Programming with C++

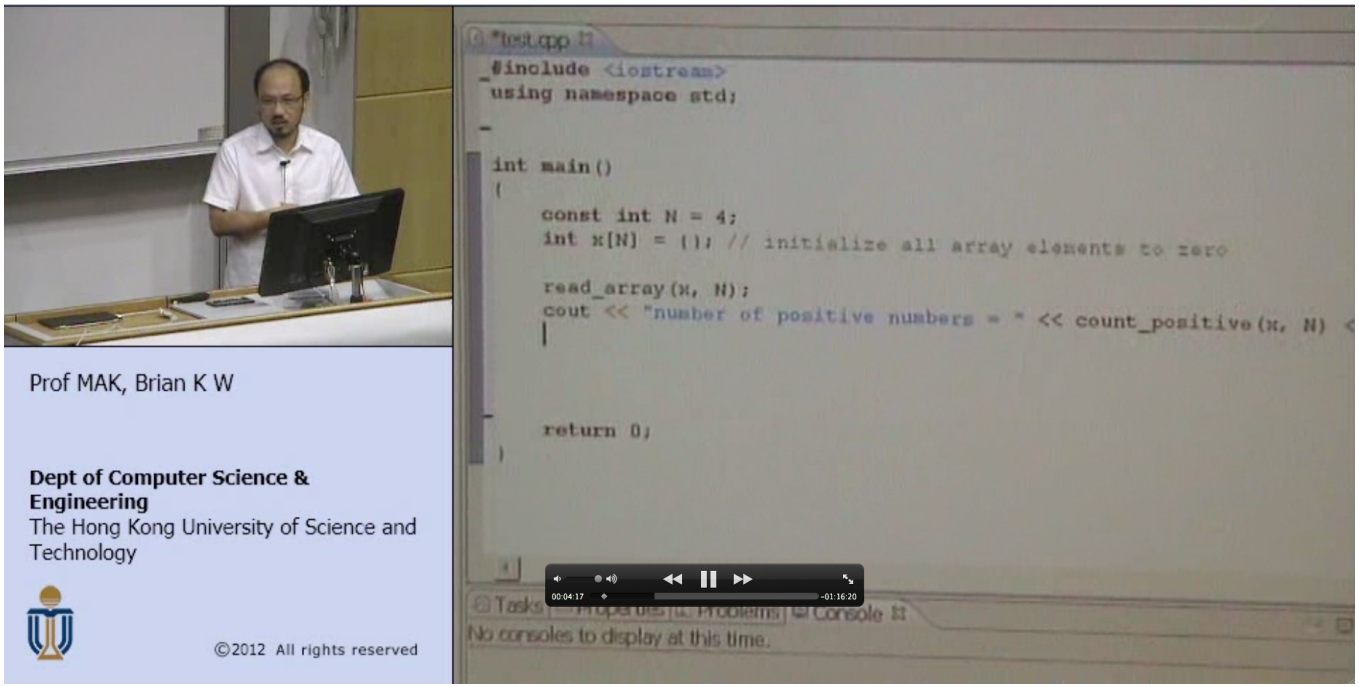
COMP2011: C++ Class

Cecia Chan
Cindy Li

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

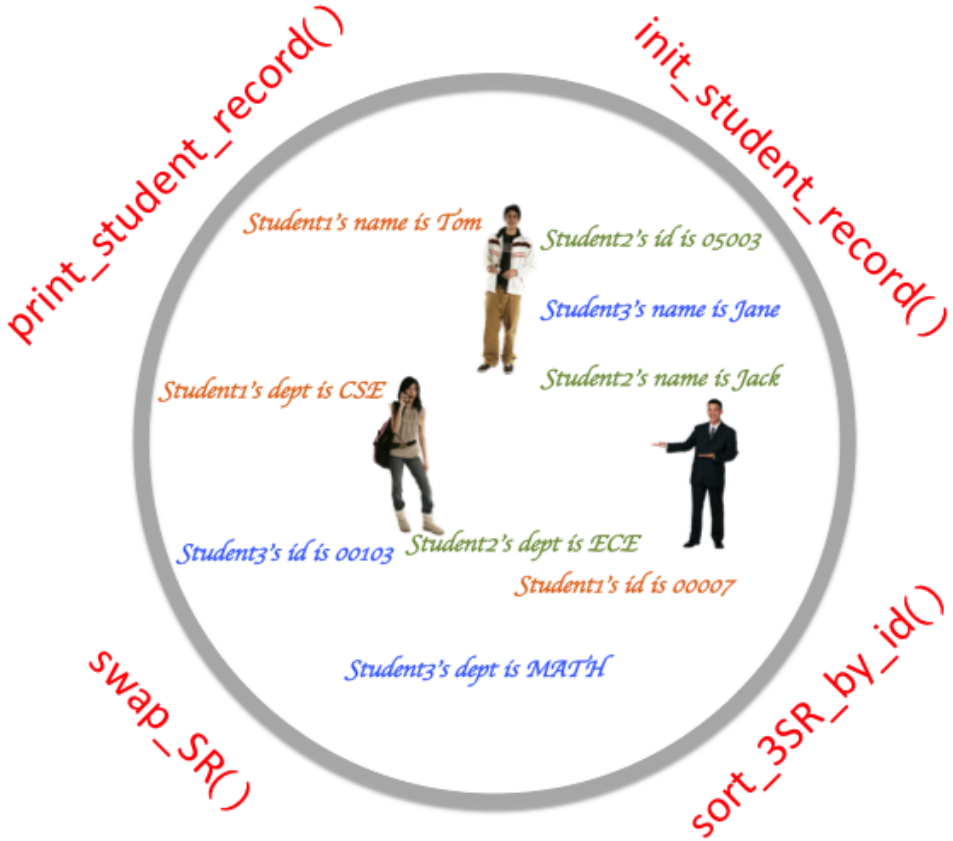
Part I

What is a C++ Class?



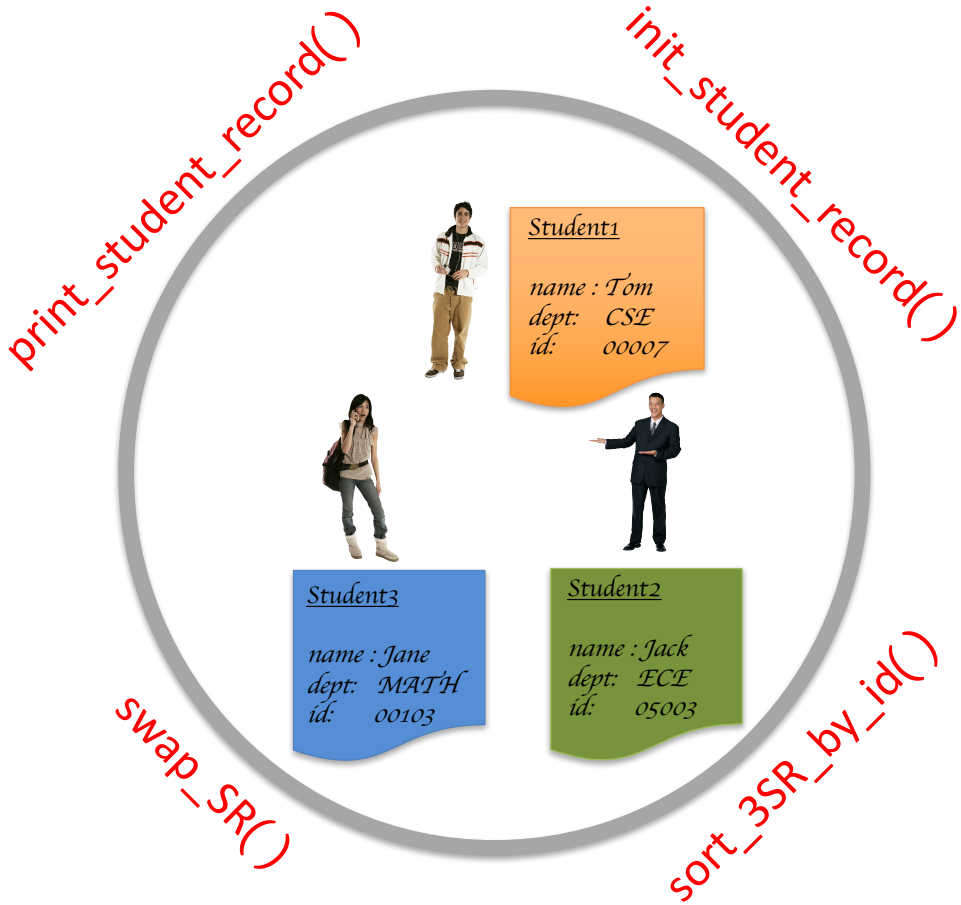
What Happens Before We Have C++ Class?

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



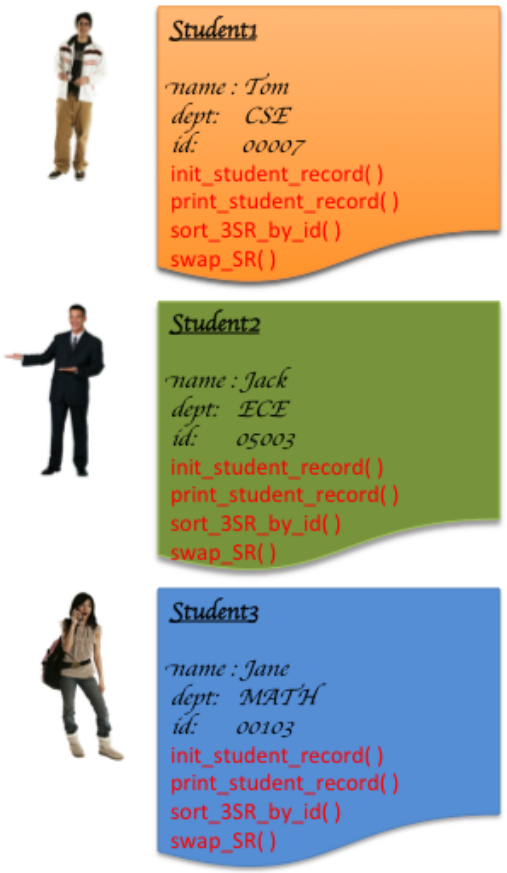
struct Helps Organize Data Better

- 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.



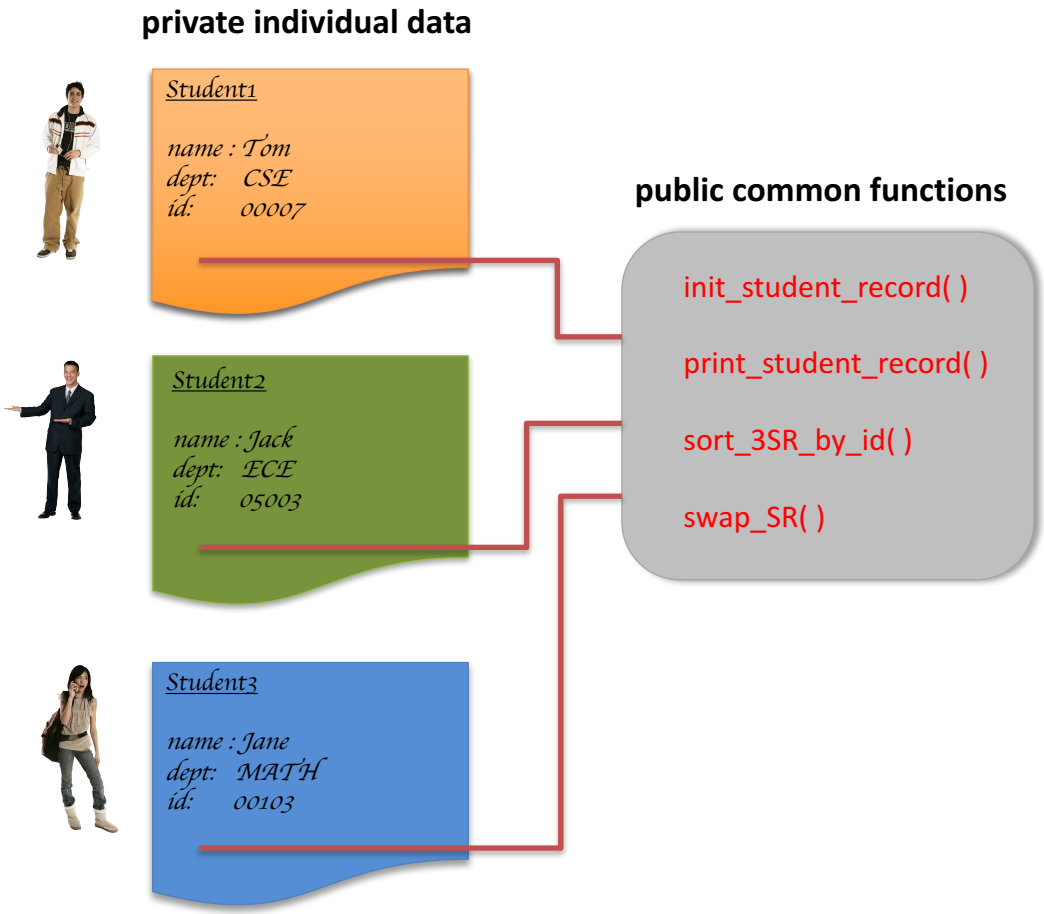
Perhaps We May Wrap Functions into struct as Well ???

- 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.)

Analogy: A Remote Control

- The internal electronic components are like data members.
- Each button is like a member function, which when pressed (c.f. calling a function), will execute a function of the remote control. e.g., next channel, volume up or down, etc.

Example: Simplified student_record Class Definition

```
const int MAX_NAME_LEN = 32;

class student_record    /* File: student-record.h */
{
  private:
    char gender;
    unsigned int id;
    char name[MAX_NAME_LEN];

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

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

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

Example: student-record-test.cpp

```
#include <iostream>     /* File: student-record-test.cpp */
using namespace std;
#include "student-record.h"

int main()
{
    student_record amy, bob; // Create 2 static student_record objects

    amy.set("Amy", 12345, 'F');   // Put values to their data members
    bob.set("Bob", 34567, 'M');

    cout << amy.get_id() << endl; // Get and print some data member
    amy.copy(bob);              // Amy wants to fake with Bob's identity
    amy.print();

    return 0;
    // Amy and Bob 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!

Manipulate Similar Objects by a C++ Class

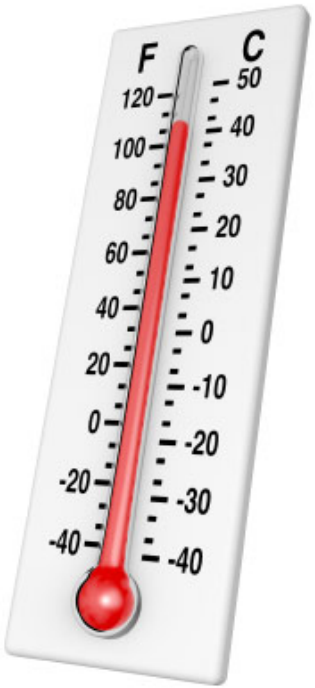







- All cars have attributes (data members) such as make, model, size, color, etc. They all are driven in a similar way (member functions) such as braking, pedaling, etc. So it is natural to create them as objects of a class called 'Car'.
- In general, create a C++ class for objects of the same kind.

Example: A C++ Class for Temperature



Example: temperature Class Definition

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

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

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

Example: temperature Class Constructors & Accessors

```
/* File: temperature-constructors-accessors.cpp */
#include "temperature.h"

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

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

Example: temperature Class Mutators

```
#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

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

int main()
{
    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
        cout << x.celsius() << " C" << endl;    // Print in Celsius
        cout << x.fahrenheit() << " F" << endl; // Print in Fahrenheit

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

    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" of basic data types (actually they are not objects in C++), 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.
- The default constructor just reserves an amount of memory big enough for an object of the class.
  - no initialization will take place (except for those with default values in C++).
- The default destructor just releases the memory acquired by the object.

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

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

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.

OOP to Maintain Data/State Consistency/Integrity

- Data/State Consistency: Each time we change the value of degree in a temperature object, make sure that the new value is valid, since not all temperature values are possible.
- For a bigger object with many data members, changing the value of one member may affect other members.
- A snapshot of the values of all data members of an object represents the state of the object.
- Applications could easily set invalid or nonsensical values, causing a system to crash. OOP forces an application to use the API that performs sanity checks on all values to maintain consistent states of objects.
- Ensuring data/state consistency is one of the major challenges in (large) software projects.
- The problem becomes even more difficult when the program is modified and new constraints are added.

Problem with non-OOP Implementation of Temperature

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

struct temperature
{
    char scale;
    double degree;
};

int main()
{
    temperature x;
    x.scale = CELSIUS;
    x.degree = -1000;    // That is IMPOSSIBLE!!!!
                         // But how can you prevent this to happen?
    return 0;
};
```

Summary: Classes and Objects

- A class is a user-defined type representing a set of objects with the same structure and behavior.
- Objects are variables of a class type.
- Instantiation: The process of creating an object of a class is called instantiating an object.
- Each object of a class has its own copies and values of its data members.
- All objects of a class share a common set of member functions.
- To call a function, before we say

  "call function X" or simply "call X".
- In OOP, we have to say

  "invoke method/operation/function X on object Y of class Z".

Summary: OOP Needs Good Design

```
struct temperature
{
    char scale;
    double degree;
};

/************** BASICALLY NO DIFFERENCE BETWEEN ***************/
/********** THE STRUCT AND CLASS DEFINITIONS HERE *************/

class temperature
{
  public:
    char get_scale() { return scale; }
    double get_degree() { return degree; }
    void set_scale(char s) { scale = s; }
    void set_degree(double d) { degree = d; }

  private:
    char scale;
    double degree;
};
```

Part III

Inline Class Member Functions; Classes With Dynamic Data; Interaction Between Classes

Class Member Functions

- These are the functions declared inside the body of a class.
- They can be defined in two ways:

1. Within the class body, then they are inline functions. The keyword inline is optional in this case.

```
class temperature
{   ...
    double kelvin() const { return degree; }
    double celsius() const { return degree - 273.15; }
};
```

Or,

```
class temperature
{   ...
    inline double kelvin() const { return degree; }
    inline double celsius() const { return degree - 273.15; }
};
```

Class Member Functions ..

2. Outside the class body, then add the prefix consisting of the class name and the class scope operator ::
   (Any benefits of doing this?)

```
/* File: temperature.h */
class temperature
{   ...
    double kelvin() const ;
    double celsius() const;
};

/* File: temperature.cpp */
#include "temperature.h"
...
double temperature::kelvin() const { return degree; }
double temperature::celsius() const { return degree - 273.15; }
```

Inline Functions

- Function calls are expensive because when a function is called, the operating system has to do a lot of things behind the scene to make that happens.

```
int f(int x) { return 4*x*x + 9*x + 1; }
int main() { int y = f(5); }
```

- For small functions that are called frequently, it is actually more efficient to unfold the function codes at the expense of program size (both source file and executable).

```
int main() { int y = 4*5*5 + 9*5 + 1; }
```

Inline Functions ..

- But functions have the benefit of easy reading, easy maintenance, and type checking by the compiler.
- You have the benefits of both by declaring the function inline.

```
inline int f(int x) { return 4*x*x + 9*x + 1; }
int main() { int y = f(5); }
```

- *However*, C++ compilers may not honor your inline declaration.
- The inline declaration is just a hint to the compiler which still has the freedom to choose whether to inline your function or not, especially when it is large!

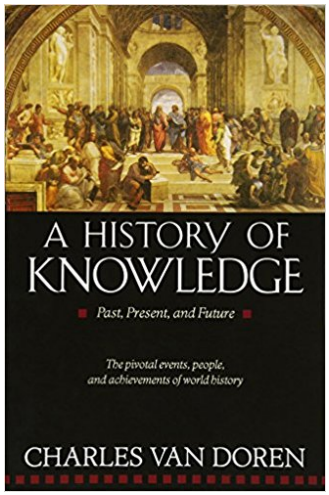
Inline Class Member Functions

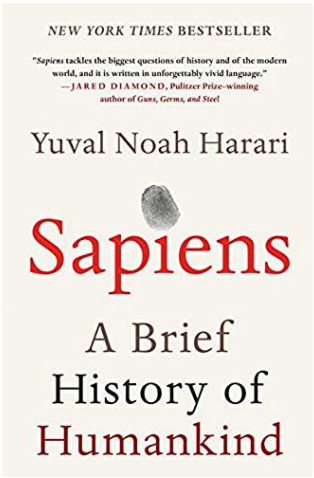
- Class member functions defined inside the class definition body are automatically treated as inline functions.
- To enhance readability, one may also define them outside the class definition but in the same header file.

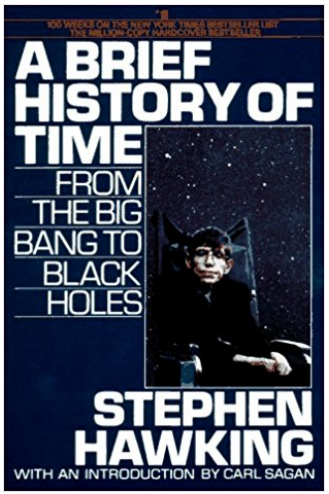
```
/* File: temperature1.h */
class temperature
{   ...
    inline double
    kelvin() const
    {
        return degree;
    }
};
```

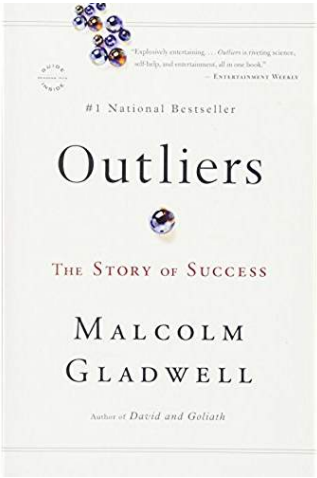
```
/* File: temperature2.h */
class temperature
{   ...
    inline double kelvin() const;
};

inline double
temperature::kelvin() const
{
    return degree;
}
```

Example: A C++ Class for Books









Example: Class with Dynamic Data Members — book.h

```
#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 = nullptr; num_pages = n; }
    Book(const char* t, const char* a, int n = 5) { set(t, a, n); }
    ~Book()
    {
        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;
    }
};
```

Example: Class with Dynamic Data Members — book.cpp

```
#include "book.h"          /* File: book.cpp */
void make_books()
{
    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()    /* An app written by an application programmer */
{
    Book x("Sapiens", "Y. N. Harari", 1000);
    Book* z = new Book("Outliers", "Gladwell", 300);
    make_books(); cout << endl;
    delete z; cout << endl;
    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 "Sapiens"
```

- 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).

Example: 2 C++ Classes — Bulbs and Lamps

Example: Bulbs and Lamps

- This example consists of 2 classes: Bulb and Lamp.
- A lamp has at least one light bulb.
- All bulbs of a lamp are the same in terms of price and wattage (power).
- The price of a lamp that is passed to the Lamp's constructor does not include the price of its bulbs which have to be bought separately.
- One installs bulb(s) onto a lamp by calling its member function install_bulbs.

Example: lamp-test.cpp

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

int main()
{
    Lamp lamp1(4, 100.5); // lamp1 costs $100.5 and needs 4 bulbs
    Lamp lamp2(2, 200.6); // lamp2 costs $200.6 and needs 2 bulbs

    // Install 4 bulbs of 20 Watts, each costing $30.1 on lamp1
    lamp1.install_bulbs(20, 30.1);
    lamp1.print("lamp1");

    // Install 2 bulbs of 60 Watts, each costing $50.4 on lamp2
    lamp2.install_bulbs(60, 50.4);
    lamp2.print("lamp2");

    return 0;
}
/* To compile: g++ -o lamp-test lamp-test.cpp bulb.cpp lamp.cpp */
```

Example: bulb.h

```
/* File: bulb.h */

class Bulb
{
  private:
    int wattage;          // A light bulb's power in watt
    float price;          // A light bulb's price in dollars

  public:
    int get_power() const;
    float get_price() const;
    void set(int w, float p); // w = bulb's wattage; p = its price
};
```

Example: bulb.cpp

```
/* File: bulb.cpp */

#include "bulb.h"

int Bulb::get_power() const { return wattage; }

float Bulb::get_price() const { return price; }

void Bulb::set(int w, float p) { wattage = w; price = p; }
```

Example: lamp.h

```
#include "bulb.h"       /* File: lamp.h */
class Lamp
{
  private:
    int  num_bulbs; // A lamp MUST have 1 or more light bulbs
    Bulb* bulbs; // Dynamic array of light bulbs installed onto a lamp
    float price; // Price of the lamp, not including its bulbs

  public:
    Lamp(int n, float p);       // n = number of bulbs; p = lamp's price
    ~Lamp();

    int total_power() const;    // Total power/wattage of its bulbs
    float total_price() const; // Price of a lamp PLUS its bulbs

    // Print out a lamp's information; see outputs from our example
    void print(const char* prefix_message) const;

    // All light bulbs of a lamp have the same power/wattage and price:
    // w = a light bulb's wattage; p = a light bulb's price
    void install_bulbs(int w, float p);
};
```

Example: lamp.cpp

```
#include "lamp.h"        /* File: lamp.cpp */
#include <iostream>
using namespace std;

Lamp::Lamp(int n, float p)
          { num_bulbs = n; price = p; bulbs = new Bulb [n]; }
Lamp::~Lamp() { delete [] bulbs; }
int Lamp::total_power() const
          { return num_bulbs*bulbs[0].get_power(); }
float Lamp::total_price() const
          { return price + num_bulbs*bulbs[0].get_price(); }
void Lamp::print(const char* prefix_message) const
{
    cout << prefix_message << ": total power = " << total_power()
         << "W" << " , total price = $" << total_price() << endl;
}
void Lamp::install_bulbs(int w, float p)
{
    for (int j = 0; j < num_bulbs; ++j)
        bulbs[j].set(w, p);
}
```

Part IV

Further Reading: 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();          // 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();
    // ACCESSOR member functions: declared const
    int length() const;
    void print() 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: A Testing Program Using the mystring Class

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

int main()
{
    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 = nullptr; }
};
```

Example: mystring Class Constructors Using LL

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

mystring::mystring() { head = nullptr; } // 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 = nullptr; 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 = nullptr;  // Set the last ll_cnode to point to NOTHING
}
```

Example: mystring Class Accessor Functions Using LL

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

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

    return length;
}

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

    cout << endl;
}
```

Example: mystring Class Mutator Functions — insert( )

```
#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 == nullptr) // 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 != nullptr;
         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 = nullptr; // Point to previous ll_cnode
    ll_cnode* current = head;  // Point to current ll_cnode
    // STEP 1: Find the item to be removed
    while (current != nullptr && current->data != c)
    {
        previous = current;    // Advance both pointers
        current = current->next;
    }

    if (current != nullptr)       // 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 LL

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

mystring::~mystring()
{
    if (head == nullptr) // No need to do destruction for 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 != nullptr; 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 <cstring>
#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();               // Construct an emtry string
    mystring(char);           // Construct from a single char
    mystring(const char[]); // Construct from a C-string
    // DESTRUCTOR member function
    ~mystring();
    // ACCESSOR member functions: Again declared const
    int length() const;
    void print() 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

```
/* File: mystring_constructors_destructor_accessors.cpp */
#include "mystring.h"
// Constructors
mystring::mystring() { 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() { }

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