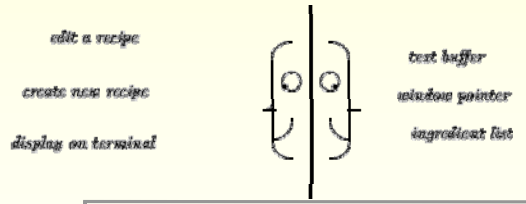
Lecture 4 :
Data Abstraction with C++ : class

Data Abstraction

- Data Abstraction
  - Creating classes with appropriate functionality, which hide the unnecessary details
- Data Encapsulation
  - Implementation is encapsulated by the interface and the only way to get to the implementation legally is through the interface
- Information Hiding
  - Implementation is hidden from user

Internal and External Views

- Data Abstraction
  - Separation of interface and implementation
  - Interface : the outside, or service view, describes **what** an object does
  - Implementation : the inside view describes **how** it does
  - A Service View
    - The ability to characterize an object by the service it provides, without knowing how it performs its task



<two views of the same system>

Benefits of Data Encapsulation

Class : Behavior and State

- Class
  - Fundamental unit of data abstraction in C++
- Behavior (method)
  - The actions that an instance can perform in response to a request.
  - Implemented by methods
- State (variable)
  - The data that an object must maintain in order to successfully complete its behavior.
  - Stored in instance variables (also known as data members, or data fields).

Methods (member function)

- Name
  - will be matched to a message to determine when the method should be executed
- Signature
  - the combination of return type and argument types
  - Methods with the same name can be distinguished by different signatures
- Body
  - the code that will be executed when the method is invoked in response to a message

Message Passing

- Differences : message & a function call
  - A message is always given to some object, called the receiver
  - The action performed in response is determined by the receiver, different receivers can do different actions in response to the same message

Message Passing Syntax

- Three different part

```
aGame.displayCard (aCard, 42, 27);
```

  - Message receiver
  - Message selector
  - Arguments (optional)

Avoid data members in the public interface

- Why?
  - Consistency
  - Precise control over the accessiblity of data members
    - No access, Read-only, read-write, write-only
  - Functional abstraction
    - You can later replace the data member with a computation

```
class AccessLevels {
public :
          int getReadOnly() const { return readOnly; }
          void setReadWrite(int value) { readWrite=value; }
          int getReadWrite() const { return readWrite; }
          void setWriteOnly(int value) { writeOnly=value; }
private :
          int noAccess; // no access to this int
          int readOnly;
          int readWrite;
          int writeOnly;
};
```

Accessor (or getter) method

- A method that simply returns an internal data value
- Why Use?
  - make the data field read-only
  - Provides better documentation that the data field is accessible
  - Make it easier to later change the access behavior (e.g. count # of accesses)

Mutator (setter) method

- used to change the variable of an object
- Mutators are less common than accessors, but reasons for using are similar

Ensuring reliability of abstractions- class invariants and assertions

- Class invariant
  - Every class might have conditions that must always be true inside an object.
  - Add the condition to ensure that every member function code checks these conditions
- In C/C++, we use assert macro : assert(e)
  - If e is true, just continue
  - If e is false, the program is stopped with an error message indicating the failure of the assertion.
  - #include <assert.h>
  - can be used to implement programs that are more reliable

Class example

```
class PlayingCard {
public:
   enum Suits {Spade, Diamond, Club, Heart};
   enum Colors {Red, Black};

   PlayingCard ();
   PlayingCard (Suits is, int ir)                // constructor
   {    suitVa;ie = is; rankValue = ir; faceUp = true; }
   ~PlayingCard () {}                            // destructor

   Suits suit () const { return suitValue; }          // getter
   int rank () const;              // getter
   void setFaceUp (bool up) { faceUp = up; }  // setter
   Colors color ();

private:
   Suits suitValue;
   int rankValue;
   bool faceUp;
};
```

Visibility Modifier

- differentiate the interface and implementation views of a class
- public
  - external (interface , service) view
  - can be seen and manipulated by anybody (all clients)
- private
  - internal (implementation) view
  - accessible only within a class (class implementors)
- protected
  - Discussed in inheritance

Constructor

- a method that initializes data members of a newly created object.
- Implicitly invoked when an object is created
- No return type/value
- has the same name as the class

```
class PlayingCard {
// constructor, initialize new playing card
public :
        PlayingCard (Suits is, int ir)
        {       suit = is; rank = ir; faceUp = true; } ...
} ;
```

- Member initializer

```
        PlayingCard (Suits is, int ir) : suit(is), rank(ir)
        {       faceUp = true; } ...
```

Default constructor

- If we did not declare any constructor in class, the compiler would generate a default constructor.
- The default constructor looks like :

```
TintStack::TIntStack() { /* Empty */ }
```

Copy Constructor (1)

- A copy constructor is called whenever a new object is created from an existing object with the same type.
- Typical constructor
  - `Person q("Mickey");` // constructor is used to build q.
- Copy constructor
  - `Person r(p);` // copy constructor is used to build r.
  - `Person p = q;` // copy constructor is used to initialize in declaration
  - `f(p);` // copy constructor initializes formal value parameter.
- Assignment operator
  - `p = q;` // Assignment operator, no constructor or copy constructor.
  - // interpreted as `p.operator=(q);`
  - `Person& operator=(const Person& p) { … }`

Copy Constructor (2)

- How to declare copy constructor?
  - `Person(const Person& p) { … }`
- Don't write a copy constructor if shallow copies are ok
  - Compiler automatically generates default copy construct, default operator assignment and default destructor if we don't provide them.
  - Shallow copy ?
  - Deep copy ?
- If you need a copy constructor, you also need
  - a destructor and
  - operator=
- Why?

Object Creation

- Example

```
PlayingCard aCard(diamond, 3);

PlayingCard* aCardPtr = new PlayingCard(diamond, 3);
PlayingCard* CardsPtr = new PlayingCard[100];
// dynamic memory allocation

delete aCardPtr;  // deallocation
delete [] CardsPtr;   // deallocation
```

- Memory Leak ?
- Dangling Pointer?
- Garbage Collection ?

static and const

- static
  - all instance share the same value.
  - One per class
- const
  - it will not be reassigned

```
static const int Spade = 1;
static const int Diamond = 2;
static const int Club = 3;
static const int Heart = 4;
```

`static` example

```
// In File.h
class File {
    static int open_files;
    // detail omitted
public :
    File(const String& filename, const String& modes);
    File(const File&);
    ~File();
    static int existing() { return open_files; } //  File::existing() will return
                                                 //  the number of existing File objects
    //  static method cannot access non-static class data and functions
}

// In File.c
int File::open_files = 0;

File::File(const String& filename, const String& modes) {
    ++open_files;
    // othe details omitted
}

File::~File() {
    --open_files;
    // other details omitted
}
```

`const` member functions

- **`const`** member function does not modify any data members of the object on which it is called. (read-only function)
- This restriction is enforced by the compiler
- Benefits of **`const`** member function
  - A **`const`** member function assures its clients of its benign nature. (safety , readability)
- Example

```
int PlayingCard::rank () const
{ return rankValue; }
```

inline member function

- What is inline function ?
- Why useful?
- inline function in classes

Order of methods

- Guideline
  - List important topics first
  - Constructors are generally very important, list them first
  - Put public features before private ones
  - Break long lists into groups
  - List items in alphabetical order to make it easier to search

Separation of definition and implementation

```
class PlayingCard {
   public: ... Colors color () ;
... };

PlayingCard::Colors PlayingCard::color ( ) {
   // return the face color of a playing card
   if ((suit == Diamond) || (suit == Heart))
       return Red;
   return Black;
}
```

friend

- **friend** function of a class has the right to access the non-public members of the class
- To declare all member functions of class ClassTwo as friends of classOne, place a declaration of the form

```
friend class ClassTwo;
```

in the definition of class ClassOne

friend example

```
class Count
{
   friend void setX(Count& , int);
public :
   Count() : x(0) { }
private :
   int x;
};

void setX(Count &c, int val)
{
   c.x=val;
}

…
```

Reference Operator '&'

- reference : alias for an object
- Frequently used for pass-by-reference
- can be used to avoid the run-time impact of passing large arguments by value
- Parameters, return values, and variables can all be defined as "references"

References

- call by reference

```
double square (double &x) { return x *= x; }
int bar (void) {
      double foo = 10.0;
      square (foo);
      cout << foo; // prints 100.0
}
```

- In C, this can be written using pointer

```
double square (double *x) { return *x *= *x; }
int bar (void) {
      double foo = 10.0;
      square (&foo);
      printf (%f, foo); /* prints 100.0 */
}
```

References

- A function can also return a reference to an object, i.e., an lvalue

```
Vector<int> v (10);
v[3] = 100; // v.operator[] (3) = 100;
int i = v[3]; // int i = v.operator[] (3);
```

References

- References are implemented similarly to const pointers.
- Difference ?
  - References must refer to an actual object, but pointers can refer to lots of other things that aren't objects, e.g.,
    - Pointers can refer to the special value 0 in C++ (often referred to as NULL)
  - In general, use of references is safer, less ambiguous, and more restricted than pointers

```
void printDouble(const double& rd)
{ cout << rd; } // no need to test rd; it must refer to a double

void printDouble(const double *pd)
{ if (pd) cout << *pd ; } // check for null pointer
```

Argument Pasing Modes (The Client's View)

```
1. void X::f(T arg)            // pass by value
2. void X::f(const T arg)      // pass by value
3. void X::f(T& arg)           // pass by reference
4. void X::f(const T& arg)     // pass by referene
5. void X::f(T* argp)          // pass by pointer
6. void X::f(const T* argp)    // pass by pointer
7. void X::f(T* const argp)    // pointer is constant
8. void X::f(const T* const argp)
```

- X : class name
- T : data type

- Do not ever use pass by reference (without const qualifier) mode for read-only parameters

Selecting correct mode for arguments

- Avoid passing large objects by value
  - Why?
- When passing primitive type arguments (or small objects), use value arguments
  - pass-by-value is safer
- Use const qualifiers for arguments and functions wherever possible.
  - Make the program more robust
- Don't pass pointers when you mean to pass objects.
  - `void f (T* object);`
  a. Pointer object can be NULL. But, there is no null reference.
  b. Passing a reference does not tansfer the ownership of the object to the callee.

Writing memory safe classes

- Avoid making copies of objects. Copying an object is expensive
  - Avoid creating new objects. Try to reuse existing objects
  - Use const reference parameters when appropriate
  - Use const member functions
  - Use pointers instead of references as data members.
  - Avoid storage allocation in the default constructor. Delay it until the member is accessed.
  - Use reference counting wherever possible.
- Resist tricks in the initial stage
  - Stick with safety first; make sure no memory leaks exist
  - Don't worry about optimization in the early stages of development.

Client's responsibilities with classes and functions

1. understand the purpose of the class. Even if the name of the class indicates that if it is what you need, the documentation may suggest otherwise.
2. Clearly understand what the class implementor expects from you, the client. There is a contract between the client and the implementor.
3. Pay attention to every member function; in particular, const member functions-ther are safer
4. Understand the arguments being passed.
5. Understand what your responsibility is when pointers and references are returned from functions.
6. If the class documentation and the information in the header file differ, get clarification as to which is correct one before using the class
7. Prefer functions that accept arguments that are pointers and references to const. Ther are safer.
8. Even if the class implementor is your brother, don't rely on any undocumented class details that he might have told you. Stick to the class interface and documentation.
9. Be wary of classes that don't implement the minimum set of member functions

C++ Advanced Topics

Guideline

- Never return a reference to a local object within the function.

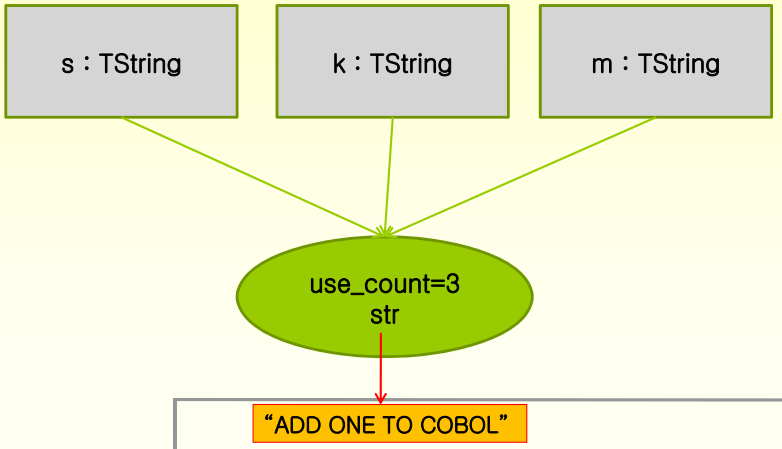
```
#include <cassert> // assert
#include <iostream> // cout, endl
int f () { int i = 2; return i;}
int* g () { int j = 2; return &j;} // warning: address of local variable 'j' returned
int& h () { int k = 2; return k;} // warning: reference to local variable 'k' returned

int main ()
{
          using namespace std;
          cout << "FunctionReturns.c++" << endl;
          { int v = f(); assert(v == 2); }
          { int* p = g(); assert(*p == 2); // ? }
          { int& r = h(); assert(r == 2); // ? }
          cout << "Done." << endl;
          return 0;
}
```

Copy-On-Write

- Object creation and destruction is expensive
  - If the copy constructor of a large object is frequently called, significant time is spent in copying and deleting the dynamically allocated memory.
- Copy-on-write
  - Allocate new memory only when writing!
  - Otherwise share the data between original and newly created object.
  - Keep reference count (use count) variable.

TString Example

- Look at TString.h and TString.cpp on class website



TString Example

- s.to_lower();
  - writing to s creates memory for new data