ){
        if ( pop ( dData ) ){
                cout <<"popping :"<<dData<<endl; …..
```


Stack implementation with arrays and template classes

```
template <class Type> class Stack {
   private:
        int size;
        Type* sPtr;
        int top;
   public:
        Stack (int s ) { size = s;
                top = -1;
                sPtr = new Type [ size ] ; //allocate the array for the stack.
        }
        bool push( Type val );
        bool pop ( Type &data );
        bool isEmpty( ) { return top == -1;}
        bool isFull ( ) { return top == ( size -1 ); }
        int getIndex ( ) { return top; }
};
```

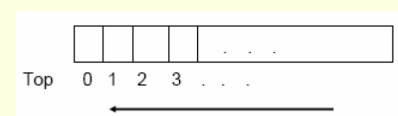

Method implementation

```
template <class Type> bool Stack<Type>::push(Type val)
{
   // Check stack is not full, increment index, then store
   if ( !isFull ( ) ){
        sPtr [ ++top ] = val;
        return true;
   }
   return false;
}
```



Method implementation (cont)

```
template <class Type> bool Stack<Type>::pop(Type &data )
{
   // Retrieve, then decrement index
   if ( ! isEmpty ( ) )
   {
        data= sPtr [ top-- ];
        return true;
   }
   return false;
}
```



Calling the methods

```
int main ( ) {
   int size;
   cout << "Please enter a size for your stack "<<endl;
   cin >> size;
   Stack <int> stack(5);
   int num;
   int val;
   for (int i = 0; i< size; i++) {
        cout << "Enter num: ";
        cin >>num;
        stack.push ( num );
   }
```


Calling the methods (cont)

```
   while ( stack. pop ( val ) ) {
        cout << "popped " << val << endl;
        cout << "index " << stack.getIndex( )<<endl;
   }

   system("PAUSE");
   return 0;
}
```