# Introduction to Object-Oriented Programming

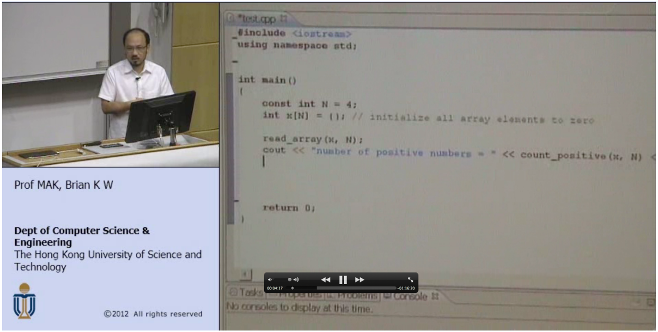
# COMP2011: C++ Class

Dr. Cindy Li
Dr. Brian Mak
Dr. Dit-Yan Yeung

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

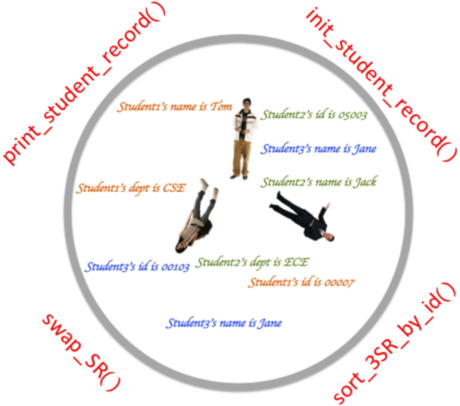
# Part I

## What is a C++ Class?



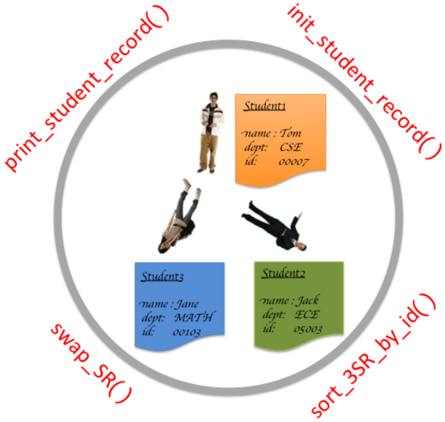
# What Happens Before We Have C++ Class?

- Pieces of information, even belonging to the same object, are scattered around.
- All functions are global and are created to work on data.



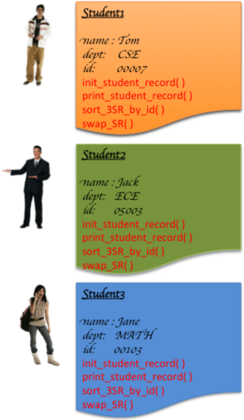
# struct Helps Organize Data Better

- Pieces of information that belong to the same object are collected and wrapped in a struct.
- All functions are still global and are created to work on structs.



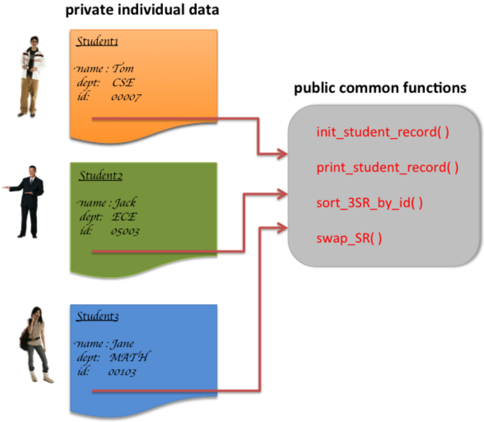
# Perhaps We May Wrap Functions into struct as Well ???

- Functions are not global anymore. However, the function codes of different objects of the same struct are the same.
- Aren't the duplicate functions a waste?



# Actual C++ Class Implementation

- Factor out the common functions so that the compiler generates only one copy of machine codes for each function.
- But functions are "struct-specific" — they can only be called by objects of the intended struct. Now you get a class!



# C++ Class

- C++ struct allows you to create new complex data type consisting of a collection of generally heterogeneous objects.
- However, a basic data type like int, besides having a value, also supports a set of operations: $+$, -, $\times$, $/$, %, $\gg$ (for input), and $\ll$ (for output).
- struct is inherited from the language C, and C++ generalizes the idea to class:

**C++ Class**

Class = data + operations + access control

or, Class = data member + member functions + access control

- Class allows you to create "smart" objects that support a set of operations. (c.f. a remote control)
- Class is also known as abstract data type (ADT).

# C++ Class ..

- data members: just like data members in a struct.
- member functions: a set of functions that work only for the objects of the class, and can only be called by them.
- In addition, C++ allows access control to each data member and member function:
  - public: accessible to any functions (both member functions of the class or other functions)
  - private: accessible only to member functions of the class
    $\Rightarrow$ enforce information hiding
  - protected

(Actually it is more complicated; we'll leave it to COMP2012.)

# Example: Simplified student_record Class Definition

```
#include <iostream>                                          /* File: student-record.h */
using namespace std;
const int MAX_NAME_LEN = 32;

class student_record
{
  private:
    char name[MAX_NAME_LEN];
    unsigned int id;

  public:
    // ACCESSOR member functions: const => won't modify data members
    const char* get_name(void) const { return name; }
    unsigned int get_id(void) const { return id; }
    void print(void) const
    { cout << "Name:\t" << name << "\nID:\t" << id << endl; }

    // MUTATOR member functions
    void set(const char* my_name, unsigned int my_id)
    { strcpy(name, my_name); id = my_id; }

    void copy(const student_record& sr) { set(sr.name, sr.id); }
};
```

# Example: student-record-test.cpp

```
#include "student-record.h"          /* File: student-record-test.cpp */

int main(void)
{
    student_record adam, brian;    // Create 2 static student_record objects

    adam.set("Adam", 12345);         // Put values to their data members
    brian.set("Brian", 34567);

    cout << adam.get_name( ) << endl; // Get and print some data member
    adam.copy(brian);                 // Adam want to fake Brian's identity
    adam.print( );

    return 0;
    // Adam and Brian are static object, which will be destroyed
    // at the end of the function — main( ) here — call.
}  /* To compile: g++ student-record-test.cpp */
```

# Part II

## OOP!








- All cars have attributes (data members) such as make, model, size, color, etc. They all are driven in a similar way (member functions) such as braking, pedaling, etc. So it is natural to create them as objects of a class called “Car”.
- In general, create a C++ class for objects of the same kind.

# Example: temperature Class Definition

```cpp
#include <iostream>                                    /* File: temperature.h */
#include <cstdlib>
using namespace std;
const char CELSIUS = 'C', FAHRENHEIT = 'F';

class temperature
{
  private:
    char scale;
    double degree;

  public:
    // CONSTRUCTOR member functions
    temperature(void);                                 // Default constructor
    temperature(double d, char s);
    // ACCESSOR member functions: don't modify data
    char get_scale(void) const;
    double get_degree(void) const;
    void print(void) const;
    // MUTATOR member functions: will modify data
    void set(double d, char s);
    void fahrenheit(void);                  // Convert to the Fahrenheit scale
    void celsius(void);                        // Convert to the Celsius scale
};
```

# Example: temperature Class Constructors

```
#include "temperature.h"                    /* File: temperature_constructors.cpp */

// CONSTRUCTOR member functions
temperature::temperature(void)                              // Default constructor
{
    degree = 0.0;
    scale = CELSIUS;
}

temperature::temperature(double d, char s)                  // A general constructor
{
    set(d, s);                                  // Calling another member function
}
```

# Example: temperature Class Accessors

```
#include "temperature.h"                    /* File: temperature_accessors.cpp */

// ACCESSOR member functions
char temperature::get_scale(void) const
{
    return scale;
}

double temperature::get_degree(void) const
{
    return degree;
}

void temperature::print(void) const
{
    cout << degree << " " << scale;
}
```

# Example: temperature Class Mutators

```cpp
#include "temperature.h"                    /* File: temperature_mutators.cpp */

void temperature::set(double d, char s)
{
    degree = d; scale = toupper(s);         // lower case -> upper case

    if (scale != CELSIUS && scale != FAHRENHEIT) {
        cout << "Bad temperature scale:  " << scale << endl; exit(-1);
    }
}

void temperature::fahrenheit(void)          // Conversion to the Fahrenheit scale
{
    if (scale == CELSIUS) {
        degree = degree*9.0/5.0 + 32.0; scale = FAHRENHEIT;
    }
}

void temperature::celsius(void)             // Conversion to the Celsius scale
{
    if (scale == FAHRENHEIT) {
        degree = (degree - 32.0)*5.0/9.0; scale = CELSIUS;
    }
}
```

# Example: Testing the temperature Class

```
#include "temperature.h"                       /* File: temperature_test.cpp */

int main(void)
{
    char scale;
    double degree;

    temperature x;                                   // Use default constructor
    x.print( ); cout << endl;                        // Check the default values

    cout << "Enter temperature (e.g., 98.6 F): ";
    while (cin >> degree >> scale)
    {
        x.set(degree, scale);
        x.fahrenheit( ); x.print( ); cout << endl;   // Convert to Fahrenheit format
        x.celsius( ); x.print( ); cout << endl;      // Convert to Celsius format

        cout << endl << "Enter temperature (e.g., 98.6 F): "; // Next input
    };

    return 0;
}
```

# Constructors and Destructors

- An object is constructed when it is
  - defined in a scope (static object on stack).
  - created with the new operator (dynamic object on heap).
- An object is destructed when
  - it goes out of a scope (static object).
  - it is deleted with the delete operator (dynamic object).
- For "objects" (actually they are not objects in C++) of basic data types, their construction and destruction are built into the C++ language.
- For (real) objects of user-defined classes, C++ allows the class developers to write their own construction and destruction functions: constructors and destructor.
- Besides creating an object, a constructor may also initialize its contents.
- A class may have more than 1 constructor (function overloading), but it can only have 1 and only 1 destructor.

# Default Constructors and Destructors

- If you do not provide a constructor/destructor, C++ will automatically generate the default constructor/destructor for you.
- For a class that does not contain dynamic data members, its default constructors and destructors are very simple:
  - the default constructor just reserves an amount of memory big enough for an object of the class; no initialization will take place.
  - the default destructor just releases the memory acquired by the object.

```
temperature::temperature(void) { }
temperature::~temperature(void) { }
```

- However, for a class that contains dynamic data members, the default constructors and destructors are usually inadequate, as they will not create and delete the dynamic data members for you.

# Contents and Interface

- The data members are the contents of the objects of a class.
  - Usually they are made private.
  - Different objects of a class usually have different contents — different values for their data members.
- The member functions represent the interface to the objects of a class.
  - Usually they are made public.
  - private member functions are for internal use only.
  - Different objects of a class have the same interface!
- Both data members and member functions are members of a class. They are uniformly accessed by the . operator.
- An application programmer should
  - only use the public interface provided by the class developer to manipulate objects of a class.
  - a good class design will prevent the application programmer from accessing and modifying the data members directly.

# Class Member Functions

- There are at least 4 types of member functions:

| | |
|---|---|
| constructor | : used to create class object. |
| destructor | : used to destruct class objects. It is needed only when objects contain dynamic data member(s). |
| accessor | : const functions that inspect data members; they do not modify any data members though. |
| mutator | : will modify some data member(s). |

- The member functions may be defined
  - inside the class definition in a .h header file.
  - outside the class definition in a .cpp source file. In that case, each function name must be prepended with the class name and the special class scope operator ::.
    ⇒ This is preferred so that application programmers won't see the implementation of the functions, and they are only given the .o object file of the class for development.
    ⇒ information hiding; protecting intellectual property.

# To Enforce Information Hiding In Practice

- Developer of the class named "myclass" will only provide
  1. myclass.h: class definition header file
  2. libmyclass.a: library file that contains the object codes of the implementation of all class member functions (usually written in several source files)
- Using the temperature class as an example, one may build its library named "libtemperature.a" on Linux as follows:

```
g++ -c temperature_constructors.cpp
g++ -c temperature_accessors.cpp
g++ -c temperature_mutators.cpp
ar rsuv libtemperature.a temperature_constructors.o \
        temperature_accessors.o temperature_mutators.o
```

- An application programmer compiles an app named "temperature_test" with the source file "temperature_test.cpp" as follows:

```
g++ -o temperature_test temperature_test.cpp -L. -ltemperature
(or, g++ -o temperature_test temperature_test.cpp libtemperature.a)
```

# Information Hiding Rules

1. DON'T expose data items in a class.
   ⇒ make all data members private.
   ⇒ class developer should maintain integrity of data members.
2. DON'T expose the difference between stored data and derived data.
   ⇒ the value that an accessor member function returns may NOT be the value of any data member. It is NOT necessary to have an accessor member function for each data member.
3. DON'T expose a class' internal structure.
   ⇒ application programmers should NOT assume the data structure used for data members.
   ⇒ class developer may change representation of data members without affecting the application programmers' codes.
4. DON'T expose the implementation details of a class.
   ⇒ class developer may change algorithm of member functions without affecting the application programmers' codes.

# Example: Better Design of class temperature

- The last design of the class temperature changes its internal data, {degree, scale}, when the user wants the temperature in different scales.
- Thus, the information hiding rules 2 and 3 are not strongly followed.
- In a better design, the user does not need to know what the inside representation of {degree, scale} is when he wants to get the temperature in various scales.
- Let's re-design the temperature class to follow the last 4 rules of information hiding more closely.

# Example: temperature Class Definition (2)

```
#include <iostream>                                        /* File: temperature.h */
#include <cstdlib>
using namespace std;
const char KELVIN = 'K', CELSIUS = 'C', FAHRENHEIT = 'F';

class temperature
{
  private:
    double degree;           // Internally it is always saved in the Kelvin scale

  public:
    // CONSTRUCTOR member functions
    temperature(void);                                  // Default constructor
    temperature(double d, char s);

    // ACCESSOR member functions: don't modify data
    double kelvin(void) const;      // Read the temperature in the Kelvin scale
    double celsius(void) const;  // Read the temperature in the Fahrenheit scale
    double fahrenheit(void) const; // Read the temperature in the Celsius scale

    // MUTATOR member functions: will modify data
    void set(double d, char s);
};
```

# Example: temperature Class Constructors & Accessors (2)

```
#include "temperature.h"        /* File: temperature_constructors_accessors.cpp */

// CONSTRUCTOR member functions
temperature::temperature(void) { degree = 0.0; }
temperature::temperature(double d, char s) { set(d, s); }

// ACCESSOR member functions
double temperature::kelvin(void) const { return degree; }
double temperature::celsius(void) const { return degree - 273.15; }
double temperature::fahrenheit(void) const { return (degree - 273.15)*9.0/5.0 +
32.0; }
```

# Example: temperature Class Mutators (2)

```
#include "temperature.h"                /* File: temperature_mutators.cpp */

void temperature::set(double d, char s)
{
    switch (s)
    {
        case KELVIN: degree = d; break;

        case CELSIUS: degree = d + 273.15; break;

        case FAHRENHEIT: degree = (d - 32.0)*5.0/9.0 + 273.15; break;

        default: cerr << "Bad temperature scale:  " << s << endl; exit(-1);
    }

    if (degree < 0.0)                          // Check for integrity of data
    {
        cerr << "Temperature less than absolute zero!" << endl;
        exit(-2);
    }
}
```

# Example: Testing the temperature Class (2)

```
#include "temperature.h"                                /* File: temperature_test.cpp */

int main(void)
{
    char scale;
    double degree;
    temperature x;                                      // Use the default constructor

    cout << "Enter temperature (e.g., 98.6 F): ";
    while (cin >> degree >> scale)
    {
        x.set(degree, scale);
        cout << x.kelvin( ) << " K" << endl;            // Print in Kelvin format
        cout << x.celsius( ) << " C" << endl;           // Print in Celsius format
        cout << x.fahrenheit( ) << " F" << endl;        // Print in Fahrenheit format

        cout << endl << "Enter temperature (e.g., 98.6 F): "; // Next input
    };

    return 0;
}
```

```
#include <iostream>                                   /* File: book.h */
using namespace std;
class Book                    // Class definition written by class developer
{
  private:
    char* title;
    char* author;
    int num_pages;

  public:
    Book(int n = 100) { title = author = NULL; num_pages = n; }
    Book(const char* t, const char* a, int n = 5) { set(t, a, n); }
    ~Book(void)
    {
        cout << "Delete the book titled \"" << title << "\"" << endl;
        delete [ ] title; delete [ ] author;
    }

    void set(const char* t, const char* a, int n)
    {
        title = new char [strlen(t)+1]; strcpy(title, t);
        author = new char [strlen(a)+1]; strcpy(author, a);
        num_pages = n;
    }
};
```

```
#include "book.h"
/* File: book.cpp; an app written by an application programmer */
void make_books(void)
{
    Book y("Love", "HKUST", 88);
    Book* p = new Book [3];
    p[0].set("book1", "author1", 1);
    p[1].set("book2", "author2", 2);
    p[2].set("book3", "author3", 3);

    delete [ ] p; cout << endl;
    return;
}

int main(void)
{
    Book x("War", "Hitler", 1000);
    Book* z = new Book("Outliers", "Gladwell", 300);

    make_books( ); cout << endl;
    delete z; cout << endl;
    z = NULL;
    return 0;
}
```

# Example: Class with Dynamic Data Members — Output

Delete the book titled "book3"

Delete the book titled "book2"

Delete the book titled "book1"

Delete the book titled "Love"

Delete the book titled "Outliers"

Delete the book titled "War"

- Delete an array of user-defined objects using delete [ ].
- delete [ ] will call the class destructor on each array element in reverse order: the last element first till the first one.
- Notice also how the destructor of a static book is automatically called when it goes out of scope (e.g., when a function returns).

# Part III

## Separation of the Programming Interface from the Actual Code Implementation

# Separation of Interface and Implementation

- If
  - the class developers follow the information hiding rules in designing their classes, and
  - the application programmers do not assume/guess their internal data representation and their implementation, and only rely on their interface

  the class developers may modify the class implementation later if they find a better way without affecting the application programmers' code.
- In the following example, a class developer produces a class called "mystring" using a linked list. A programmer is only given 2 files:
  - "mystring.h": header file containing its interface
  - "libmystring.a": its library file for a particular OS/machine

  And an application programmer writes the program "mystring_test" with the code in "mystring_test.cpp".

# Example: mystring Class Definition Using Linked List

```
#include "ll_cnode.h"                                  /* File: mystring.h */

class mystring
{
  private:
    ll_cnode* head;

  public:
    // CONSTRUCTOR member functions
    mystring(void);              // Default constructor from an emtry string
    mystring(char);                      // Construct from a single char
    mystring(const char[ ]);               // Construct from a C-string

    // DESTRUCTOR member function
    ~mystring(void);

    // ACCESSOR member functions: declared const
    int length(void) const;
    void print(void) const;

    // MUTATOR member functions
    void insert(char c, unsigned n);           // Insert char c at position n
    void remove(char c);             // Delete the first occurrence of char c
};
```

```
#include "mystring.h"                                    /* File: mystring_test.cpp */

int main(void)
{
    mystring s1, s2('A'), s3("met");
    cout << "length of s1 = " << s1.length( ) << endl;
    cout << "length of s2 = " << s2.length( ) << endl;
    cout << "length of s3 = " << s3.length( ) << endl;

    cout << endl << "After inserting 'a' at position 2 to s3" << endl;
    s3.insert('a', 2); s3.print( );
    cout << endl << "After removing 'e' from s3" << endl;
    s3.remove('e'); s3.print( );
    cout << endl << "After removing 'm' from s3" << endl;
    s3.remove('m'); s3.print( );
    cout << endl << "After inserting 'e' at position 9 to s3" << endl;
    s3.insert('e', 9); s3.print( );
    cout << endl << "After removing 't' from s3" << endl;
    s3.remove('t'); s3.print( );
    cout << endl << "After removing 'e' from s3" << endl;
    s3.remove('e'); s3.print( );
    cout << endl << "After removing 'a' from s3" << endl;
    s3.remove('a'); s3.print( );

    cout << endl << "After removing 'z' from s3" << endl;
    s3.remove('z'); s3.print( );
    cout << endl << "After inserting 'h' at position 9 to s3" << endl;
    s3.insert('h', 9); s3.print( );
    cout << endl << "After inserting 'o' at position 0 to s3" << endl;
    s3.insert('o', 0); s3.print( );
    return 0;
}
```

# Example: Linked-list Char Node Class Definition

```
#include <iostream>                                    /* File: ll_cnode.h */
using namespace std;
const char NULL_CHAR = '\0';

class ll_cnode
{
  public:
    char data;                                          // Single character node
    ll_cnode* next;                     // The link to the next character node

   // CONSTRUCTOR
    ll_cnode(char c = NULL_CHAR) { data = c; next = NULL; }
};
```

# Example: mystring Class Constructors Using Linked List

```
#include "mystring.h"                    /* File: mystring_constructors.cpp */

mystring::mystring(void) { head = NULL; }   // Default constructor

mystring::mystring(char c) { head = new ll_cnode(c); }

mystring::mystring(const char s[ ])
{
    if (s[0] == NULL_CHAR)          // Empty linked list due to empty C string
    {
        head = NULL;
        return;
    }

    // First copy s[0] to the first node of mystring
    ll_cnode* p = head = new ll_cnode(s[0]);

    // Add a new ll_cnode for each char in the char array s[ ]
    for (int j = 1; s[j] != NULL_CHAR; j++, p = p->next)
        p->next = new ll_cnode(s[j]);
    p->next = NULL;                 // Set the last ll_cnode to point to NOTHING
}
```

# Example: mystring Class Accessor Functions Using Linked List

```
#include "mystring.h"                    /* File: mystring_accessors.cpp */

int mystring::length(void) const
{
    int length = 0;
    for (const ll_cnode* p = head; p != NULL; p = p->next)
        length++;
    return length;
}


void mystring::print(void) const
{
    for (const ll_cnode* p = head; p != NULL; p = p->next)
        cout << p->data;
    cout << endl;
}
```

```
#include "mystring.h"                          /* File: mystring_insert.cpp */
// To insert character c to the linked list so that after insertion,
// c is the n-th character (counted from zero) in the list.
// If n > current length, append to the end of the list.
void mystring::insert(char c, unsigned n)
{                       // STEP 1: Create the new ll_cnode to contain char c
    ll_cnode* new_cnode = new ll_cnode(c);
    if (n == 0 || head == NULL)        // Special case: insert at the beginning
    {
        new_cnode->next = head;
        head = new_cnode;
        return;
    }

    // STEP 2: Find the node after which the new node is to be added
    ll_cnode* p = head;
    for (int position = 0;
         position < n-1 && p->next != NULL;
         p = p->next, ++position)
         ;
    // STEP 3,4: Insert the new node between the found node and the next node
    new_cnode->next = p->next;                              // STEP 3
    p->next = new_cnode;                                    // STEP 4
}
```

# Example: mystring Class Mutator Functions — remove( )

```
#include "mystring.h"                              /* File: mystring_remove.cpp */
// To remove the character c from the linked list.
// Do nothing if the character cannot be found.
void mystring::remove(char c)
{
    ll_cnode* previous = NULL;                     // Point to previous ll_cnode
    ll_cnode* current = head;                      // Point to current ll_cnode
    // STEP 1: Find the item to be removed
    while (current != NULL && current->data != c)
    {
        previous = current;                        // Advance both pointers
        current = current->next;
    }

    if (current != NULL)                           // Data is found
    {                                              // STEP 2: Bypass the found item
        if (current == head)                       // Special case: Remove the first item
            head = head->next;
        else
            previous->next = current->next;
        // STEP 3: Free up the memory of the removed item
        delete current;
    }
}
```

# Example: mystring Class Destructors Using Linked List

```
#include "mystring.h"                          /* File: mystring_destructor.cpp */

mystring::~mystring(void)
{
    if (head == NULL)     // No need to do destruction for an empty mystring
        return;

    ll_cnode* current;                                 // Point to current ll_cnode
    ll_cnode* next;                                       // Point to next ll_cnode

    // Go through the linked list and delete one node at a time
    for (current = head; current != NULL; current = next)
    {
        next = current->next;
        delete current;                    // Free up the memory of each ll_cnode
    }
}
```

# Change Internal Representation of Class mystring to Array

- The last design of the class mystring uses a linked-list of characters to represent a character string.
- The class developer later decides to change the representation to a character array. As a consequence, he also has to change the implementation of all the member functions.
- Thanks to the OOP approach, the class developer can do that without changing the public interface of the class mystring. As a consequence,
  - class developer has to give the new class definition header file,
  - and the new library file to the application programmer,
  - and the application programmer does not need to change their programs, but only re-compiles his programs with the new library.

# Example: mystring Class Definition Using Array

```
#include <iostream>                                    /* File: mystring.h */
#include <cstdlib>
using namespace std;
const int MAX_STR_LEN = 1024;
const char NULL_CHAR = '\0';

class mystring
{
  private:
    char data[MAX_STR_LEN+1];
  public:
    // CONSTRUCTOR member functions
    mystring(void);                           // Construct an emtry string
    mystring(char);                         // Construct from a single char
    mystring(const char[ ]);                  // Construct from a C-string
    // DESTRUCTOR member function
    ~mystring(void);
    // ACCESSOR member functions: Again declared const
    int length(void) const;
    void print(void) const;
    // MUTATOR member functions
    void insert(char c, unsigned n);         // Insert char c at position n
    void remove(char c);            // Delete the first occurrence of char c
};
```

# Example: mystring Constructors, Destructor, Accessors Using Array

```
#include "mystring.h"    /* File: mystring_constructors_destructor_accessors.cpp */

// Constructors
mystring::mystring(void) { data[0] = NULL_CHAR; }
mystring::mystring(char c) { data[0]=c; data[1]=NULL_CHAR; }
mystring::mystring(const char s[ ])
{
    if (strlen(s) > MAX_STR_LEN)
    {
        cerr << "mystring::mystring --- Only a max.  of "
             << MAX_STR_LEN << " characters are allowed!" << endl;
        exit(1);
    }
    strcpy(data, s);
}

// Destructor
mystring::~mystring(void) { }

// ACCESSOR member functions
int mystring::length(void) const { return strlen(data); }
void mystring::print(void) const { cout << data << endl; }
```

# Example: mystring Class Mutator — insert( ) Using Array

```
#include "mystring.h"                    /* File: mystring_insert.cpp */

void mystring::insert(char c, unsigned n)
{
    int length = strlen(data);
    if (length == MAX_STR_LEN)
    {
        cerr << "mystring::insert --- string is already full!" << endl;
        exit(1);
    }

    int insert_position = (n >= length) ? length : n;

    for (int j = length; j != insert_position; j--)
        data[j] = data[j-1];

    data[insert_position] = c;
    data[length+1] = NULL_CHAR;
}
```

# Example: mystring Class Mutator — remove( ) Using Array

```
#include "mystring.h"                    /* File: mystring_remove.cpp */

void mystring::remove(char c)
{
    int j;
    int mystring_length = length( );

    for (j = 0; j < mystring_length; j++)
        if (data[j] == c)
            break;

    if (j < mystring_length)                          // c is found
        for (; j < mystring_length; j++)
            data[j] = data[j+1];
}
```