Chapter - 13
Simple Classes

Stack Definition

Data:

A place to store the items put on and taken from the stack. (Implemented as an array).

Obvious operations:

Push -- Add an element to the top of the stack (other elements are pushed down).
Pop -- Remove the top element from the stack (other elements are popped up).

Hidden operations:

Construction -- The creation and initialization of the stack
Destruction -- The clean up done when the stack is destroyed.

Stack Implementation as Struct

```
/********************************************************
 * Stack                                                *
 *      A set of routines to implement a simple integer *
 *      stack.                                          *
 *                                                      *
 * Procedures                                           *
 *      stack_init -- initialize the stack.             *
 *      stack_push -- put an item on the stack.         *
 *      stack_pop -- remove an item from the stack.     *
 ********************************************************/
#include <stdlib.h>
#include <iostream>


const int STACK_SIZE = 100;     // Maximum size of a stack


// The stack itself
struct stack {
   int count;                   // Number of items in stack
   int data[STACK_SIZE];        // The items themselves
};


/********************************************************
 * stack_init -- initialize the stack.                  *
 *                                                      *
 * Parameters                                           *
 *      the_stack -- stack to initialize                *
 ********************************************************/
inline void stack_init(struct stack &the_stack)
{
    the_stack.count = 0;        // Zero the stack
}
```

Stack Program (cont.)

```
/********************************************************
 * stack_push -- push an item on the stack.             *
 *                                                      *
 * Warning: We do not check for overflow.               *
 *                                                      *
 * Parameters                                           *
 *      the_stack -- stack to use for storing the item  *
 *      item -- item to put in the stack                *
 ********************************************************/
inline void stack_push(struct stack &the_stack,
                       const int item)
{
    the_stack.data[the_stack.count] = item;
    ++the_stack.count;
}
/********************************************************
 * stack_pop -- get an item off the stack.              *
 *                                                      *
 * Warning: We do not check for stack underflow.        *
 *                                                      *
 * Parameters                                           *
 *      the_stack -- stack to get the item from         *
 *                                                      *
 * Returns                                              *
 *      The top item from the stack.                    *
 ********************************************************/
inline int stack_pop(struct stack &the_stack)
{
    // Stack goes down by one
    --the_stack.count;


    // Then we return the top value
    return (the_stack.data[the_stack.count]);
}
```

Using the Stack

```
// A short routine to test the stack
main()
{
    struct stack a_stack;       // Stack we want to use


    stack_init(a_stack);


    // Push three value on the stack
    stack_push(a_stack, 1);
    stack_push(a_stack, 2);
    stack_push(a_stack, 3);


    // Pop the item from the stack
    std::cout << "Expect a 3 ->" << stack_pop(a_stack) << '\n';
    std::cout << "Expect a 2 ->" << stack_pop(a_stack) << '\n';
    std::cout << "Expect a 1 ->" << stack_pop(a_stack) << '\n';


    return (0);
}
```

Stack as a Class

```
class stack {
    private:
        int count;              // Number of items in the stack
        int data[STACK_SIZE];   // The items themselves
    public:
        // Initialize the stack
        void init(void);


        // Push an item on the stack
        void push(const int item);


        // Pop an item from the stack
        int pop(void);
};
```

Stack member functions

```
inline void stack::init(void)
{
    count = 0;  // Zero the stack
}


inline void stack::push(const int item)
{
    data[count] = item;
    ++count;
}


inline int stack::pop(void)
{
    // Stack goes down by one
    --count;


    // Then we return the top value
    return (data[count]);
}
```

Using a class

Declaring a class variable (called an instance of a class):

```
    class stack a_stack;        // Stack we want to use
```

or more commonly:

```
    stack a_stack;        // Stack we want to use
```

Calling member functions:

```
    a_stack.init();
    a_stack.push(1);
    result = a_stack.pop();
```

Constructor

A constructor is called when a variable is created.
The member function for the constructor is the same as the class’s name.

```
class stack {
        // ...
    public:
        // Initialize the stack
        stack(void);
        // ...
};


inline stack::stack(void)
{
    count = 0;  // Zero the stack
}


main()
{
    stack a_stack;      // Stack we want to use
                        // Calls stack::stack()
```

Destructor

A destructor is called when a variable is destroyed (goes out of scope). The member function for the destructor is named the same as the class with a tilde (~) in front of it.

```
stack::~stack(void) {
    if (count != 0)
        std::cerr <<
          "Error: Destroying a non-empty stack\n";
}
```

Parametrized Constructors

```
class person {
    public:
        std::string name;      // Name of the person
        std::string phone;     // His phone number
        // .....
    public:
        person(const std::string& i_name,
               const std::string& i_phone);
    // ... rest of class
};


person::person(const std::string& i_name,
               const std::string& i_phone)
{
    name = i_name;
    phone = i_phone;
}




main()
{
    person sam("Sam Jones", "555-1234");
    person sam; // Illegal
```

Overloaded Constructors

```
class person {
    public:
        std::string name;      // Name of the person
        std::string phone;     // His phone number
        // .....
    public:
        person(const std::string& i_name,
               const std::string& i_phone);
        person(const std::string& i_name);
    // ... rest of class
};
person::person(const std::string& i_name)
{
    name = i_name;
    phone = "No Phone";
}


main()
{
    person sam("Sam Jones", "555-1212");
    person john("John Smith");
    person joe;         // Illegal
```

Parameterized Destructors

No such thing.

Copy Constructor

```
stack::stack(const stack &old_stack)
{
    int i;     // Index used to copy the data

    for (i = 0; i < old_stack.count; ++i) {
        data[i] = old_stack.data[i];
    count = old_stack.count;
}


main()
{
    stack old_stack;

    old_stack.push(1);
    old_stack.push(2);

    stack new_stack(old_stack);
```

Hidden Member Function Calls

```
void use_stack(stack local_stack)
{
    local_stack.push(9);
    local_stack.push(10);
    .. Do something with local_stack
}



main()
{
    stack a_stack;     // Generate a default stack

    a_stack.push(1);
    a_stack.push(2);

    use_stack(a_stack);

    // Prints "2"
    std::cout << a_stack.pop() << '\n';
```

Automatically Generated Member Functions

```
class::class()
```

Default constructor

```
class::class(const class &old_class)
```

Copy constructor

```
class::~class()
```

Destructor

```
class class::operator = (const class &old_class)
```

Assignment operator.

Shortcuts

```
class stack {
     public:
          // .... rest of class

          // Push an item on the stack
          void push(const int item) {
              data[count] = item;
              ++count;
          }
     };
```

Class Style

- Use the "short form" only for very short functions who's purpose is obvious.
- Use the "short form" only if you can use it and keep the structure of the class clear and easy to understand.
- Remember the "big 4". These four member functions should be explicitly supplied, else include or a comment indicating that you are using the default.

Big 4:

1. Default constructor
2. Destructor
3. Copy constructor
4. Assignment operator

Style Example

```
// Comments describing the class
class queue {
    private:
        int data[100];    // Data stored in the queue
        int first;        // First element in the queue
        int last;         // Last element in the queue
    public:
        queue();          // Initialize the queue
        // queue(const queue &old_queue)
        //     Use automatically generated copy constructor


        // queue operator = (const queue &old_queue)
        //     Use automatically generated assignment operator


        // ~queue()
        //    Use automatically generated destructor


        void put(int item);// Put an item in the queue
        int get(void);    // Get an item from the queue
};
```

Classes that can't be copied

If you want a class that does not contain a copy constructor, you can’t just leave the constructor out. C++ will generate a default.

The trick is to declare the constructor **private**:

```
class no_copy {
           // Body of the class
             private:
        // There is no copy constructor
        no_copy(const no_copy &old_class);
};
```