Topic 4
C++ Review Part II: Understanding “Classes”

資料結構與程式設計
Data Structure and Programming

09.30.2015

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

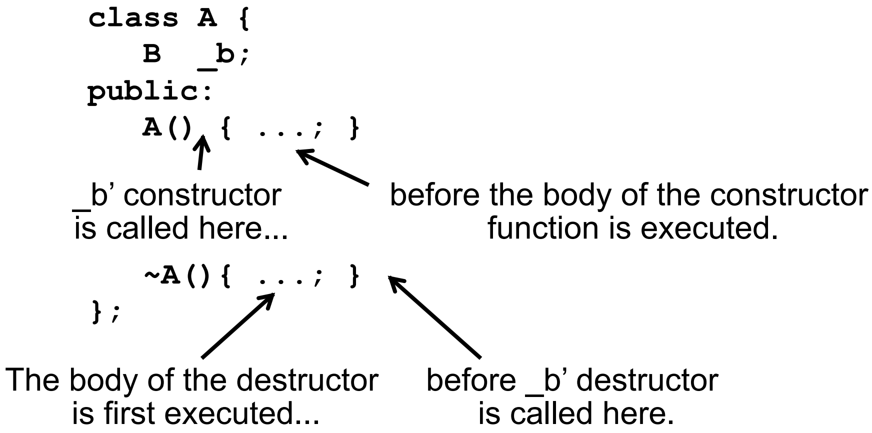
  ➔ That's why we have "this" in member functions

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 calls the constructors of its data members



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

_b' constructor is called here...

before the body of the constructor function is executed.

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

The body of the destructor is first executed...

before _b' 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) { ...; }
  ➔
  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
    ```
    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

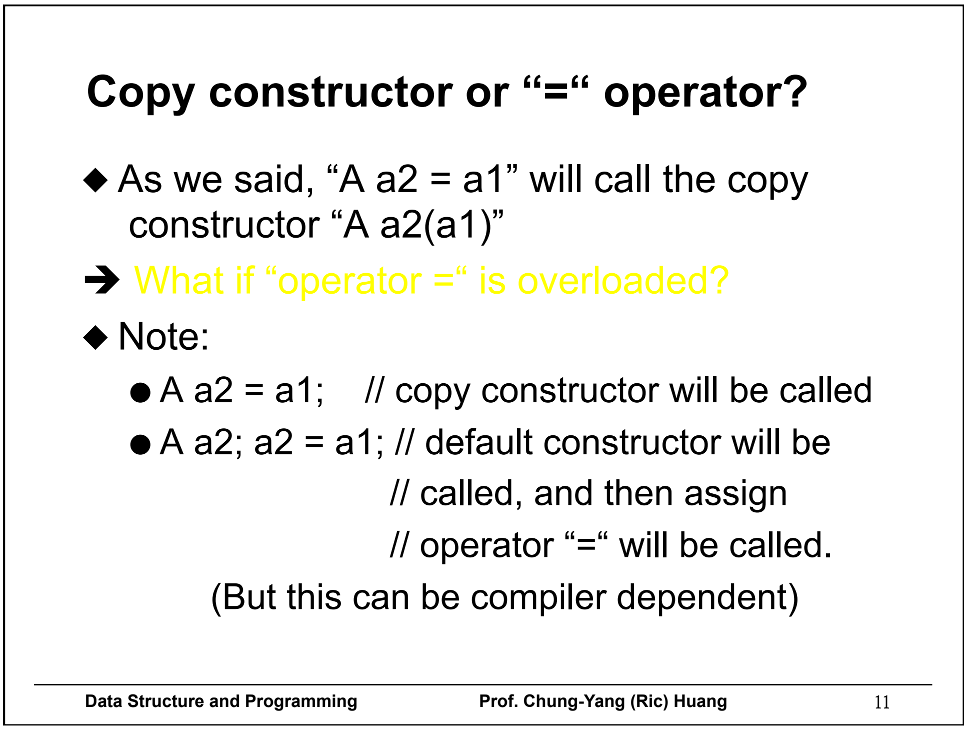
- When an assignment is performed on a class object (e.g. A a2 = a1), the "copy constructor" will be implicitly inferred. That is, conceptually, "A a2(a1)" will be implicitly called.
  - The prototype for copy constructor: A(const A&)
- You don't need to define your own copy constructor. Compiler will 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; }
```

➔ Will B's copy constructor be called (i.e. a2._b(a1._b))?



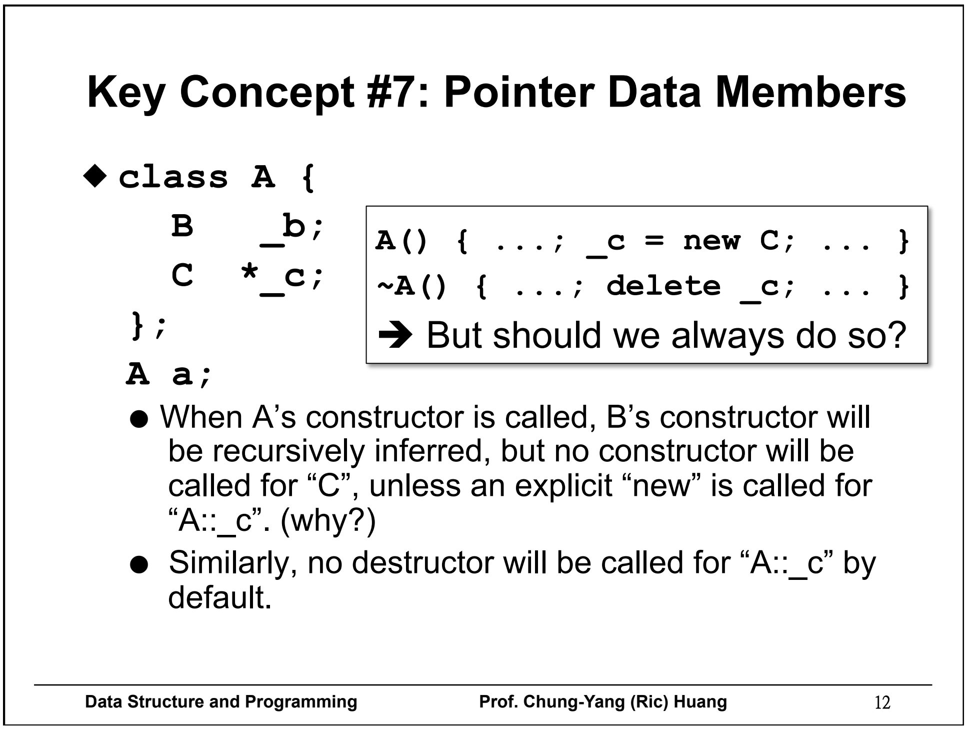
Copy constructor or “=“ operator?

- 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;
```

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

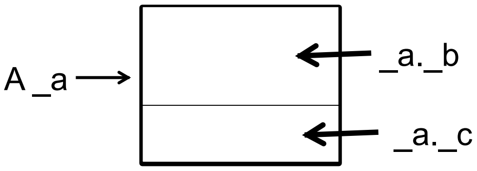
➔ But should we always do so?

- 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.

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!!

Key Concept #9: Class Wrapper

1. To create a “record” type with a cleaner interface
   - e.g. When passing too many parameters to a function, creating a class to wrap them up.

   ➔Making sure data integrity (checked in constructor)

   ➔Creating member functions to enact assumptions, constraints, etc.

Key Concept #9: Class Wrapper

2. To manage the memory allocation/deletion of pointer variables
   - Recap: pointer data member will not be explicitly constructed in class constructor
   - Memory allocation/deletion problems for pointer variables
     - There may be many pointer variables pointing to the same piece of heap memory
     - The memory can NOT be freed until the “last” pointer variable become useless   (HOW DO WE KNOW!!?)
     - What about the pointer (re-)assignment?
   - Recap: The memory of an object variable is allocated when entering the scope, and released when getting out.
   - Recap: The heap memory must be explicitly allocated and deleted.

Object-Wrapped Pointer Variables

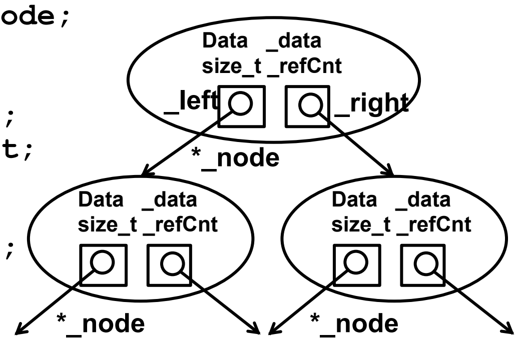
If your program contains pointer-pointed memory that is highly shared among different variables

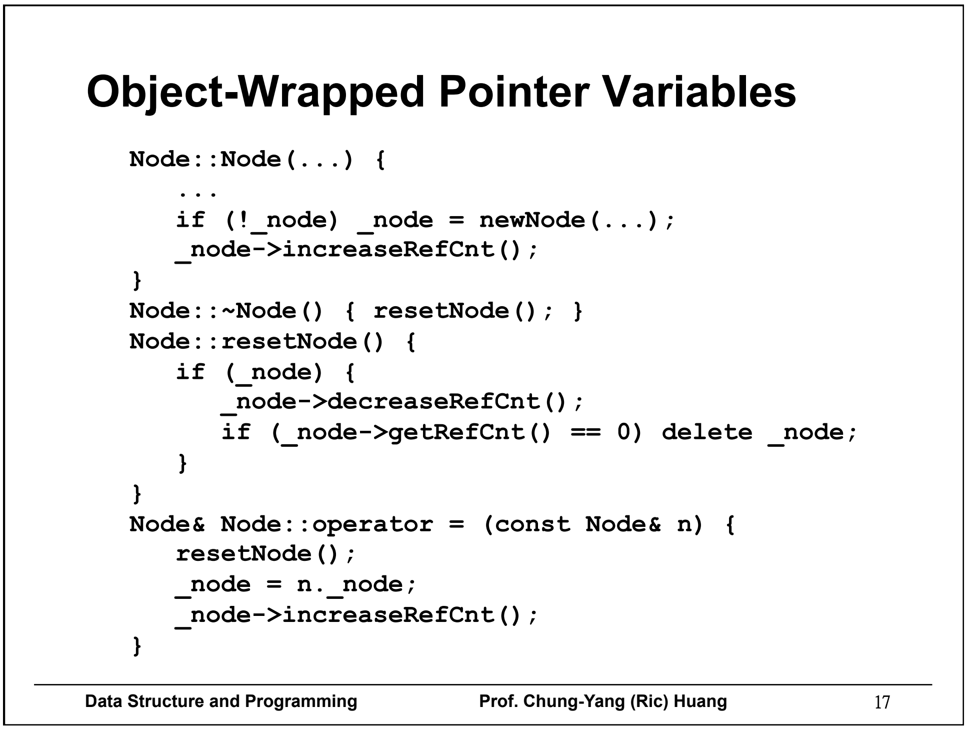
◆Keep the reference count

◆Pointer   NodeInt)

Object   user interface   (e.g. class Node)

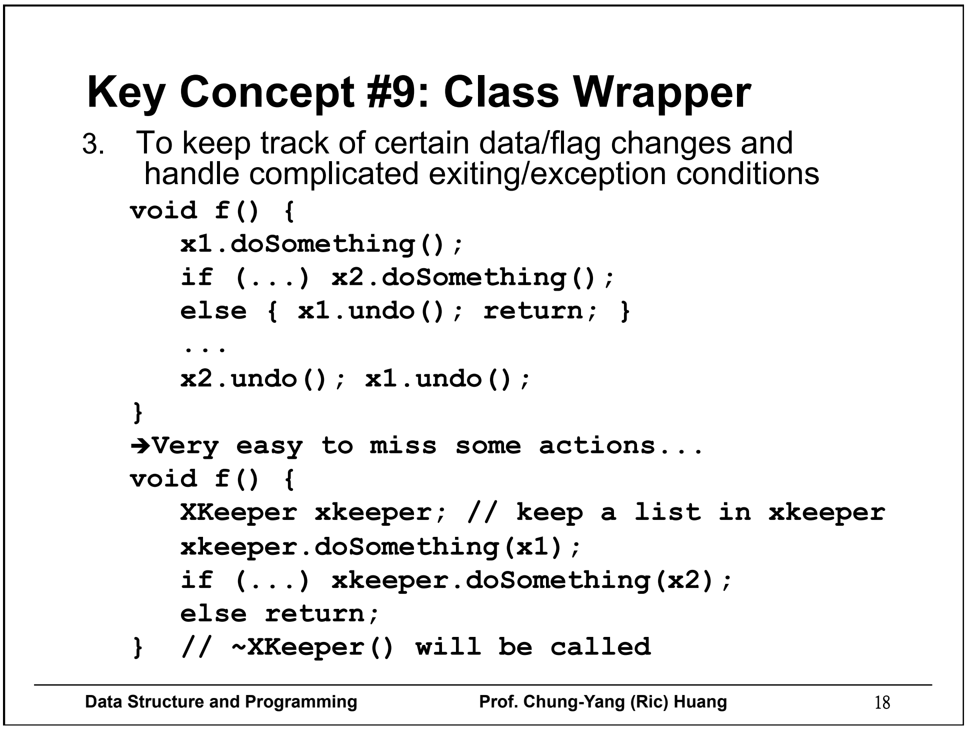
```
class NodeInt {     // a private class
   friend class Node;
   Data    _data;
   Node    _left;
   Node    _right;
   size_t  _refCnt;
};
class Node {
   NodeInt *_node;
};
```





Object-Wrapped Pointer Variables

```
Node::Node(...) {
   ...
   if (!_node) _node = newNode(...);
   _node->increaseRefCnt();
}
Node::~Node() { resetNode(); }
Node::resetNode() {
   if (_node) {
      _node->decreaseRefCnt();
      if (_node->getRefCnt() == 0) delete _node;
   }
}
Node& Node::operator = (const Node& n) {
   resetNode();
   _node = n._node;
   _node->increaseRefCnt();
}
```



Key Concept #9: Class Wrapper

3. To keep track of certain data/flag changes and handle complicated exiting/exception conditions

```
void f() {
   x1.doSomething();
   if (...) x2.doSomething();
   else { x1.undo(); return; }
   ...
   x2.undo(); x1.undo();
}
➔Very easy to miss some actions...
void f() {
   XKeeper xkeeper; // keep a list in xkeeper
   xkeeper.doSomething(x1);
   if (...) xkeeper.doSomething(x2);
   else return;
}  // ~XKeeper() will be called
```

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 determiend
    ...
    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];
  }
  ```

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;
  ```

More about int [] and int*

-
  ```
  int a[10] = { 0 }; // type of a: "int *const"
  int *p = new int[10];
  *a = 10;
  *p = 20;  // OK
  *(a + 1) = 20;
  *(a++) = 30; // Compile error; explained later
  a = p; // Compile error; non-const to const
  p = a; // OK, but memory leak...
  *(p++) = 40; // OK, but what about "delete [] p"?
  int *q = a;
  q[2] = 20;
  *(q+3) = 30;
  *(q++) = 40;  // OK
  delete a; // compile error/warning; runtime crash...
  delete p; // compile OK, but can't delete p (p = a)
  delete []q; // compile OK, but may get fishy result
  ```
- What about:
  ```
  int a = 10; int *p = &a; ... delete p;
  ```

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: Array vs. Pointer

- An array variable represents a “const pointer”
  - int a[10]; 🡐 treating “a” as an “int * const”
    a = anotherArr; // Error; can’t reassign “a”
  - int *p = new int[10];
    p = anotherPointer; // Compile OK, but memory leak?
    p = new int(20);   // also compile OK
- An array variable (the const pointer) must be initialized
  - Recall: “const” variable must be initialized
  - Key: the size of the array must be known in declaration
  - int a[10]; // OK, as the memory address is assigned.
    int a[10] = { 0 }; // Initialize array variable and its content
    int a[ ];   // NOT OK; array size unknown
    int a[ ] = { 1, 2, 3 }; // OK array size determined by RHS

Const pointer vs. pointer to a const

```
◆ int a = 10;
  const int c = 10;
  a = c;  // OK
  c = a;  // NOT OK; even though 10 = 10
◆ int a[10] = { 0 };
  int b[10];
  int *c;
  const int *d;
  int *const e; // Error: uninitialized
  b = a;  // Error
  c = a; d = a; // OK
  e = a;  // Error
◆ void f(const int* i) { ... }
  int main() {
     int * const a = new int(10);
     f(a);  // Any problem?
  }
```

Key Concept #15: Pointer Arithmetic

- ‘+’ / ‘-’ operator on a pointer variable points to the memory location of the next / previous element
  - int *p = new int(10);
    int *q = p + 1; // memory addr += sizeof(int)
  - A *r = new A;
    r -= 2;   // memory addr -= sizeof(A) * 2
- For an array variable “arr”, “arr + i” points to the memory location of arr[i]
  - int arr[10];
    *(arr + 2) = 5;   // equivalent to “arr[2] = 5”

Key Concept #16: const class object (revisited)

- Remember:
  const A& B::blah (const C&   c) const {...}
  - When an object of class B calls this member function, this object will become a “const class object”.
  - That is, the B’s data members will be treated as const (i.e. can’t be modified) in this function.
  - Also, “this” cannot call non-const functions in “blah()”, nor can the data members call non -const functions.

Key Concept #17: 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?

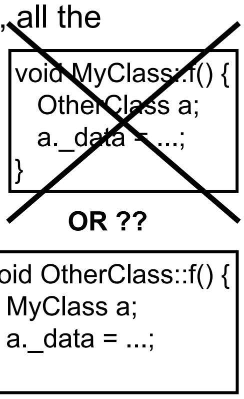
- The key: know the scope you are in!!
  - Class scope:
    1. Inside the definition of the class body “class { };”
    2. In the member function definition, even in a separate .cpp file
- Inside the class scope
  - All the member functions and objects of the same class can access ALL (including private) 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 #18: 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 {`

```
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 = ...;
}
```

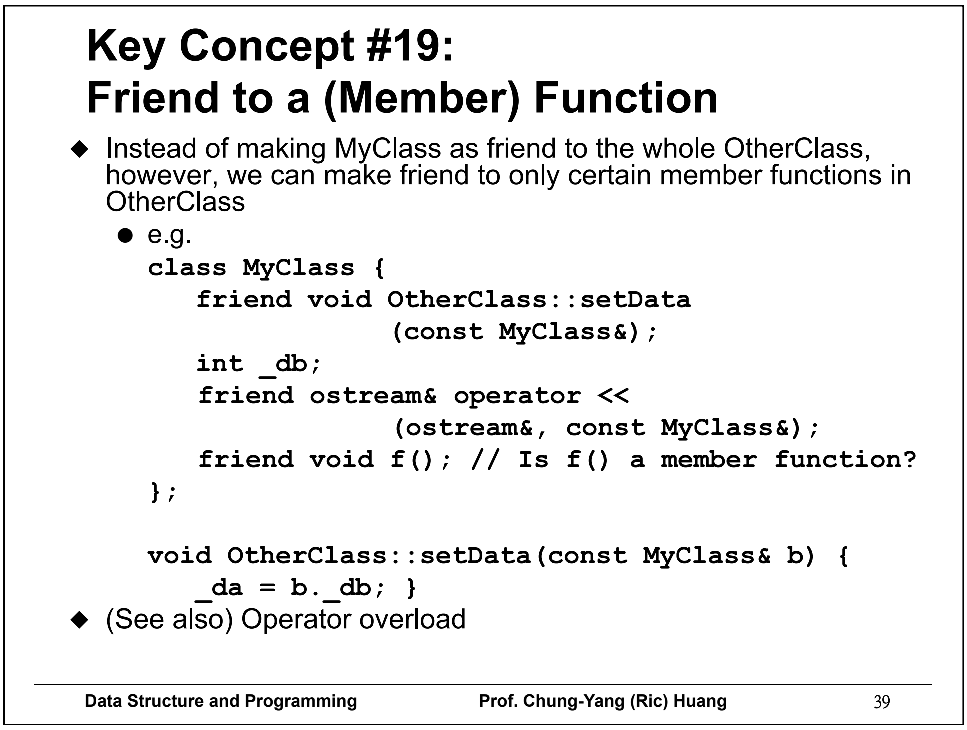
OR ??

```
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); }
};
```



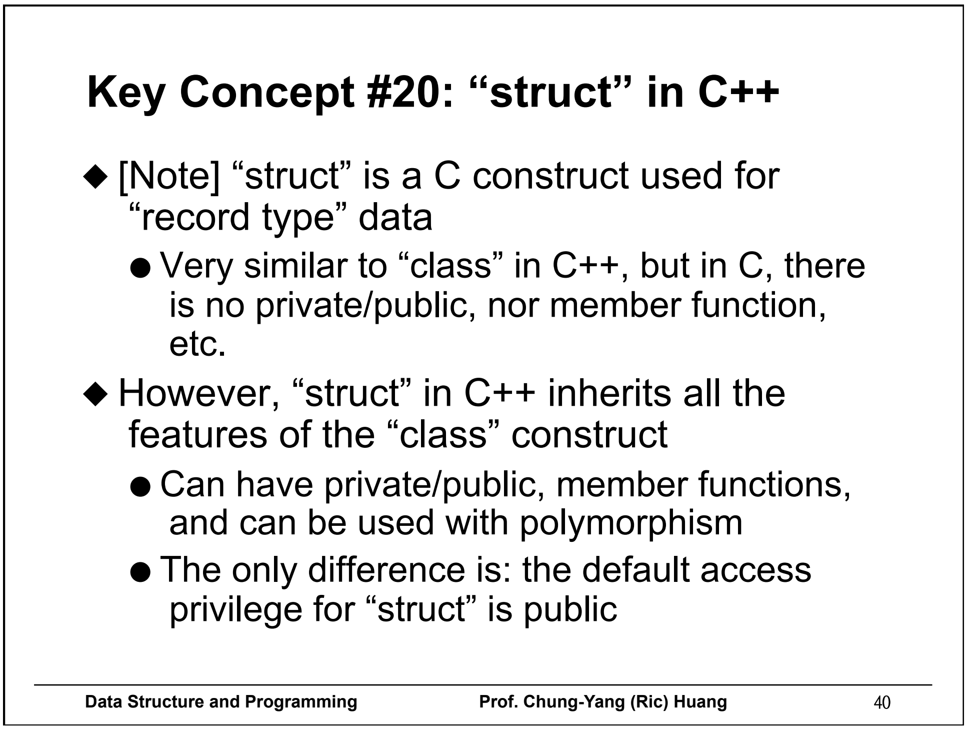
Key Concept #19: Friend to a (Member) Function

- 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



Key Concept #20: “struct” in C++

- [Note] “struct” is a C construct used for “record type” data
  - Very similar to “class” in C++, but in C, there is no private/public, nor member function, etc.
- However, “struct” in C++ inherits all the features of the “class” construct
  - Can have private/public, member functions, and can be used with polymorphism
  - The only difference is: the default access privilege for “struct” is public

Key Concept #21: “union” in C++

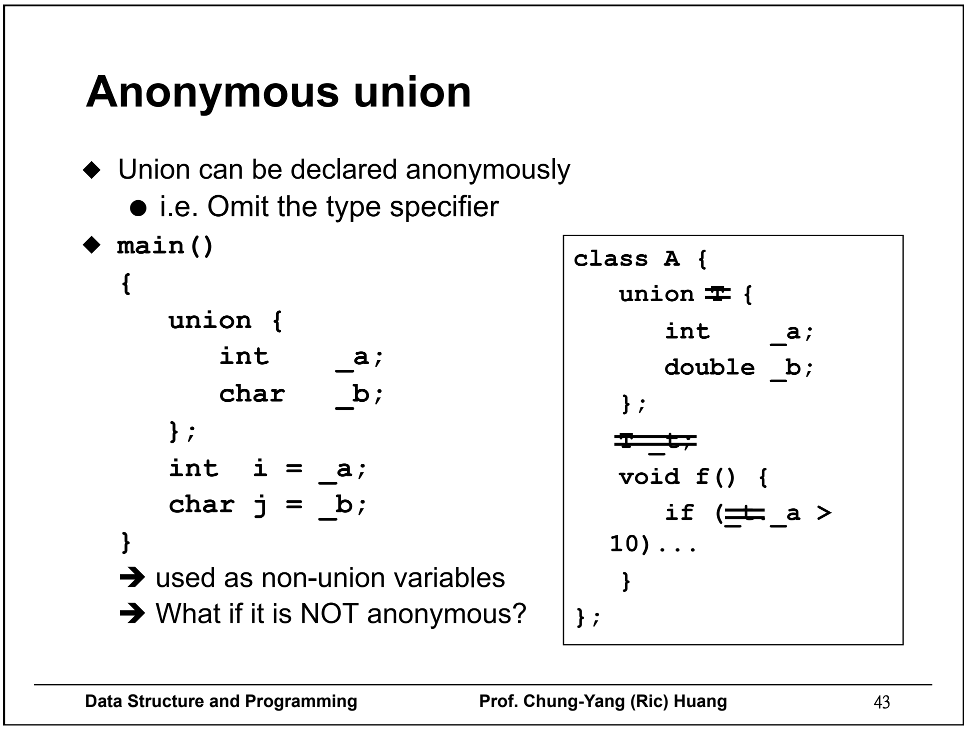
- ◆ At any given time, contains only one of its data members
  - ● To avoid useless memory occupation
  - ● i.e. data members are mutual exclusive
    - ▪ Use “union” to save memory
  - ● size = *max(size of its data members)*
- ◆ A limited form of “class” type
  - ● Can have private/public/protected, data members, member functions
    - ▪ default = public
  - ● Can NOT have inheritance or static data member

Example of “union”

```
union U
{
 private:
   int  _a;
   char _b;
 public:
   U() { _a = 0; }
   int getA() const
       { return _a; }
   void setA(int i)
       { _a = i; }
   char getB() const
       { return _b; }
   void setB(char c)
       { _b = c; }
};
```

```
int
main()
{
   U u;
   u.setB('a');
   cout << u.getA()
        << endl;
   return 0;
}
```

- ◆ What is the output???



Anonymous union

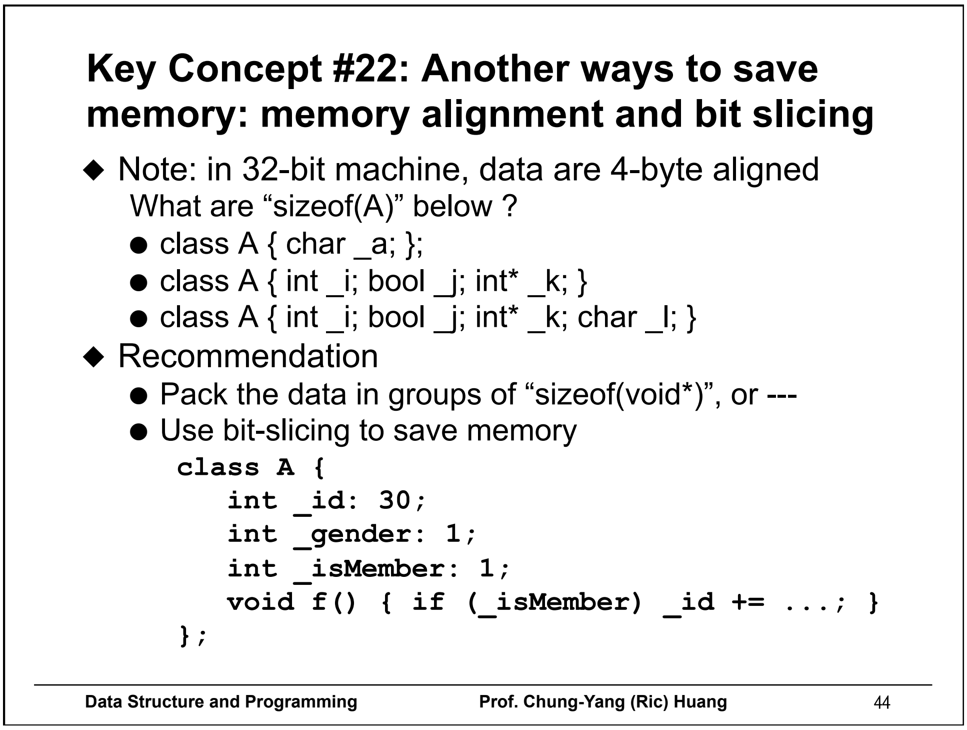
- ◆ Union can be declared anonymously
  - ● i.e. Omit the type specifier
- ◆ `main()`

```
{
   union {
      int    _a;
      char   _b;
   };
   int  i = _a;
   char j = _b;
}
```

➔ used as non-union variables

➔ What if it is NOT anonymous?

```
class A {
   union ~~T~~ {
      int    _a;
      double _b;
   };
   ~~T _t;~~
   void f() {
      if (~~_t.~~_a >
  10)...
   }
};
```



Key Concept #22: Another ways to save memory: memory alignment and bit slicing

- ◆ Note: in 32-bit machine, data are 4-byte aligned

  What are “sizeof(A)” below ?
  - ● class A { char _a; };
  - ● class A { int _i; bool _j; int* _k; }
  - ● class A { int _i; bool _j; int* _k; char _l; }
- ◆ Recommendation
  - ● Pack the data in groups of “sizeof(void*)”, or ---
  - ● Use bit-slicing to save memory

```
class A {
   int _id: 30;
   int _gender: 1;
   int _isMember: 1;
   void f() { if (_isMember) _id += ...; }
};
```

How about bit-slicing for pointers?

- No, size of pointers is fixed. You cannot bit slice them.
- One “tricky” way to save memory is to use the fact that pointer addresses are multiple of 4’s (for 32-bit machines)

```
#define BDD_EDGE_BITS     2
#define BDD_NODE_PTR_MASK
        ((UINT_MAX >>
          BDD_EDGE_BITS) <<
          BDD_EDGE_BITS)
class BddNode {
private:
   size_t       _nodeV;
   // Private functions
   BddNodeInt* getBddNodeInt()
   const {    return
      (BddNodeInt*)(_nodeV &
       BDD_NODE_PTR_MASK); }
```

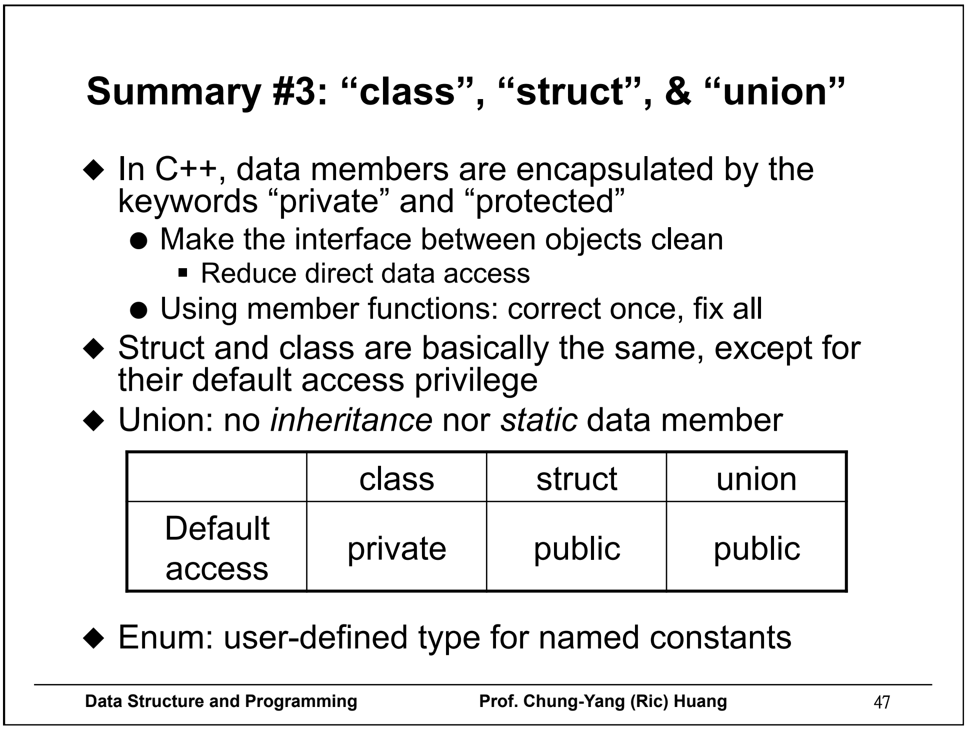
```
   bool isNegEdge() const {
      return (_nodeV &
              BDD_NEG_EDGE); }
};

class BddNodeInt
{
   BddNode        _left;
   BddNode        _right;
   unsigned       _level    : 16;
   unsigned       _refCount : 15;
   unsigned       _visited  : 1;
};
```

A Closer Look at the Previous Example

```
class BddNode {  // wrapper class for BddNodeInt
private:
   size_t       _nodeV;
};
class BddNodeInt {  // as pointer variables
   …
};
```

- Important concepts:
  - No extra memory usage when wrapping a pointer variable with a class
  - However, you gain the advantages in using constructor/destructor, operator overloading, etc, which are not applicable for pointer type variables.
  - The LSBs can be used as flags or stored other information.

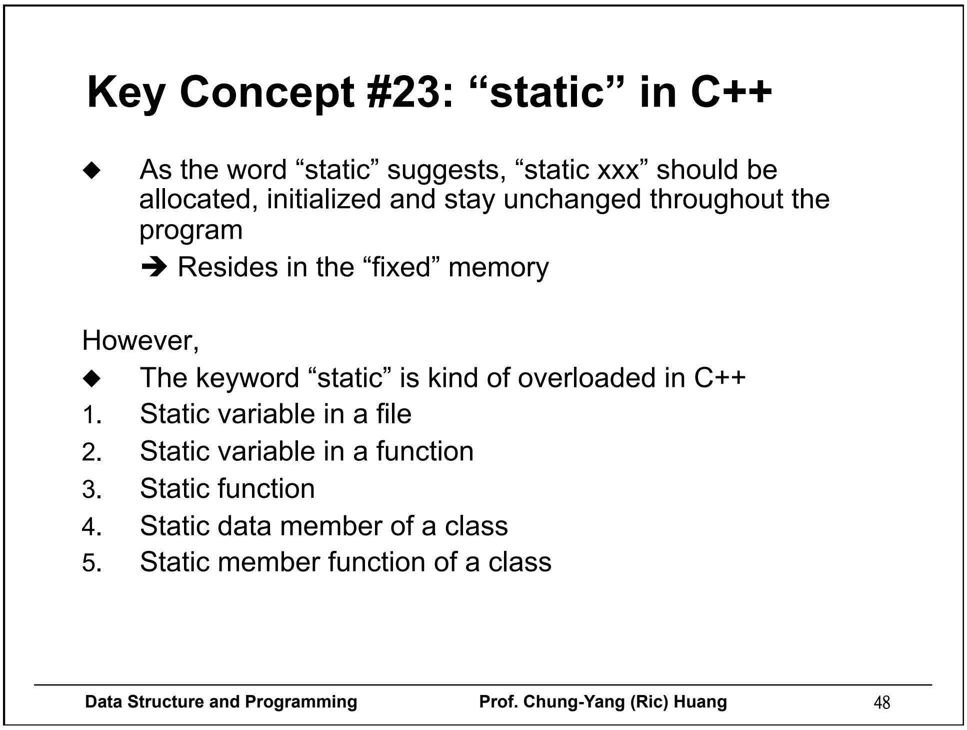


Summary #3: “class”, “struct”, & “union”

- In C++, data members are encapsulated by the keywords “private” and “protected”
  - Make the interface between objects clean
    - Reduce direct data access
  - Using member functions: correct once, fix all
- Struct and class are basically the same, except for their default access privilege
- Union: no *inheritance* nor *static* data member

	class	struct	union
Default access	private	public	public

- Enum: user-defined type for named constants



Key Concept #23: “static” in C++

- As the word “static” suggests, “static xxx” should be allocated, initialized and stay unchanged throughout the program

  ➔ Resides in the “fixed” memory

However,

- The keyword “static” is kind of overloaded in C++

1. Static variable in a file
2. Static variable in a function
3. Static function
4. Static data member of a class
5. Static member function of a class

So, what does “static” mean anyway?

- “static” here,
  refers to “memory allocation” (storage class)
  - The memory of “static xxx” is allocated before the program starts (i.e. in fixed memory), and stays unchanged throughout the program

  [cf] “auto” storage class
    - Memory allocated is controlled by the execution process (e.g. local variables in the stack memory)

Key Concept #24:
Visibility of “static” variable and function

1. Static variable in a file
   - It is a file-scope global variable
   - Can be seen throughout this file (only)
   - Variable (storage) remained valid in the entire execution
2. Static variable in a function
   - It is a local variable (in terms of scope)
   - Can be seen only in this function
   - Variable (storage) remained valid in the entire execution
3. Static function
   - Can only be seen in this file

- Static variables and functions can only be seen in the defined scope
  - Cannot be seen by other files
  - No effect by using “extern”

[Note] Storage class vs. visible scope

- Remember, “static” refers to static “memory allocation” (storage class)
  - We’re NOT talking about the “scope” of a variable
- The scope of a variable is determined by where and how it is declared
  - File scope (global variable)
  - Block scope (local variable)

➔ However, the “static” keyword does constrains the maximum visible scope of a variable or function to be the file it is defined

Key Concept #25: “static” Data Member in a Class

- Only one copy of this data member is maintained for all objects of this class
  - All the objects of this class see the same copy of the data member (in fixed memory)
  - (Common usage) Used as a counter

```
class T
{
   static int _count;
public:
   T() { _count++; }
   ~T() { _count--; }
};
------------------------------------------------
int T::_count=0;
// Static data member must be initialized in some
//     cpp file ==> NOT by constructor!!!  (why?)
```

Key Concept #26: “static” Member Function in a Class

- Useful when you want to access the “static” data member but do not have a class object
  - Calling static member function without an object
    - e.g. T::setGlobalRef();
  - No implicit “this” argument (no corresponding object)
  - Can only see and use “static” data members , enum, or nested types in this class
    - Cannot access other non-static data members
- Usage
  - T::staticFunction(); // OK
  - object.staticFunction(); // OK
  - T::staticFunction() { ... staticMember... } // OK
  - T::staticFunction() { ... this... } // Not OK
  - T::staticFunction() { ... nonStaticMember... } // Not OK
  - T::nonstaticFunction() { ... staticMember... } // OK

Example of using “static” in a class

```
class T
{
   static unsigned   _globalRef;
   unsigned          _ref;

public:
   T() : _ref(0) {}
   bool isGlobalRef(){ return (_ref == _GlobalRef); }
   void setToGlobalRef(){ _ref = _global Ref; }
   static void setGlobalRef() { _globalRef++; }
}
```

- Use this method to replace “setMark()” functions in graph traversal problems
  (How??)

Key Concept #27:
static_cast<T>(a)... Cast away static?? ☹

- ◆ Convert object “a” to the type “T”
  - ● No consistency check (i.e. sizeof(T))
    - ➔ static implies “compile time”
    - ➔ May not be safe
    - ➔ cf. dynamic_cast<T>(a)
  - ● (Common use) // more safer use

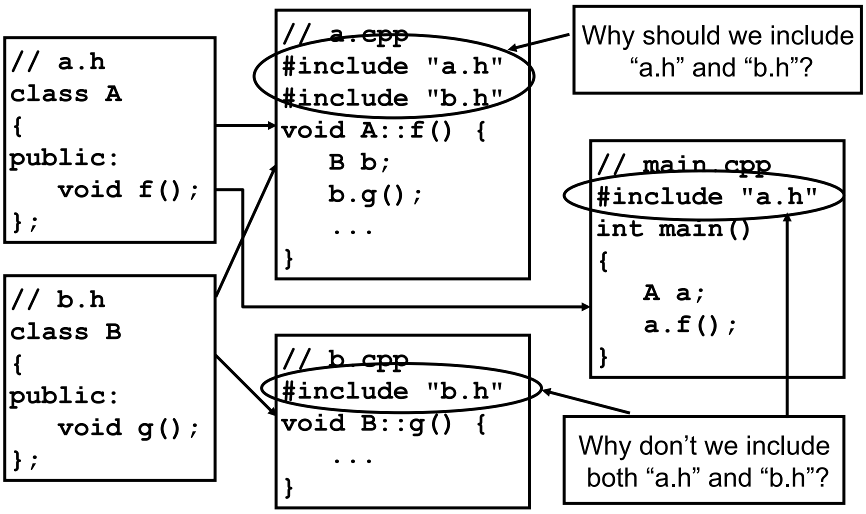
// Parent-class pointer object wants to
//   call the child-only method

```
class Child : public Dad { ... };
----------------------------------
void f()
{
   Dad* p = new Child;
   ...
   static_cast<Child *>(p)->childOnlyMethod();
};
```

Remember in a software project...

- ◆ Your program may have many classes...
- ◆ You should create multiple files for different class definitions ---
  - ● .h (header) files
    ➔ class declaration/definition, function prototype
  - ● .cpp (source) files
    ➔ class and function implementation
  - ● Makefiles
    ➔ scripts to build the project

Key Concept #28: Define classes in header files



```
// a.h
class A
{
public:
   void f();
};
```

```
// b.h
class B
{
public:
   void g();
};
```

```
// a.cpp
#include "a.h"
#include "b.h"
void A::f() {
   B b;
   b.g();
   ...
}
```

```
// b.cpp
#include "b.h"
void B::g() {
   ...
}
```

```
// main.cpp
#include "a.h"
int main()
{
   A a;
   a.f();
}
```

Why should we include “a.h” and “b.h”?

Why don’t we include both “a.h” and “b.h”?

Key Concept #29: “#include”

- A compiler preprocessor
  - Process before compilation
  - Perform copy-and-paste
- This is NOT OK

  -
    ```
    // no #include “b.h”
    class A {
       B   _b;
    };
    ```
- This is OK

  -
    ```
    // no #include “b.h”
    class B; // forward declaration
    class A {
       B   *_b;
    };
    ```

➔ The rule of thumb is “need to know the size of the class”!!

Key Concept #30: #include “ ” or <> ?

- Standard C/C++ header files
  - Stored in a compiler-specified directory
    - e.g. /usr/include/c++/4.1.2
- #include <> will search it in the standard header files
- #include “” will search it in the current directory (‘.’), or the directories specified by “-I” in g++ command line.

Key Concept #31: Undefined or Redefined Issues

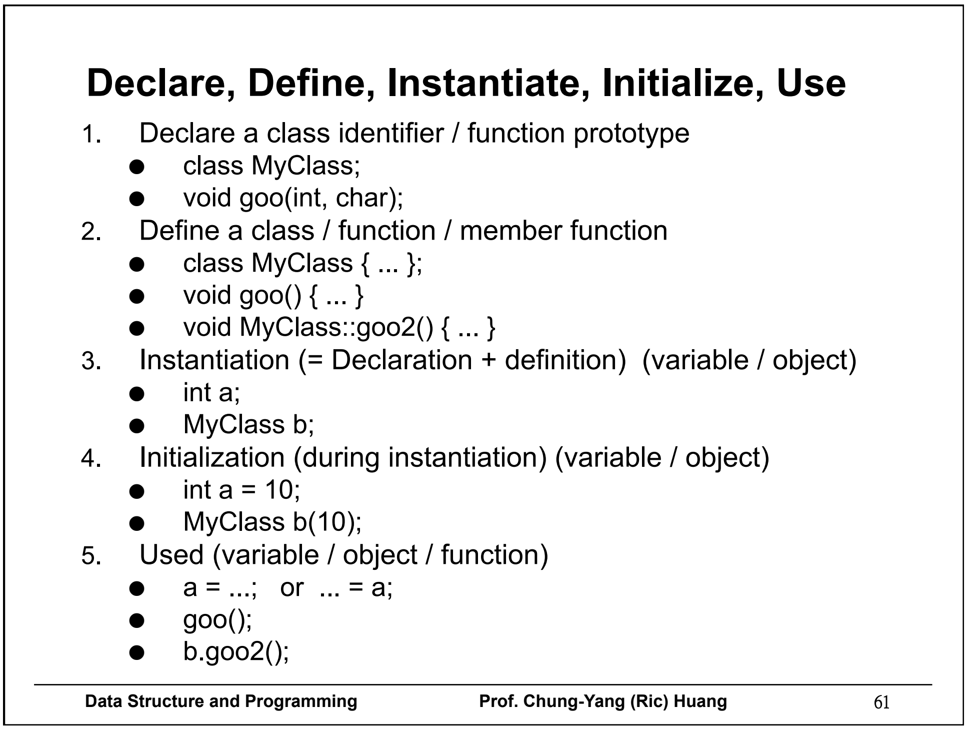
- Undefined errors for variable/class/type/function
  - The following will cause errors in compiling a source file ---

    ```
    int i = j;   // If j is not declared before this point
    A a;   // If class A is not defined before this point
    A *a;   // If class A is not declared before this point
    goo(); // If no function prototype for goo() before this point
    ```

  - The following is OK when compiling each source file, but will cause error during linking --

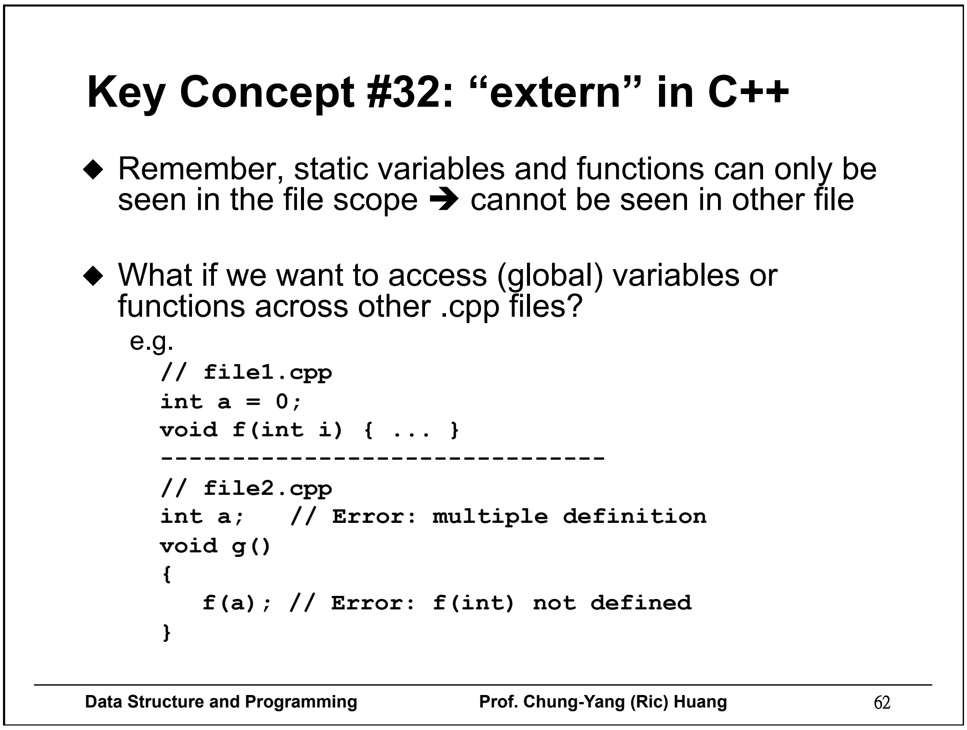
    ```
    int goo(); // forward declaration
    ...
    int b = goo();
    // If goo() is NOT defined in any other source file
    ```

- Redefined errors
  - Variable/class/function is defined in multiple places
  - May be due to multiple inclusions of a header file



Declare, Define, Instantiate, Initialize, Use

1. Declare a class identifier / function prototype
   - class MyClass;
   - void goo(int, char);
2. Define a class / function / member function
   - class MyClass { ... };
   - void goo() { ... }
   - void MyClass::goo2() { ... }
3. Instantiation (= Declaration + definition) (variable / object)
   - int a;
   - MyClass b;
4. Initialization (during instantiation) (variable / object)
   - int a = 10;
   - MyClass b(10);
5. Used (variable / object / function)
   - a = ...;   or ... = a;
   - goo();
   - b.goo2();



Key Concept #32: “extern” in C++

- ◆ Remember, static variables and functions can only be seen in the file scope ➔ cannot be seen in other file
- ◆ What if we want to access (global) variables or functions across other .cpp files?

e.g.

```
// file1.cpp
int a = 0;
void f(int i) { ... }
-------------------------------
// file2.cpp
int a;   // Error: multiple definition
void g()
{
   f(a); // Error: f(int) not defined
}
```

Using External Variables and Functions

e.g.

```
// file1.cpp
int a = 0;
void f(int i) { ... }
-------------------------------
// file2.cpp
extern int a; // a is an external variable
void f(int);  // f() is an external function
              // "extern" can be omitted here
void g()
{
   f(a);
}
```

Key Concept #33: Forward Declaration

[Bottom line]

Sometimes we just want to include part of the header file, or refer to some declarations

➔ We don't want to include the whole header file

➔ To reduce:

1. Executable file size
2. Compilation time due to dependency

e.g.

```
// MyClass.h
class HisClass;   // forward declaration
class HerClass;   // forward declaration
class MyClass
{
   HisClass*  _hisData;  // OK
   HerClass   _herData;  // NOT OK; why?
};
```

Key Concept #34: Namespace

- e.g.

- namespace MyNS = MyNameSpace; // alias
- Must declare in global scope

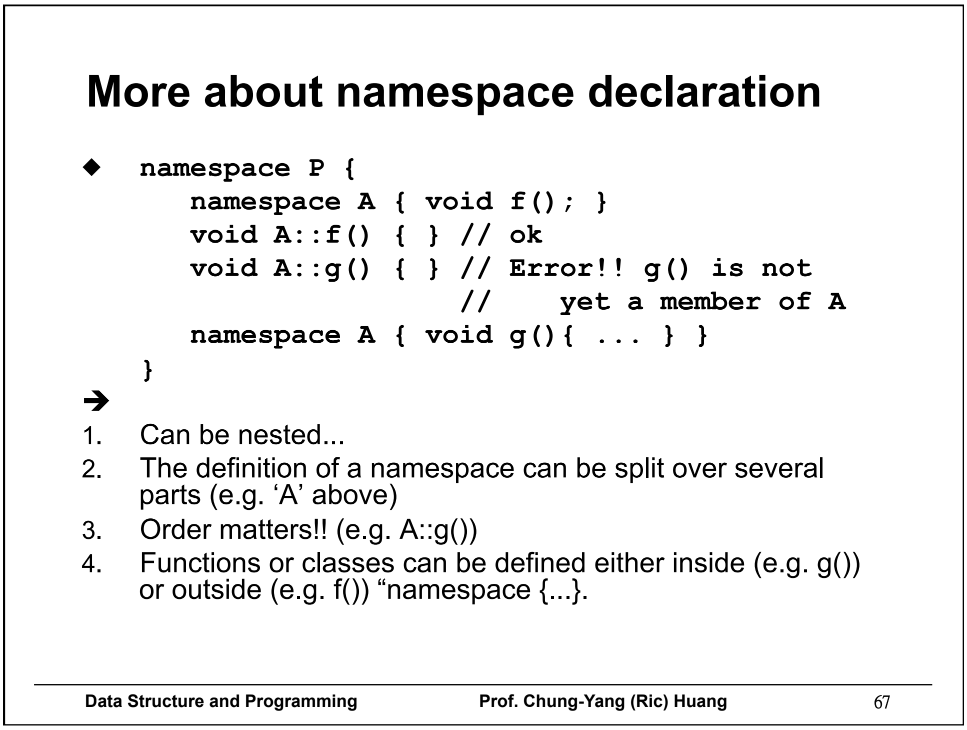
  -
    ```
    int main()
    {
       namespace XYZ { ... }   // Error!!
    }
    ```

Using namespace

```
1. void g() {
      MyNameSpace::a = 10;
   }  // "::" is the scope operator

2. using MyNameSpace::a;
   void g() {
      a = 10;
   }

3. using namespace MyNameSpace;
   void g() {
      a = 10;
      f();
   }
```

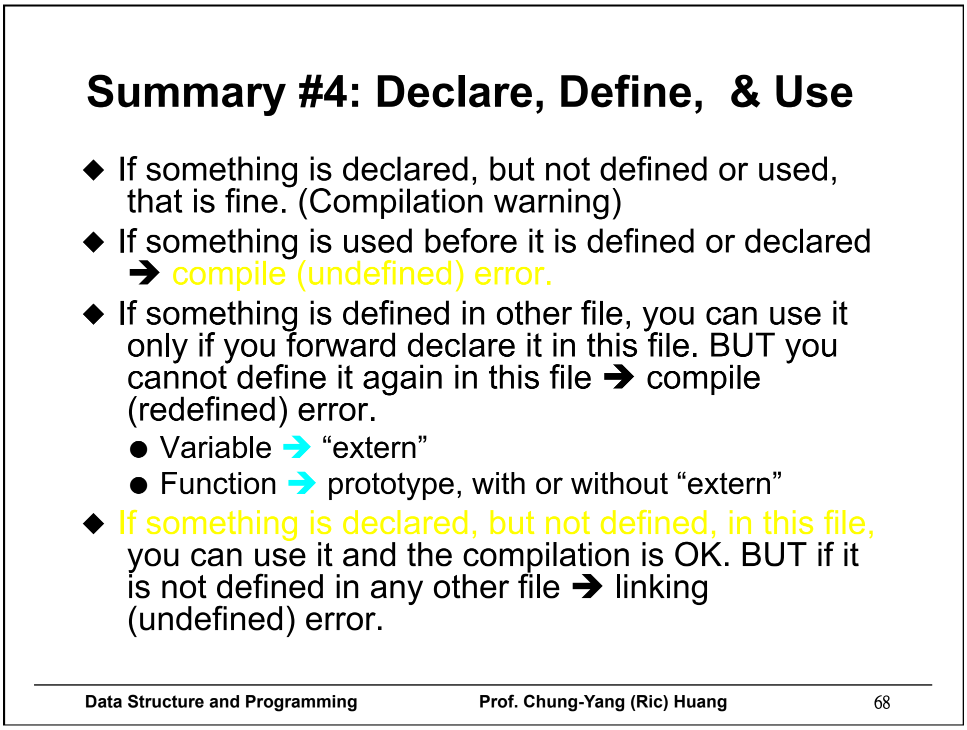


More about namespace declaration

◆
```
namespace P {
   namespace A { void f(); }
   void A::f() { } // ok
   void A::g() { } // Error!! g() is not
                   //    yet a member of A
   namespace A { void g(){ ... } }
}
```

➔

1. Can be nested...
2. The definition of a namespace can be split over several parts (e.g. 'A' above)
3. Order matters!! (e.g. A::g())
4. Functions or classes can be defined either inside (e.g. g()) or outside (e.g. f()) "namespace {...}.



Summary #4: Declare, Define, & Use

- If something is declared, but not defined or used, that is fine. (Compilation warning)
- If something is used before it is defined or declared ➔ compile (undefined) error.
- If something is defined in other file, you can use it only if you forward declare it in this file. BUT you cannot define it again in this file ➔ compile (redefined) error.
  - Variable ➔ "extern"
  - Function ➔ prototype, with or without "extern"
- If something is declared, but not defined, in this file, you can use it and the compilation is OK. BUT if it is not defined in any other file ➔ linking (undefined) error.

Key Concept #35: #define

- #define is another compiler preprocessor
  - All the compiler preprocessors start with “#”
- “#define” performs pre-compilation inline string substitution
- “#define” has multiple uses in C++
  1. Define an identifier (e.g. #define NDEBUG)
  2. Define a constant (e.g. #define SIZE 1024), or substitute a string
  3. Define a function (Macro)

“#define” for an Identifier

1. To avoid repeated definition of a header file in multiple C/C++ inclusions
   -
     ```
     #ifndef MY_HEADER_H
     #define MY_HEADER_H
     // header file body...
     // ...
     #endif
     ```
2. Conditional compilation
   -
     ```
     #ifndef NDEBUG
     // Some code you want to compile by default
     // (i.e. debug mode)
     // For optimized mode,
     // define “NDEBUG” in Makefile.
     #endif
     ```

“#define” for a Constant or a String

- #define <identifier> [tokenString]
  - e.g.

```
#define SIZE          1024
#define CS_DEFAULT    true
#define HOME_DIR      “/home/ric”
             (why not /home/ric?)
```

- Advantage of using “#define”
  - Correct once, fix all
- What’s the difference from “const int xxx”, etc?
  - Remember: “#define” performs pre-compilation inline string substitution
  - “const int xxx” is a global variable
    ➔ Fixed memory space
    ➔ Better for debugging!!

“#define” for a MACRO function

- #define <identifier>(<argList>) [tokenString]
  - e.g.

```
#define MAX(a, b)   ((a > b)? a: b)
            // Why not “((a > b)? a: b)” ?
```

  - e.g.

```
// Syntax error below!! Why??
#define MAX(int a, int b) ((a > b)? a: b)
```

- Disadvantage
  - “#define” MACRO function is difficult to debug!!
    ➔ Cannot step in the definition (Why??)
  - Use inline function (i.e. inline int max(int a, int b)) instead

Key Concept #36: Enum

- A user-defined type consisting of a set of named constants called enumerators
  - e.g.

```
class T {
   enum COLOR {
      RED,        // value = 0
      BLUE,       // value = 1
      GREED = 5,
      YELLOW      // value = 6
   };
};
```

- By default, first enumerator's value = 0
- Each successive enumerator is one larger than the value of the previous one, unless explicitly specified (using "=") with a value

Scope of "enum"

- Enumerators are only valid within the scope it is defined
  - e.g.

```
class T {
  enum COLOR { RED, BLUE };
};
```

  ➔ RED/BLUE is only seen within T

  - To access enumerator outside of the class, use explicit class name qualification
    - e.g. void f() { int i = T::RED; }

  ➔ But the enum must be defined as *public*

Common usage of “enum”

1. Used in function return type
   - Color getSignal() { ... }
2. Used as “status” and controlled by “switch-case”

   -
     ```
     ProcState f() { ...; return ...; }
     ...
     ProcState state = f();
     switch (state) {
        case IDLE   : ...; break;
        case ACTIVE: ...; break;
     }  // What’s the advantage??
     ```
3. Used as “bit-wise” mask

Bitwise Masks

- To manipulate multiple control “flags” in a single integer
- **enum ErrState {**

```
   NO_ERROR  = 0,
   DIV_ZERO  = 0x1, // 001
   OVERFLOAT = 0x2, // 010
   INTERRUPT = 0x4, // 100
   BAD_STATUS= DIV_ZERO | OVERFLOAT | INTERRUPT
};
int ~~ErrState~~ status = NO_ERROR; // This line is OK
// To set the error status
status |= OVERFLOAT;
// To unset the error status
status &= ~DIV_ZERO;
// To test the error status
if ((status & INTERRUPT) != 0)
   ...
➔ Compilation error... WHY???
```

Key Concept #37: “#define” vs. “enum”

```
1. #define RED    0
   #define BLUE   1
   #define GREEN  5
2. enum COLOR {
      RED,       // value = 0
      BLUE,      // value = 1
      GREED = 5
   };
```

- What’s the difference in terms of debugging?
  - Using “#define” ➔ Can only display “values”
  - Using “enum” ➔ Can display “names”

  Recommendation: using “enum”

What we have learned in this lecture note?

- What is “class”?
- Constructor, destructor
- new, new [], delete, delete []
- A*, A**, A***....
- Access privilege: private/protected/public
- Friend
- Class, struct, union
- “static”, “extern”
- Namespace
- #include, #define, enum