Topic 1

C++ Review Part I:

The Basic: Variables, Classes, IO Streams

資料結構與程式設計

Data Structure and Programming

09.12.2018

A Proclaimer...

- This is NOT a concise “Computer Programming in C++” lecture note!!
  - I assume you know the basics
- Contents are NOT organized as a complete C++ tutorial
  - More like an itemized focal review
- But, anyway, if you think some contents are not clear, feel free to raise your questions!!

A Proclaimer...

- This lecture note contains a lot of details...
  - Not to memorize the details, but to understand why the language is designed that way.
- You need to have a good sense for programming, and at the same time be precise on the details.

Part I: Understanding “Variables”

- What is a variable?
- The concept of “memory”
- Object, pointer, reference

Key Concept #1: Variable

```
int a = 10;
```

0x7fffa33be5d4	10

- Variables are stored in memory
  - Where is it stored?
    ➔ Memory address
  - What is it stored?
    ➔ Memory content (value)

?? What about "a" ??

- The name of the variable
  ➔ NOT part of the "executable".

?? Why "int" ??

  - Used by compiler to associate the assignments and operations with the variable (in the symbol table)
  ➔ For ease of programming and debugging
- The type of the variable
  ➔ To determine the "size" of the memory
  ➔ To interpret the meaning of the memory content

Key Concept #2: Memory Sizes

- Basic "memory size" unit ➔ Byte (B)
  - 1 Byte = 8 bit
- 1 memory address ➔ 1 Byte
  - Like same sized apartments
- Remember: the variable type determines the size of its memory
  - char, bool: 1 Byte (addr += 1)
  - short, unsigned short: 2 Bytes (addr += 2)
  - int, unsigned, float: 4 Bytes (addr += 4)
  - double: 8 Bytes (addr += 8)
  - long long: 8 Bytes (addr += 8)

Key Concept #3: Operation on Variables

- Operation on variables
  ➔ Perform operation on the corresponding memory contents
  - a + b
    ➔ retrieve the contents of "a" and "b" and perform the addition
  - Where is the result stored?
  - What about the "=" operator in "c = a + b"?

```
int a = 10;
int b = 20;
int c = a + b;
```

0x7fffa33be5d4	10
0x7fffa33be5d8	20

Key Concept #4: '=' operator

- '=' operator in C/C++ performs "assignment", not "equal to" (so "a = a + 1" makes sense)
  - "Assignment" means "copy the value of the right hand side expression to the location of the left hand side variable"
    - `c = a + b;`
    ➔ Where is the result of "a+b" stored?
  - int a = b; // let b = 10 now
    b = 20; // what is the value of 'a'?
  - What about:
    - ```
      int *p = q;
      int *r = new int(10);
      ```
      ```

Key Concept #5: Pointer Variables

◆ Pointers are also variables
- int a;
  The memory location of "a" stores an integer value.
- int *p;
  The memory location of "p" stores a memory address, which points to an integer memory location.

◆ "a" vs. "p"
- Both are variables
- Different types: "int" vs. "int *"

Key Concept #6: Size of a Pointer

◆ Remember:
A pointer variable stores a memory address
- What is the memory size of a memory address?

◆ The memory size of a memory address depends on the machine architecture
- 32-bit machine: 4 Bytes
- 64-bit machine: 8 Bytes

◆ Remember: 1 memory address ➔ 1 Byte
➔ The memory content of the pointer variables
: For 32-bit machine, the last 2 bits are 0's
: For 64-bit machine, the last 3 bits are 0's

Key Concept #7: Reference Variables

◆ A reference variable is an "alias" ("symbolic link") to another variable
- Has the same address entry in the symbol table as the referred variable
- Gets modified simultaneously with the referred variable
- int& a = b; // let b = 10 now
  b = 20; // what is the value of 'a'?

◆ Must be initialized (defined) when declared (why?)
- (Good) int& i = a; // a is an int
- (Bad) int& i;
- (Bad) int& i = 20; // Why not??

◆ Used like the referred variable
- MyClass& o1 = o2;
  o1.getName(); // no (*o1), nor o1->getName()

Reminder: C++ operators

◆ a.dataMember;
a.memberFunction();
- 'a' as an object type variable to access its data member and member function

◆ p->dataMember;
p->memberFunction();
- 'p' as a pointer type variable to access its data member and member function

◆ int *p = &i;
- '&' is to return the address of 'i'

◆ int b = *p;
- '*' is to return the content (value) of the memory that 'p' is pointing to

Summary #1: Types of Variables

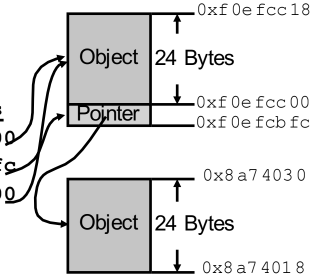
1. Object type
   - int i = 10;
   - MyClass data;
   - data.memFunction(); (&data)->memFunction();
2. Pointer type
   - int* i = new int(10);
   - MyClass* data = new MyClass("ric");
   - data->memFunction(); (*data).memFunction();
3. Reference type
   - int& i = j;
   - MyClass& data = origData;
   - MyClass *& pointer = origPointer;

Object, Pointer, Reference?

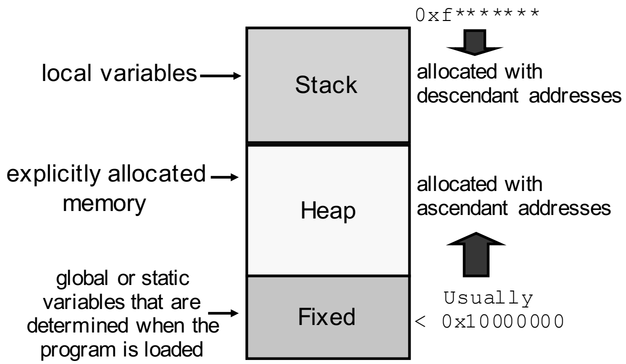
◆
```
void goo(){
   MyClass    aaa;  // Object(Let size = 24Byte)
   MyClass*   ppp;  // Pointer
   MyClass&   rrr = aaa;  // Reference
   ...
}
```

◆ Symbol table

name	address
aaa	0xf0efcc00
ppp	0xf0efcbfc
rrr	0xf0efcc00

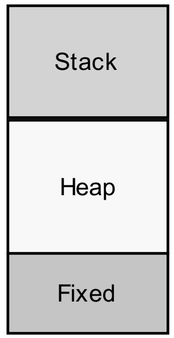


Key Concept #8: Types of Memory Allocations



Scope and Visibility

1. Local variable (Stack mem)
   - Stack: first in last out
   - Only visible within the local scope (i.e. {…})
   - Constructed when entering the scope; destructed when exiting
2. Explicitly allocated (Heap mem)
   - Must be explicitly allocated and freed (e.g. by "new", "delete")
     ➔ Otherwise, memory leaks
3. Global variable (Fixed mem)
   - Visible by the entire program
   - Existed when program starts
   - Use "extern" to refer to global variable that is defined in other file



Key Concept #9:
Every variable that is NOT global, is local.

- { int a; ... }
  - ‘a’ is a local variable stored in stack memory
- { int *p; ... }
  - ‘p’ is also a local variable stored in stack memory
- The content of ‘a’ is an “int” (integer), while the content of ‘p’ is an “int *” (an address, pointing to a memory location that stores an integer)

Address vs. Content

- Address
  - The memory location where a variable is stored
  - int i; // the address of i is in stack memory
  - int *p; // the address of p is ALSO in stack memory
- Content
  - The data which the memory location contains
  - int i = 10; // the content of i is 10
  - int *p = &i; // the content of p is the address of i
    ➔ So, can we do “delete p”?

Key Concept #10:
int *p1 = &i; vs. int *p2 = new int;

- p1 and p2 are both local variables stored in stack memory
  - The contents of p1 and p2 are both memory addresses
  - However, p1 points to a location in stack memory, while p2 points to a location in heap memory
- [Note]
  Pointer variables are NOT necessarily pointing to a “heap” memory
  - Pointer variables are NOT necessarily related to “new” operators
  - Therefore, NOT all pointer variables are required to be “deleted”

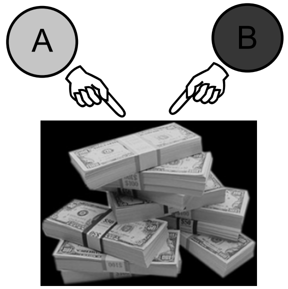
Key Concept #11:
“new” and “delete” operators

- “new” is to acquire memory from system; “delete” is to release memory to system
  - Refer to the “heap” memory
- “new” operator returns the “address” of the memory it acquires
  - int *p = new int(20)
    ➔ What is the content of ‘p’?
    ➔ What about ’20’?
- Why “heap” memory? What are the differences from the stack memory?
  - “stack”: first in, last out.
    ➔ [Think] How is the program executed?
    How are the variables arranged?
  - “heap” memory: something will “live” unless it is explicitly killed/freed (e.g. by “deleted”)

Can you answer this...

- Why do we need "pointer" in C/C++?

"Share" !!




```
compared:
int a = 10;
int b = a;
a += 10;
```

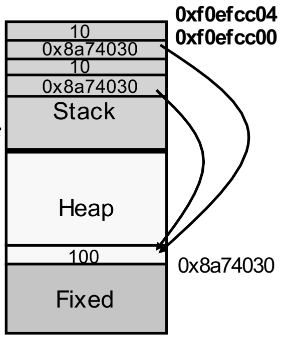
Share what?
Not the memory locations of the variables A, B,
but the memory location they point to.

Summary:
A Simple Example

- ```
  int  i = 10;
  int* p = new int(100);
  int  j = i;
  int* q = p;
  ```

- Symbol table

name	address
i	0xf0efcc00
p	0xf0efcbfc
j	0xf0efcbf8
q	0xf0efcbf4



What's the address of i?
What's the address of p?
What's the content of i?
What's the content of p?

Remember: '=' performs assignment

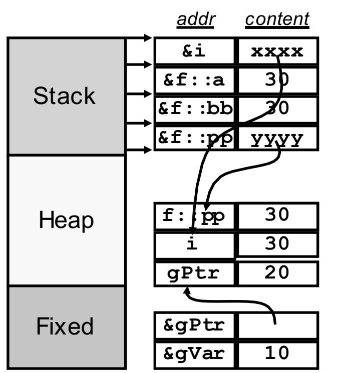
- int a = b;
  - Copy the content (value) of "b" to "a"
- int *p = q;
  - Copy the content (value) of "q", which is a memory address, to "p"
  - (Question) Is "int *p = 10" OK?
  - (Question) Is "int *p = (int *)10" OK?
- int *p = &a;
  - Copy the address of "a" to (the content of) "p"
- int a = *p;
  - Copy the content of the memory location that "p" points to, to "a"
  ```

Copy the content, but, what is the content?

- ```
  int a = 10;
  int b = 20;
  int *p = &a;
  int *q = p;
  *q = 30;    // what are the values of a, b, p, q?
  p = &b;     // what are the values of a, b, p, q?
  b = 40;     // what are the values of a, b, p, q?
  ```
- ```
  int a = 10;
  int b = 20;
  int& i = a;
  int j = i;  // what are the values of a, b, i, j?
  j = 30;     // what are the values of a, b, i, j?
  i = b;      // what are the values of a, b, i, j?
  ```


Another Memory Allocation Example

Operation : exiting function




```
int  gVar = 10;
int* gPtr = new int(20);

void f(int a)
{
   int  bb = a;
   int* pp = new int;
   *pp = bb;
   delete pp;
}

int main()
{
   int* i = new int(30);
   f(*i);
   f(gVar);
   f(*gPtr);
}
```


Can you answer this...

- Why do we need "reference" in C/C++?

"Share" vs. "Alias"!!

Why should we share?
Why should we clone?

Uh?

Share vs. Alias…

What's the difference?

Part II: Understanding "Classes"

- What is a "class"?
- Constructor, destructor
- new, new [], delete, delete []
- A*, A**, A***....
- Access privilege: private/protected/public
- Friend

Key Concept #1: Class = data type

- A class is a user-defined data type
  - Compared to: predefined data types (int, char, ..., etc)
- A variable of a class type is called an object
  - int i;
  - A a;
- Classes define the "data structure" of the program
  - Data members: What to operate?
  - Member functions: How to operate?

Key Concept #2: Data Members, Member Functions

- "Data members" define what the contents of a class type are
  - Every instantiated class object "constructs" a copy of these data members
- "Member functions" define how to operate the object of a class type
  - When a member function is called, you should note that there is an object of this class type that calls the function

  ➔ That's why we have "this" in member functions

Key Concept #3: Constructor/Destructor

- Constructor is to "construct" (initialize) a class object, NOT to allocate the memory
  - Memory is automatically allocated by system (i.e. local variable in hash memory), OR explicitly allocated by the "new" operator in heap memory.
  - Memory has already been allocated when the constructor is called.
- Similarly, destructor is to reset the class object, NOT to release the memory
  - The destructor is called before the memory is released.

Data member initialization and reset

- Constructor will recursively call the constructors of its data members

```
class A {
   B  _b;
public:
   A() { ...; }
```

_b's constructor is called here... before the body of the constructor function is executed.

```
   ~A() { ...; }
};
```

The body of the destructor is first executed... before _b's destructor is called here.

Key Concept #4: Data Member Initializer

- What if we need to pass in parameters to the data member's constructor?
  - A(int i) { ... _b(i); ... } // Error: _b is not a function. This is eq to "_b.operator() (i)".
  - A(int i) { ... _b = B(i); ... } // OK, but extra object copy is performed.
- A(int i) : _b(i) { ...; }
  ➔ Calling _b's constructor and passing in parameter(s)
  ➔ The only chance to pass in parameters for data members' constructors

Key Concept #5: Default constructor

- Constructor in a class can be omitted.
  If there's no constructor defined for a class, the compiler will implicitly invoke a "default constructor" which is conceptually equal to "A() { }"
  - class A { // assume no constructor is defined
       B _b;
    };
    A a; // This is OK. A() will be implicitly defined and called
- The behavior of the default constructor is just recursively calling constructors of its data members

Missing Default Constructor

- However, if any (other) constructor is defined, no implicit default constructor will be assumed
  - ```
    class A {
       A(int) { ...; }
    };
    A a;
    ```
    // Error: A() is not explicitly defined!!
- Solutions:
  1. Define default argument
     ```
     A(int i = 0) { ...; }
     ```
  2. Explicit define default constructor
     ```
     A() { ...; }
     A(int i) { ...; }
     ```


Key Concept #6: Copy Constructor

- When an assignment is performed on a class object (e.g. A a2 = a1), the "copy constructor" will be implicitly inferred. That is, conceptually, "A a2(a1)" will be implicitly called.
  - The prototype for copy constructor: A(const A&)
- You don't need to define your own copy constructor. Compiler will explicitly define one.
  - The default behavior of the copy constructor is to perform the member-wise copy (i.e. calling copy constructors for all its data members)

Customized Copy Constructors

- Of course, if you define your own copy constructor, your own copy constructor will be called (but make sure you do it right!)
  - class A {
    public: A(const A&) { cout << "Haha…\n"; }
    private: B _b;
    };
    int main() { A a1; A a2 = a1; }
    → The problem is:
    Will B's copy constructor be called (i.e. a2._b(a1._b))?
    → How to fix it?

Copy constructor or "=" operator?

- As we said, "A a2 = a1" will call the copy constructor "A a2(a1)"

→ What if "operator =" is overloaded?

- Note:
  - A a2 = a1; // copy constructor will be called
  - A a2; a2 = a1; // default constructor will be
    // called, and then assign
    // operator "=" will be called.

  (But this can be compiler dependent)
     ```

Key Concept #7: Pointer Data Members

- `class A {`
```
   B   _b;
   C  *_c;
};
A a;
```
  - When A's constructor is called, B's constructor will be recursively inferred, but no constructor will be called for "C", unless an explicit "new" is called for "A::_c". (why?)
  - Similarly, no destructor will be called for "A::_c" by default.

Be careful if we initialize the pointer data members…

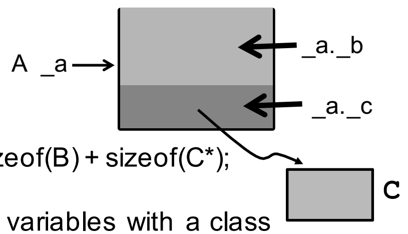
- `class A {`
```
   B   _b;
   C  *_c;
};
A a1, a2;
// do something on a1…
a2 = a1; // copy a1 to a2
```

```
Something like:
A() { ...; _c = new C; ... }
~A () { ...; delete _c; ... }
```

  - The program will crash when program exits… (Why?)
  - There will be memory leak (Where?)

Key Concept #8: Size of a Class

- The size of a class (object) is equivalent to the summation of the sizes of its data members
```
class A {
   B   _b;
   C  *_c;
};
```



  ➔ sizeof(A) = sizeof(B) + sizeof(C*);

- Wrapping some variables with a class definition DOES NOT introduce any memory overhead!!

Summary #1: Calling Constructors

1. When a program enters a scope, all the memory of the local variables will be allocated, and their constructors will be called when the corresponding lines of codes are executed.
2. When the constructor of a class object is called, the constructors of its data members will be recursively called.
3. When the "new" operator is executed, the required memory will be granted, and the constructor of that class will be called.

Summary #2: Memory and constructor

- The memory of an object is allocated before the constructor is called.
- Don't use "malloc()", "calloc()", "free()", etc. C functions to allocate/delete memory
  - → Constructor and destructor will NOT be called!!

```
class A {
   string    _str;
};
A *a = (A*)malloc(sizeof(A));
a->...; // crash later!!
```


Constructor/Destructor, how many are called?

```
MyClass MyClass::g()
{
   return (*this);          <- copying (*this) to return object
}

MyClass f(MyClass a)        <- copy constructor a(const MyClass& i)
{
   MyClass b = a.g();       <- copy constructor b(const MyClass&)
   return b;                <- copying b to return object
}                           <- destructing 'b', 'a' (b before a)

main()
{
   MyClass i;               <- constructing 'i'
   MyClass j = f(i);        <- copy constructor j(const MyClass&)
}                           <- destructing 'j', 'i' ( j before i )
```


Key Concept #10: Array Variables

- An array variable occupies continuous memory locations.
  - int a[10]; // occupies 10 * sizeof(int)
  - int *b[10]; // occupies 10 * sizeof(int *)
  - int c[5][10]; // 5 * int[10]
- Array of class objects
  - A a[10]; // A's constructor is called 10 times
  - A *b[10]; // no constructor will be called
  - A c[5][10]; // How many constructors are called?

Key Concept #11: new and new []

- "new" is to allocate the memory for a single variable; "new []" is to allocate an array variable.
- "new A(i)" passes "i" as an argument for A's constructor; but there's NO "new A[c] (i)".
  - int *p = new int(10); // points to an int = 10
  - int *q = new int[10]; // points to an array int[10]
  - int **r = new int* (&a); // a is an int variable
  - int **s = new int* [10]; // points to an int *[10]
- "new []" is often used to created "dynamic array"
  - int *p; // declared, but size is not yet determined
    …
    p = new int[size];

int, int [], int *[], new int(), new int [], new int*, new int *[] ... orz

- ```
  int   a = 10;
  int   arr[10] = { 0 };
  ```
- ```
  int   *arrP[10];
  for (int i = 0; i < 10; ++i)
     arrP[i] = &arr[i];
  ```
- ```
  int   *p1 = new int(10);
  int   *p2 = new int[10];
  ```
- ```
  int **p3 = new int*;
  *p3 = new int(20);
  ```
- ```
  int **p4 = new int*[10];
  for (int i = 0; i < 10; ++i)
     p4[i] = new int(i + 2);
  ```
- ```
  int **p5 = new int*[10];
  for (int i = 0; i < 10; ++i)
     p5[i] = new int[i+2];
  ```


Key Concept #12: Dynamic Array

- If you are not sure about the size of the array in the beginning, make it a dynamic array.
  - int *arr;
    ...
    size = ...;
    ...
    arr = new int[size];
- "Double pointer" can be used as an array of dynamic arrays, in which each of the dynamic arrays can have different sizes
  - int **darr = new int *[size];
    for (int i = 0; i < size; ++i) {
      darr[i] = new int[size_i];
    }

However…

- ```
  int size;
  cin >> size;
  int a[size];    // this is OK
  string b[size]; // this is NOT OK
  ```
  - error: variable length array of non-POD element type
    // POD = Plain Old Data structure

Key Concept #13: delete and delete []

- "delete" releases the memory of a single occupation; "delete []" releases the memory of an array occupation.
  - int *p = new int(10); …; delete p;
    int *q = new int[10]; …; delete [] q;
  - int *p = new int(10); …; delete [] p;
    // compilation OK, but strange things may happen
    int *q = new int[10]; …; delete q;
    // compilation Ok, but may have memory leak
- No "delete [][]"
  - int **p = new int* (&a); …; delete p;
  - int **q = new int* [10];
    for (int i = 0; i < 10; ++i) { q[i] = new int; }
    …
    for (int i = 0; i < 10; ++i) { delete q[i]; }
    delete [] q;
  ```

See how constructors/destructors are called...

1. What's the difference?
   - `T t1(10);`
   - `T t2[10];`
   - `T* t3 = new T;`
   - `T* t4 = new T(10);`
   - `T* t5 = new T[10];`
   - `T** t6 = new T*[10];`
   - `T* t7 = (T*)calloc(10, sizeof(T));`
   - `delete t3; delete t4;`
   - `delete []t5; delete []t6;`
   - `free(t7);`
2. Any diff?

```
{ ...                  { ...
  return T();            T t; return t;
}                      }
```


Key Concept #14: Access Privilege

- By default, all the data members and member functions in a class are all private
  - To ensure data encapsulation
  - Implementation details are kept in the class. Only public interfaces are open to the users.
- Therefore, in defining a class, put the public session on top.

```
class A {
   public: ...
   private: ...
};
```


public, private, data, functions?

- // In .h file

```
class A
{
public:
  int  _dPub;
  void aPub1() {
    _dPub = 2;
    _dPrivate = 4;
    aPub2();
    aPrivate2();
  }
  void aPub2();
  void aPub3() {}
private:
  int    _dPrivate;
  void aPrivate1() {
    _dPub = 2;
    _dPrivate = 4;
    aPub2();
    aPrivate2();
  }
  void aPrivate2();
  void aPrivate3() {}
};
```

- // In .cpp file

```
void A::aPub2()
{
  _dPub = 2;
  _dPrivate = 4;
  aPub3();
  aPrivate3();
}
void A::aPrivate2()
{
  _dPub = 2;
  _dPrivate = 4;
  aPub3();
  aPrivate3();
}

int main()
{
  A a;
  a._dPub = 2;
  a._dPrivate = 4;
  a.aPub1();
  a.aPrivate1();
}
```


Is this OK?

- `// In .h file`

```
Class A
{
public:
   void f();
private:
   int   _data;
};
class B
{
private:
   int _id
};
```

- `// In .cpp file`

```
void A::f() {
   A a;
   a._data = 10;
   B b;
   b._id = 20;
   _data = 30;
}
```

➔ Any problems?

public, private, data, functions?

- The key: know the scope you are in!!
  - Class scope:
    1. Inside the definition of the class body "class { };"
    2. In the member function definition, even in a separate .cpp file
- Inside the class scope
  - All the member functions and objects of the same class can access ALL (including private) the data members and member functions
  - Objects of other classes can only access to the public data members and member functions
  - Local variables in the member functions still only have the block scope
- Outside the class scope
  - All the functions and class objects can only access the public data members and member functions, even it is an object of the same class

Key Concept #15: Making "friends" between classes

- When a data member is declared "private", all the other classes cannot access it directly
  - ➔ Must call through "member functions"
- Unless, declare myself (MyClass) as "friend" of other class (OtherClass)
  - ```
    class MyClass {
       friend class OtherClass;
       ...
    };
    ```
  - ➔ **<u>Friendship is granted, not taken</u>**
  - ➔ OtherClass can access MyClass's data members
  - ➔ Not recommended (unless no better way)

```
void MyClass::f() {
  OtherClass a;
  a._data = ...;
}
```

OR ??

```
void OtherClass::f() {
  MyClass a;
  a._data = ...;
}
```


Common usage of friend class

- If some class A is designed specifically for another certain class B, and is intended to hide from others...
  - ➔ Making A a private class and only friend to B
- For example,

```
class ListNode
{
   friend class List;
   ...
};
class List
{
   ListNode* _head;
   void push_front(const T& d) {
      _head = new ListNode(d, _head); }
};
```


Key Concept #16: Friend to a (Member) Function

- Instead of making MyClass as friend to the whole OtherClass, however, we can make friend to only certain member functions in OtherClass
  - e.g.

```
class MyClass {
   friend void OtherClass::setData
            (const MyClass&);
   int _db;
  friend ostream& operator <<
           (ostream&, const MyClass&);
  friend void f(); // Is f() a member function?
};

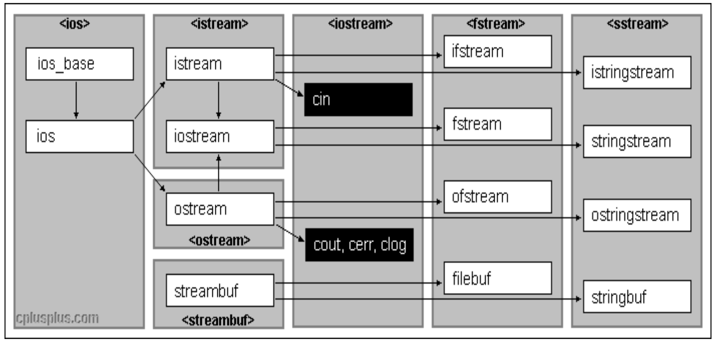
void OtherClass::setData(const MyClass& b) {
   _da = b._db; }
```

- (See also) Operator overload
```

Part III: Understanding “I/O Streams”

- C++ standard I/O
  - Introduction
  - Class hierarchy and included files
  - Class data members and member functions
- File I/O
- I/O manipulators

Key Concept #1: C++ Stream Classes



For more information, recommended:
http://www.cplusplus.com/reference/iostream/

Stream classes, objects, and manipulators

- “Stream”, a nice name
  ➔ Data are conveyed in a steam by “<<“ or “>>”

1. Header files
   - iostream, fstream, sstream, iomanip
2. Classes
   - istream, ostream, iostream, ifstream, ofstream, fstream, istringstream, ostringstream
3. Objects
   - Standard: cin, cout, cerr, clog
   - User defined
4. Manipulators
   - dec, endl, ends, flush, hex, oct, left, right, ws, setbase(n), setw(n), setioflags(i), resetioflags(i), setfill(c), setprecision(n)
5. Member functions

C++ Standard I/O Library Files

- `<iostream>`
  - Basic services for ALL stream-I/O operations
  - Defines `cin`, `cout`, `cerr` and `clog`
  - For both unformatted- and formatted-I/O services
- `<iomanip>`
  - Formatted I/O with parameterized stream manipulators
- `<fstream>`
  - User-controlled file processing
- `<sstream>`
  - String manipulations as I/O stream

Key Concept #2: Standard I/O Stream Objects

<u>Standard Input</u>

- `cin`
  - Connected to the standard input device, usually the keyboard

<u>Standard Output</u>

- `cout`
  - Connected to the standard output device, usually the display screen
- `cerr`
  - Connected to the standard error device
  - Unbuffered - output appears immediately
- `clog`
  - Connected to the standard error device
  - Buffered - output is held until the buffer is filled or flushed

Key Concept #3: User Defined Stream Objects

- File I/O

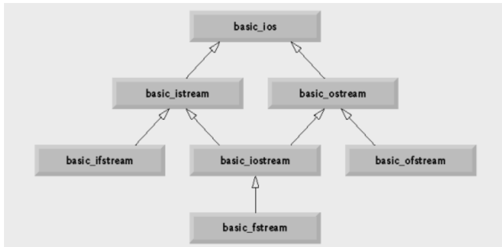
```
ifstream inFile("test.in");
ofstream outFile("test.out");
fstream ioFile;
if (!inFile) {
   cerr << "Cannot open file" << endl;
   exit(0);
}
int i, j, k;
inFile >> i >> j >> k;
outFile.close();
ioFile.open("test.io");
```


Key Concept #4: File Stream

- A file is viewed by C++ as a sequence of bytes
- Ends either with an end-of-file marker (Ctrl-d for Linux and Ctrl-z for Windows) or at a system-recorded byte number (Why diff?)
- Communication between a program and a file is performed through stream objects
  - `<fstream>` header file
    - Stream class templates
      - `basic_ifstream` – for file input
      - `basic_ofstream` – for file output
      - `basic_fstream` – for file input and output
    - Files are opened by creating objects of stream template specializations
      - (i/o)fstream are the char-type template specializations

(FYI) basic_iostream

- Actually, in <iostream>, the I/O stream classes are defined as basic_iostream template classes
  - template <class Elem, class Tr = char_traits<Elem> >
class basic_iostream : public basic_istream<Elem, Tr>,
public basic_ostream<Elem, Tr>
{ ... };



(FYI) iostream vs. basic_iostream

- `istream`
  - typedef basic_iostream<char, char_traits<char> > iostream;
  - Represents a specialization of `basic_istream`
  - Enables `char` input
- `ostream`
  - Represents a specialization of `basic_ostream`
  - Enables `char` output
- `iostream`
  - Represents a specialization of `basic_iostream`
  - Enables `char` input and output

Key Concept #5: Open a file

- Two methods
  - By passing arguments to (i/o)fstream constructor
  - By calling member function open()
- Two arguments
  - A filename // mandatory; char*, not string
  - A file-open mode // optional; default = “out” for ostream, “in” for istream
    - Can use ‘|’ for multiple modes
    - fstream fstr("test.txt", fstream::in | fstream::out | fstream::app);

Mode	Description
`ios::app`	Append all output to the end of the file.
`ios::ate`	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file.
`ios::in`	Open a file for input.
`ios::out`	Open a file for output.
`ios::trunc`	Discard the file’s contents if they exist (this also is the default action for `ios::out`).
`ios::binary`	Open a file for binary (i.e., nontext) input or output.

fstream object

- open(): takes the same arguments as constructor
- Note: you cannot “copy” a stream object
  - So, vector<ifstream> is not possible
- bool operator !()
  - Returns true if either the `failbit` or `badbit` is set
  - if (!fin) { // or if (!fin()) ???
    cerr << “Open file failed...” << endl;
    exit(-1);
  }
  - Note: this is OK too “if (fin) {…}” // covered later

Key Concept #6: Close a file

- Releases the file resource (recommended!!!)
- Two methods
  - By destructor (exit the scope)
  - By calling member function close()

Key Concept #7: I/O Stream Manipulators

1. endl
2. Number base (sticky)
   - hex (e.g. 0x38ab), oct (e.g. 0236), dec (all others)
   - showbase(), setbase(int) // int = 16, 8, 10
3. Precision of floating-point numbers (sticky)
   - fixed, scientific
   - setprecision(int)
   - Note: precision(int) is a member function
4. Field width (not sticky)
   - setw(int) // c.f. "width()" member function
   - For both istream (input size) and ostream (display size)

I/O Stream Manipulators

5. Alignment (sticky)
   - left, right
   - internal (padding fill characters between sign and magnitude)
6. I/O formatting (sticky)
   - showpoint, noshowpoint
   - showpos, noshowpos
   - uppercase, nouppercase
   - boolalpha, noboolalpha
   - setfill (cf. fill() member function)
   - skipws
7. Flush stream buffer
   - flush

A small program: a spinning bar

```
static char s[4]={ '|', '/', '-', '\\' };

int main()
{
   int a = 0;
   while (true) {
      cout << s[a%4];
      cout.flush();
      // add some delay here
      a++; cout << '\b';
   }
}
```


Key Concept #8: Sticky or not sticky?

- Most IO manipulators are "sticky"
  - Exception: field width
- "Sticky" to the manipulated object
  - Not across to another object of the same stream class, or any other object of other stream classes

Use of Manipulators

```
int main()
{
   int i = 100;
   fstream iof("ttt");  // make sure "ttt" exists
   if (!iof) { cerr << "Error" << endl; exit(0); }
   iof << hex << i << endl;
   iof.close();

   int j;
   iof.open("ttt");
   iof >> j;

   cout << setw(10) << right << j << endl;
}
// What's in file "ttt"? What's the output??
```


What's the difference?

```
int main()
{
   int i = 100;
   fstream iof("ttt");
   if (!iof) { cerr << "Error" << endl; exit(0); }
   iof << hex << i << endl;
   iof.close();

   int j;
   iof.open("ttt");
   iof >> dec >> j;

   cout << setw(10) << right << j << endl;
}
// What's in file "ttt"? What's the output??
```


(FYI) About I/O Manipulators

- About floating number display
  - fixed --- in fixed-point notation (e.g. 3.14159)
  - scientific --- in scientific notation (e.g. 3.14159e+002)
  - (none) --- in default floating-point notation; floating-point number's value determines the output format
- About the precision of display
  - setprecision(*numDigits*)
    - For "fixed" and "scientific", *numDigits* is the number of digits after the decimal point
    - For default floating-point notation, *numDigits* is the total number of digits to display