

Topic 2

C++ Review Part II: More on Functions, Variables, Classes

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

09.26.2018

Part I: Understanding “Functions”

- ◆ Global vs. member functions
- ◆ Function signature, prototype , definition
- ◆ Function parameters, arguments

Key Concept #1: Global vs. Member Functions

- ◆ Global functions are defined in global scope
 - `void f(...) { ... }`
 - There is no so-called local functions
- ◆ Member functions are defined in class scope
 - `void A::f(...) { ... }`
 - A member function is called by an object of its class type

Key Concept #2: Function Signature

- ◆ 4 things to define “function signature”
 1. Function name
 2. Number of parameters
 3. Types of parameters
 4. Order of parameters→ No “return type” (why?)
- ◆ There cannot be functions with the same function signature, unless —
 1. Separated by different name spaces
 2. Defined as “static” in different file scopes
- ◆ However, functions can have the same name, but different signature (overloading)

Key Concept #3: Function Prototype vs. Function Definition

- ◆ Think, which one is better?

```
1. void f() {  
    ...  
}  
int main() {  
    ...  
    f();  
}  
2. void f(); ← function prototype  
int main() {  
    ...  
    f();  
}  
void f() {  
    ...  
} ← function definition
```

Key Concept #4: Default Argument

- ◆ Note:
 - void f(int x) { ... } // x is f's parameter
 - f(10); f(a); // 10, a as arguments to f()
- ◆ Parameters with default assignments → function with default arguments
 - Can be skipped when calling the function
 - e.g.
void f(int x, int y = 0);
f(10);
 - Can only appear towards the end of parameter list
 - (Not OK) void f(int x = 0, int y);
- ◆ Given a function, its default argument can only be defined ONCE
 - void f(int x = 0);
void f(int x = 0) { ... } → Compilation ERROR

Key Concept #5: Parameters in a function

- ◆ When a function is called, the caller performs “=” operations on its arguments to the corresponding parameters in the function

```
• void f(int a, char c, int *p) { ... }  
...  
int main() {  
    f(i, cc, pp); // int a = i;  
                  // char c = cc;  
                  // int *p = pp;  
}
```

Key Concept #6: Passed by Object, Pointer, and Reference

- ◆ // passed by object
void f(int a) { ... }
int main() { int b; ...; f(b); }
- ◆ // passed by object
void h(A a) { ... }
int main() { A aa; ...; f(aa); }
- ◆ // passed by pointer
void g(int *p) { ... }
int main() { int *q = ...; f(q); }
- ◆ // passed by reference
void k(A& a) { ... }
int main() { A aa; ...; k(aa); }

Passed by Object, Pointer, and Reference

[Rule of thumb] Making an '=' (i.e. copy) from the passed argument in the caller, to the parameter of the called function.

<pre>void f1(int a) { a = 20; } void f2(int& a) { a = 30; } void f3(int* p) { *p = 40; } void f4(int* p) { p = new int(50); } void f5(int* &p) { p = new int(60); }</pre>	<pre>main() { int a = 10; int* p = &a; int a1,a2,a3,a4,a5; f1(a); a1 = a; f2(a); a2 = a; f3(p); a3 = *p; f4(p); a4 = *p; f5(p); a5 = *p; }</pre>
---	--

What are the values of a1, a2, a3, a4, and a5 at the end?

Summary #1:

Passed by pointer or passed by reference

1. If you have some data to share among functions, and you don't want to copy (by '=') them during function calling, you can use "passed by pointers"

```
class A {
    int _i; char _c; int *_p; ...
};
void f(A *a) { ... }
...
int main() {
    A *a = ...;
    f(a);
}
```

Summary #1:

Passed by pointer or passed by reference

2. However, if originally the data is not a pointer type, "passed by pointers" is kind of awkward. You should use "passed by references"

```
class A {
    int _i; char _c; int *_p; ...
};
void f(A *a) { ... }
void g(A& a) { ... }
...
int main() {
    A a = ...; // an object, not a pointer
    f(&a);      // Awkward!! C style ☹
    g(a);       // Better!!
}
```

Summary #1:

Passed by pointer or passed by reference

3. But, sometimes we just want to share the data to another function, but don't want it to modify the data.

```
int main() {
    A a = ...;
    g(a);
}

void g(A& a) { ... }
// "a" may get modified by g()
→ Using "const" to constrain !!
```

Part II: More on “Variables”

- ◆ “const” keyword
- ◆ Array vs. pointers
- ◆ Pointer arithmetic
- ◆ Memory sizes of variables
- ◆ Return value of a function
- ◆ Compilation issues
- ◆ Compiler preprocessors

Key Concept #1: Const

- ◆ Const is an adjective
 - When a variable is declared “const”, it means it is “READ-ONLY” in that scope.
 - ➔ Cannot be modified
- ◆ Const must be initialized
 - `const int a = 10;` // OK
 - `const int b;` // NOT OK
 - `int i;` // Not initialized...
 - `const int j = i;` // Is this OK?
 - `const int& k = i;` // Is this OK?
 - `f(j);` // `f(int m) { ... }`; Is this OK?
 - `i = 10;` // will j, k be changed? Is this OK?
- ◆ “const int” and “int const” are the same
- ◆ “const int *” and “int * const” are different !!

What? const *&#\$&@%#q

- ◆ Rule of thumb
 - Read from right to left
- 1. `f(int* p)`
 - Pointer to an int (integer pointer)
- 2. `f(int*& p)`
 - Reference to an integer pointer
- 3. `f(int*const p)`
 - Constant pointer to an integer
- 4. `f(const int* p) = f(int const * p)`
 - Pointer to a constant integer
- 5. `f(const int*& p)`
 - Reference to a pointer of a constant int
- 6. `f(const int*const& p)`
 - Reference to a constant pointer address, which points to a constant integer

Passed in a reference to a constant object 'c'
➔ 'c' cannot be modified in the function

`const A& B::blah(const C& c) const {...}`

Return a reference to a
constant object
➔ The returned object
can then only call
constant methods

This is a constant method,
meaning this object is treated as
a constant during this function
➔ None of its data members
can be modified

Key concept #2: The Impact of const

- ◆ Supposed “_data” is a data member of class MyClass

```
void MyClass::f() const
{
    _data->g();
}
```

 - Because this object is treated as a constant, its data field “_data” is also treated as a constant in this function
→ “g()” must be a constant method too!!
 - Compiler will signal out an error if g() is NOT a const method
- ◆ [Coding tip] If we really want a member function to be a read-only one (e.g. getXX()), putting a “const” can help ensure it

Const vs. non-const??

- ◆ Passing a non-const argument to a const parameter in a function

```
void f(const A& i) { ... }
void g(const A j) { ... }

int main() {
    A a; ...
    f(a); // a reference of “a” is treated const in f()
    g(a); // a copy of “a” is treated const in g()
}
```

Const vs. non-const??

- ◆ Passing a const argument to a non-const parameter in a function

```
void f(A& i) { ... }
void g(A j) { ... }

int main() {
    const A a(...);
    f(a); // Error → No backdoor for const
    g(a); // a copy of “a” is treated non-const in g()
}
```

Const vs. non-const??

- ◆ Non-const object calling a const method

```
T a;
a.constMethod(); // OK
```

 - “a” will be treated as a const object within “constMethod()”
- ◆ Const object calling non-const method

```
const T a;
a.nonConstMethod(); // not OK
```

 - A const object cannot call a non-const method
→ compilation error

Casting “const” to “non-const”

```
const T a;  
a.nonConstMethod(); // not OK  
Trying...
```

1. T(a).nonConstMethod();
 - Static cast; OK, but may not be safe (why?)
 - Who is calling nonConstMethod()?
2. const_cast<T>(a).nonConstMethod();
 - Compilation error!!
 - “const_cast” can only be used for pointer, reference, or a pointer-to-data-member type
3. const_cast<T*>(&a)->nonConstMethod();
 - OK, but kind of awkward

const_cast<T>() for pointer-to-const object

```
const T* p;  
p->nonConstMethod(); // not OK
```

➔ `const_cast<T*>(p)->nonConstMethod();`
A const object can now call non-const method

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 #3:

“mutable” --- a back door for const method

- ◆ However, sometimes we MUST modify the data member in a const method
 - `void MyClass::f() const`

```
{  
    _flags |= 0x1; // setting a bit of the _flags  
}
```
 - In such case, declare “_flag” with “mutable” keyword
 - e.g.
`mutable unsigned _flag;`

Key Concept #4: 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, but memory leak !
- ◆ 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; // This is OK!
  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?
  }
```

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 runtime crash (p = a)
  delete []q; // compile OK, but may get fishy result
◆ What about:
  int a = 10; int *p = &a; ... delete p;
```

Key Concept #5: Pointer Arithmetic

- ◆ ‘+’ / ‘-’ operator on a pointer variable points to the memory location of the next / previous element (as in an array)
 - `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”

(Recapped) Key Concept #6: 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 #7: 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 #8: Memory Alignment

- ◆ What are the addresses of these variables?

```
int *p = new int(10); // let addr(p) = 0x7ffe84ff0e0
char c = 'a';
int i = 20;
int *pp = new int(30);
char cc = 'b';
int *ppp = pp;
int ii = 40;
char ccc = 'c';
char cccc = 'd';
int iii = 30;
```
- Given a variable of predefined type with memory size S (Bytes), its address must be aligned to a multiple of S

Key Concept #9: Return value of a function

- ◆ Every function has a return type. At the end of the function execution, it must return a value or a variable of the return type.
 - “void f()” means no return value is needed
- 1. Return by object
 - ```
MyClass f(...) {
 MyClass a; ...; return a; }
MyClass b = f(...);
MyClass& c = f(...);
// What's the diff? Is it OK?
// The referenced object must have a
// valid memory addr outside f()
```



## Return by Object, Pointer, and Reference

2. Return by pointer
  - `MyClass* f(...) { MyClass* p; ...; return p; }`
  - `MyClass* q = f(...);`  
// Should we "delete q" later?
3. Return by reference (reference to whom?)
  - `MyClass& f(...) { ...; return r; }`  
// r cannot be local (why?)  
`MyClass& s = f(...);` // <-----|  
`MyClass t = f(...);` // What's the diff?  
// Is it OK?
  - [NOTE] Should NOT return the reference of a local variable  
→ `int& f() { int a; ...; return a; }`  
→ compilation warning
  - `MyClass& MyClass::f(...)`  
`{ ...; return (*this); }`  
`MyClass s;`  
`MyClass& t = s.f(...);` // <-----|  
`MyClass v = s.f(...);` // What's the diff?

## When is "return by reference" useful?

```

◆ template<class T> class Array
{
public:
 Array(size_t i = 0) { _data = new T[i]; }
 T& operator[] (size_t i) { return _data[i]; }
 const T& operator[] (size_t i) const {
 return _data[i]; }
 Array<T>& operator= (const Array& arr) {
 ... return (*this); }
private:
 T *_data;
};

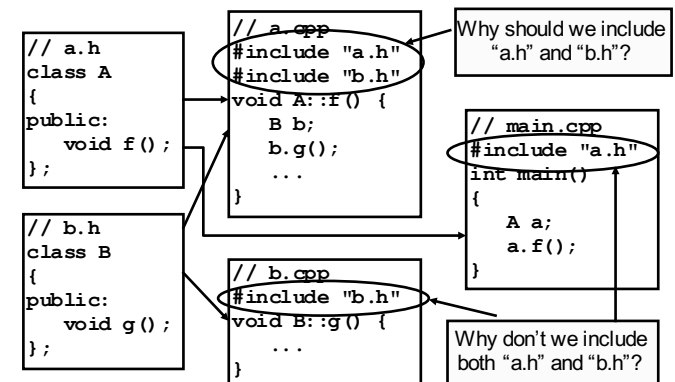
int main()
{
 Array<int> arr(10); // declare an array of size 10
 int t = arr[5]; // <-----|
 arr[0] = 20; // Which one will be called?
 Array<int> arr2; arr2 = arr;
} // Why not "Array<int> arr2 = arr;"?

```

## 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 #10: Define classes in header files



## Key Concept #11: "#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 #12: #include " " or <> ?

- ◆ Standard C/C++ header files
  - Stored in a compiler-specified directory
    - e.g. `/usr/local/include/c++/8.2.0/`
- ◆ #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 #13: 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 if `goo()` is NOT defined in any other source file --

```
int goo(); // forward declaration
...
int b = goo();
```
- ◆ 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 #14: "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 during linking
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 #15: 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 #16: Namespace

◆ e.g.

```
namespace MyNameSpace {
 int a;
 void f();
 class MyClass;
} // Note: no ';'
```

◆ 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 #2: 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 #17: #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

## Part III: More on “Classes”

- ◆ Class, struct, union, enum
- ◆ Bit-slicing
- ◆ Class wrapper
- ◆ “static” keyword

## Key Concept #1: “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 #2: “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 A{
 int _a;
 double _b;
 };
 A
 void f() {
 if (A_a >
 10) ...
 }
};
```

### Key Concept #3: Another ways to save memory: memory alignment and bit slicing

- ◆ Note: in 64-bit machine, data are 8-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 += ...; }
};
```

Data Structure and Programming

Prof. Chung-Yang (Ric) Huang

62

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 8's (for 64-bit machines)

```
#define BDD_EDGE_BITS 3
#define BDD_NODE_PTR_MASK
    ((~(size_t(0)) >>
      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;
    size_t _level : 32;
    size_t _refCount : 31;
    size_t _visited : 1;
};
```

Data Structure and Programming

Prof. Chung-Yang (Ric) Huang

63

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.  
→ `BddNode a, b, c; ... ; c = a & b;`
    - The LSBs can be used as flags or stored other information.

Data Structure and Programming

Prof. Chung-Yang (Ric) Huang

65

### Summary #2: “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

Data Structure and Programming

Prof. Chung-Yang (Ric) Huang

66

## Key Concept #4: 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 in this case, 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
    OVERFLOW = 0x2, // 010
    INTERRUPT = 0x4, // 100
    BAD_STATUS= DIV_ZERO | OVERFLOW |
    INTERRUPT
};
int ErrState status = NO_ERROR; // This line is OK
// To set the error status
status |= OVERFLOW;
// To unset the error status
status &= ~DIV_ZERO;
// To test the error status
if ((status & INTERRUPT) != 0)
    ...
→ Compilation error... WHY???
```


Key Concept #5: “#define” vs. “enum”

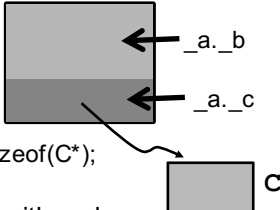
```
1. #define RED    0
   #define BLUE  1
   #define GREEN  5
2. enum COLOR {
    RED,          // value = 0
    BLUE,         // value = 1
    GREED = 5
};
```

- ◆ What's the difference in terms of debugging?
 - Using “#define” → Can only display “values”
 - Using “enum” → Can display “names”
- Recommendation: using “enum”

Recall: 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 #6: 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 #6: 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 should 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 → internal class (e.g. class NodeInt)

Object → user interface (e.g. class Node)

class NodeInt { // a private class

friend class Node;

Data _data;

Node _left;

Node _right;

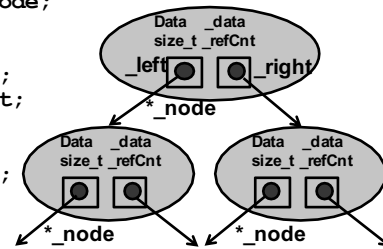
size_t _refCnt;

};

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 #6: 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
```

Key Concept #7: “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 #8:

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 constrain the maximum visible scope of a variable or function to be the file it is defined

Key Concept #9:

“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 #10: “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 #11: `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 Dad { ... };
class Child : public Dad { ... };

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