

```
#ifndef __STACK_H_
#define __STACK_H_

// Note:
// Some extra tricks employed to keep lines
// no longer than 43 characters, to nicely
// fit on OH with 18pt fontsize

#include <exception>
#include <stdexcept>

typedef int Stack_Data;

class Stack
{
public:
    Stack() : top_(0) {}
    ~Stack();

    void push(Stack_Data const& d);
    Stack_Data top() const;
    Stack_Data pop();
    int size() const;
    bool is_Empty() const;

// make private if not implemented
//private:
    Stack(Stack const&);
    Stack& operator=(Stack const&);
};
```

```

private:
    struct Stack_Node
    {
        Stack_Node(Stack_Data value,
                    Stack_Node* n = 0)
            : data(value), next(n) {}

        ~Stack_Node() { delete next; }

        Stack_Data data;
        Stack_Node* next;
    };

    void unlink(Stack_Node*& node);
    int sum_Size(Stack_Node* from) const;
    Stack_Node* clone(Stack_Node* from) const;

    Stack_Node* top_;
};

class Stack_Error : public std::exception
{
public:
    Stack_Error(std::string const& m) throw()
        : exception(), message(m) {}
    ~Stack_Error() throw() {}

    const char* what() const throw()
    {
        return message.c_str();
    }

private:
    std::string message;
};

#endif
// End of header file

```

```
#include <algorithm>

// public implementation

void Stack::push(Stack_Data const& d)
{
    top_ = new Stack_Node(d, top_);
}

Stack_Data Stack::top() const
{
    if (is_Empty())
        throw Stack_Error("empty stack");
    else
        return top_->data;
}

Stack_Data Stack::pop()
{
    Stack_Data result = top();
    unlink(top_);
    return result;
}

int Stack::size() const
{
    return sum_Size(top_);
}

bool Stack::is_Empty() const
{
    return top_ == 0;
}
```

```

// private implementation

Stack::Stack(Stack const& rhs)
{
    top_ = clone(rhs.top_);
}

Stack& Stack::operator=(Stack const& rhs)
{
    if (this != &rhs)
    {
        Stack copy(rhs);
        std::swap(copy.top_, top_);
    }
    return *this;
}

// presumes node != 0
void Stack::unlink(Stack_Node*& node)
{
    Stack_Node* victim = node;
    node = node->next;
    victim->next = 0; // stop destructor
    delete node;    // recursion here
}

```

[illegible]

```
// iterative versions
```

```
Stack::~~Stack()
```

```
{  
    while (top_ != 0)  
        unlink(top_);  
}
```

```
int Stack::sum_Size(Stack_Node* from) const
```

```
{  
    int size = 0;  
  
    for (Stack_Node* here = from; here != 0;  
         here = here->next)  
        size = size + 1;  
  
    return size;  
}
```

```
Stack::Stack_Node*
```

```
Stack::clone(Stack_Node* from) const
```

```
{  
    if (from == 0)  
        return 0;  
  
    Stack_Node* copy;  
    Stack_Node* last;  
    last = copy = new Stack_Node(from->data);  
  
    for (Stack_Node* here = from->next;  
         here != 0;  
         here = here->next)  
    {  
        last->next = new Stack_Node(here->data);  
        last = last->next;  
    }  
    return copy;  
}  
#endif
```

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    Stack s;
    int input;

    while (cin >> input)
    {
        s.push(input);
    }

    Stack ss = s;

    while ( ! s.is_Empty() )
    {
        cout << setw(2) << s.size() << " : "
              << setw(2) << s.top() << " : "
              << setw(2) << s.pop() << endl;
    }

    while ( ! ss.is_Empty() )
    {
        cout << setw(2) << ss.size() << " : "
              << setw(2) << ss.top() << " : "
              << setw(2) << ss.pop() << endl;
    }

    ss = s;

    return 0;
}
```