Topic 3 (Part I: Variables)

C++ advanced features review: *when can/should I use them?*

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

Sep, 2011

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

- Object, pointer, reference
- Const, static, extern, type cast
- Namespace

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)
  - The name of the variable
    ➔ NOT part of the program.
    Used by compiler to associate the assignments and operations of the vaiable
    ➔ For ease of programming and debugging
  - The type of the variable
    ➔ To determine the "size" of the memory

Key Concept #2: '=' operator

- '=' operator in C/C++ performs "assignment", not "equal to"
  - Assignment := copy the value of the right hand side expression to the location of the left hand side variable
    - `a = b + c;`

    ➔ Where is the result of "b+c" stored?
  - What about:
    - ```
      int *p = q;
      int *r = new int(10);
      ```
      ```

Key Concept #3: 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 variable
  - Different types: "int" vs. "int *"

Key Concept #4: 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
- 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()

Summary #1: Types of Variables

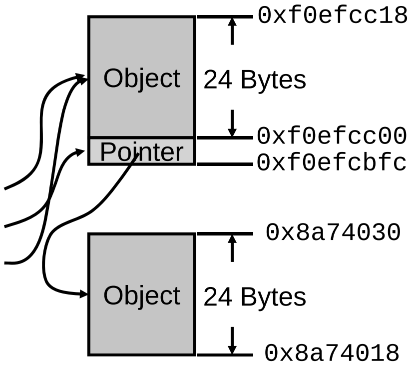
1. Object type
   - int i = 10;
   - MyClass data;
2. Pointer type
   - int* i = new int(10);
   - MyClass* data = new MyClass(“ric”);
3. Reference type
   - int& i = j;
   - MyClass& data = origData;

Object, Pointer, Reference?

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

◆ Symbol table

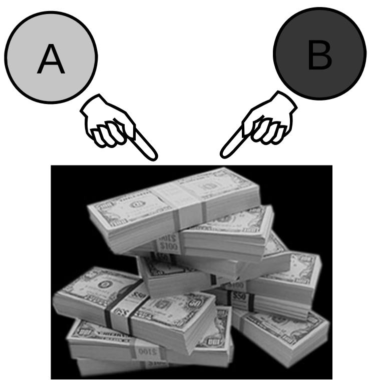
name	address
aaa	0xf0efcc00
ppp	0xf0efcbfc
rrr	0xf0efcc00



Can you answer this...

◆Why do we need “pointer” in C/C++?

“Share” !!

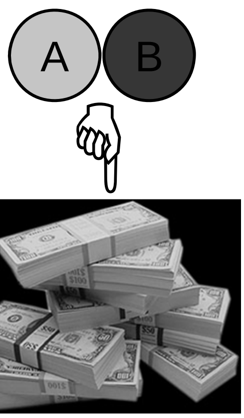



```
compared:
int a = 10;
int b = a;
b += 10;
```

Can you answer this...

◆Why do we need "reference" in C/C++?

"Share" and "Clone"!!



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?
- 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?
  ```

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;
}
```


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.

```
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); }
```

```
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;
}
```

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

Summary #2: Called by pointers; called by references

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 "call by pointers"

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


Summary #2: Called by pointers; called by references

2. However, if originally the data is not a pointer type, "called by pointers" is kind of awkward. You should use "called 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 #2: Called by pointers; called by references

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 = ...;
   f(&a);
   g(a);
}
```

// a may get modified by f() or g()

➔ Using "const" to constrain !!

Key Concept #6: 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 a;
    ...
    const int b = a; // Is this OK?
    const int& c = a; // 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

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 this kind of inconsistency

- If we really want the function "f()" to be a read-only one, 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 int& i) { ... }
void g(const int j) { ... }
int main() {
   int 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(int& i) { ... }
void g(int j) { ... }
int main() {
   const int 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

“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 #7: 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 to initialize the return type.
  - “void f()” means no return in needed

1. Return by object
   - ```
     MyClass f(...) {
        MyClass a;...; return a; }
     MyClass b = f(...);
     MyClass& c = f(...);
     // What’s the diff? Is it OK?
     ```
     ```

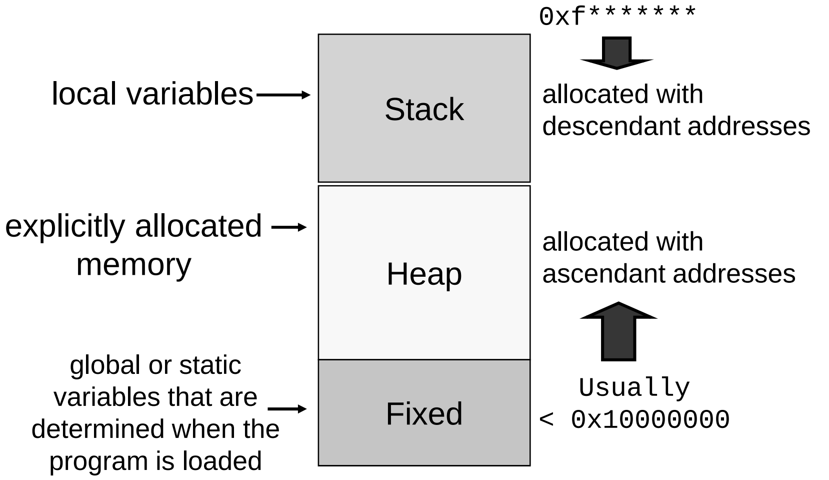
Return by Object, Pointer, and Reference

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?

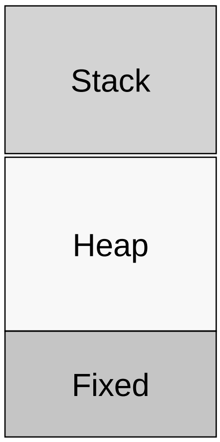
```
template<class T> class Array
{
  public:
     Array(size_t i = 0) { _data = new T[i]; }
     T& operator[] (size_t i) { return _data[i]; }
     const T& operator[] (size_t i) const {
        return _data[i]; }
     Array<T>& operator= (const Array& arr) {
     ... return (*this); }
  private:
 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

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
     ➔ Otherwise, memory leaks
3. Global variable (Fixed mem)
   - Visible by all files
   - Use “extern” to refer to global variable that is defined in other file

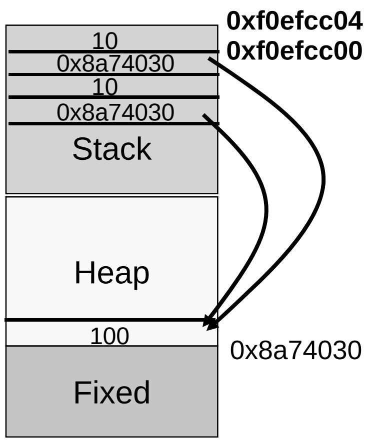


Address vs. Content

- 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

- 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

A Simple Example



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

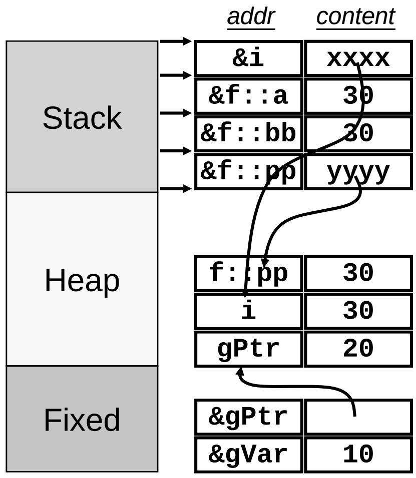
- Symbol table

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?

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);
}
```


Key Concept #9: 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)

Key Concept #10: 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

Key Concept #11: Memory Alignment

- What are the addresses of these variables?

```
int *p = new int(10);   // let addr(p) = 0x7fffe84ff0e0
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 #12: 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]

Key Concept #13: 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 **r = 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 #14: More on Array Variables

- 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; // OK, but memory leak?
    p = new int(20); // also OK
- An array variable (the const pointer) must be initialized
  - Recall: “const” variable must be initialized
  - int a[10]; // OK
    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

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: 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; explained later
  p = a; // OK, but memory leak...
  *(p++) = 40; // OK, but potential memory leak
  int *q = a;
  q[2] = 20;
  *(q+3) = 30;
  *(q++) = 40;  // OK
  delete a; // compile error/warning; runtime crash...
  delete p; // OK, but memory leak; explained later
  delete []q; // compile OK, but may get fishy result
  ```
- **What about:**
  `int a = 10; int *p = &a; ... delete p;`

Summary #3: 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]; }
  ```

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?
  }
```


Not everything can be const…

- What's the problem?
  1. void f(…) const { … }
  2. int & const a = …;
  3. class A
     {
        const int _data = 10;
     };

Key Concept #17: “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)

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

“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?)
```


“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??)

static_cast<T>(a)... Cast away static?? ☹

- Convert object "a" to the type "T"
  - No consistency check (i.e. sizeof(T))
    - ➔ 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();
};
```

Key Concept #18: “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);
}
```

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?
};
```


Let's do a review...

- Classified by "scope of visibility"
  - Global: seen by all files/functions
    Local: seen in the scope/function it is defined
- Attributes to a variable
  - const: "read-only"
  - static: memory of that variable remains valid
  - extern: something is declared outside this scope

What if two variables or functions with the same name need to be seen in the same scope?

Key Concept #19: 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 {...}.

What's next?

- Understanding "variables"
- Understanding "classes"
- Understanding "overloading"
- Understanding "polymorphism"
- Understanding "libraries"
- Exception handling