Topic 3

C++ Review Part I: Understanding "Variables"

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

09.23.2015

A Proclaimer...

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

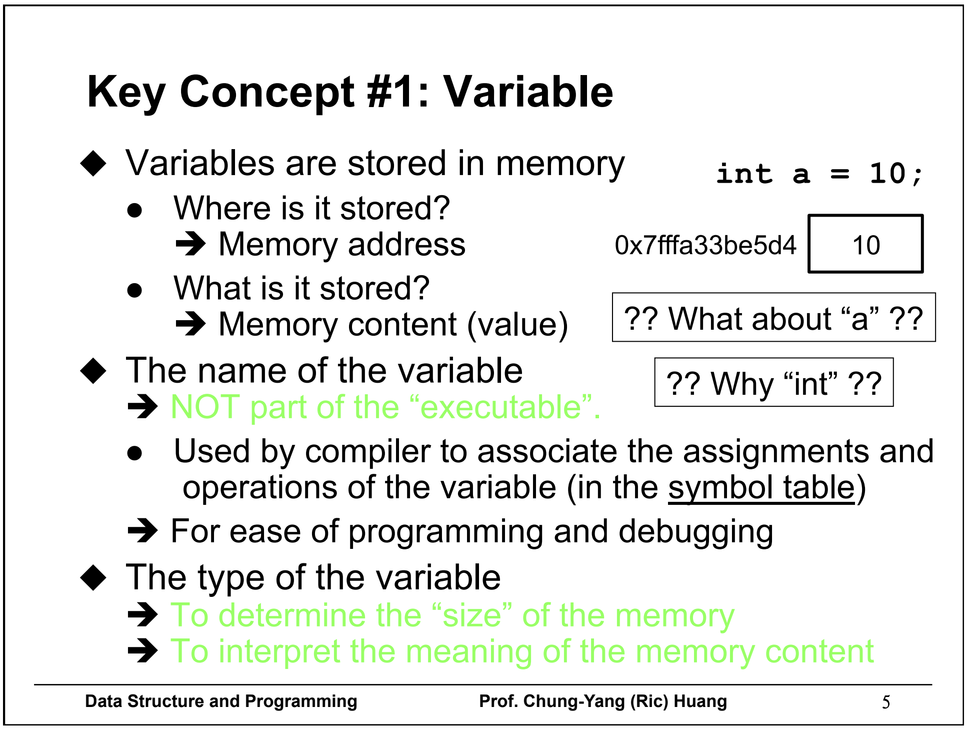
A Proclaimer...

- This lecture note contains a lot of details...
  - Not to memorize the details, but to understand why the language is designed that way.

- You need to have a good sense for programming, and at the same time be precise on the details.

Part I: Understanding “Variables”

- What is a variable?
- The concept of “memory”
- Object, pointer, reference
- Const
- Parameters and return value of a function



Key Concept #1: Variable

- 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 "executable".
  - Used by compiler to associate the assignments and operations of the variable (in the symbol table)

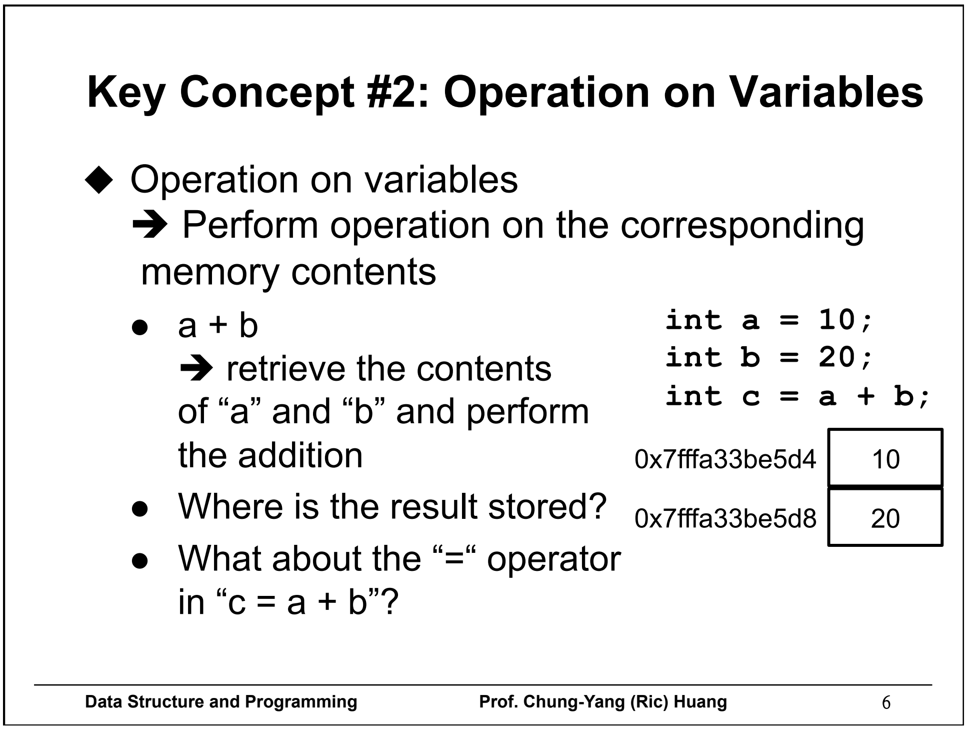
  ➔ For ease of programming and debugging
- The type of the variable
  ➔ To determine the "size" of the memory
  ➔ To interpret the meaning of the memory content

```
int a = 10;
```

0x7fffa33be5d4	10

?? What about "a" ??

?? Why "int" ??



Key Concept #2: Operation on Variables

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

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

0x7fffa33be5d4	10
0x7fffa33be5d8	20

Key Concept #3: ‘=‘ operator

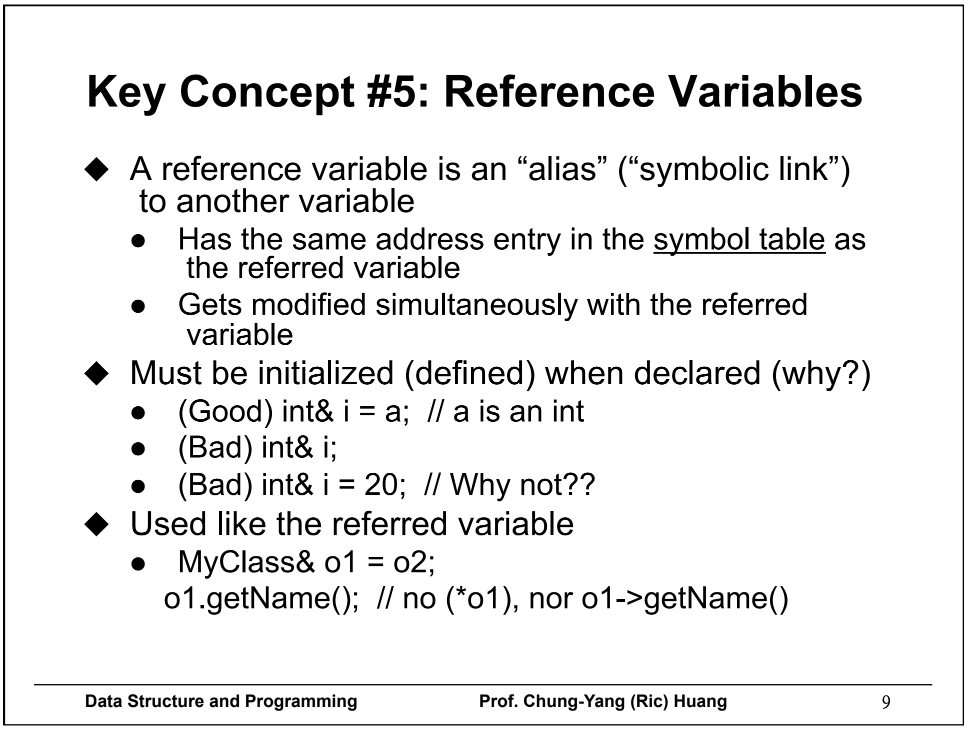
- ‘=‘ operator in C/C++ performs “assignment”, not “equal to” (so “a = a + 1” makes sense)
  - “Assignment” means “copy the value of the right hand side expression to the location of the left hand side variable”
    - `c = a + b;`

    ➔ Where is the result of “a+b” stored?
  - What about:

    -
      ```
      int *p = q;
      int *r = new int(10);
      ```

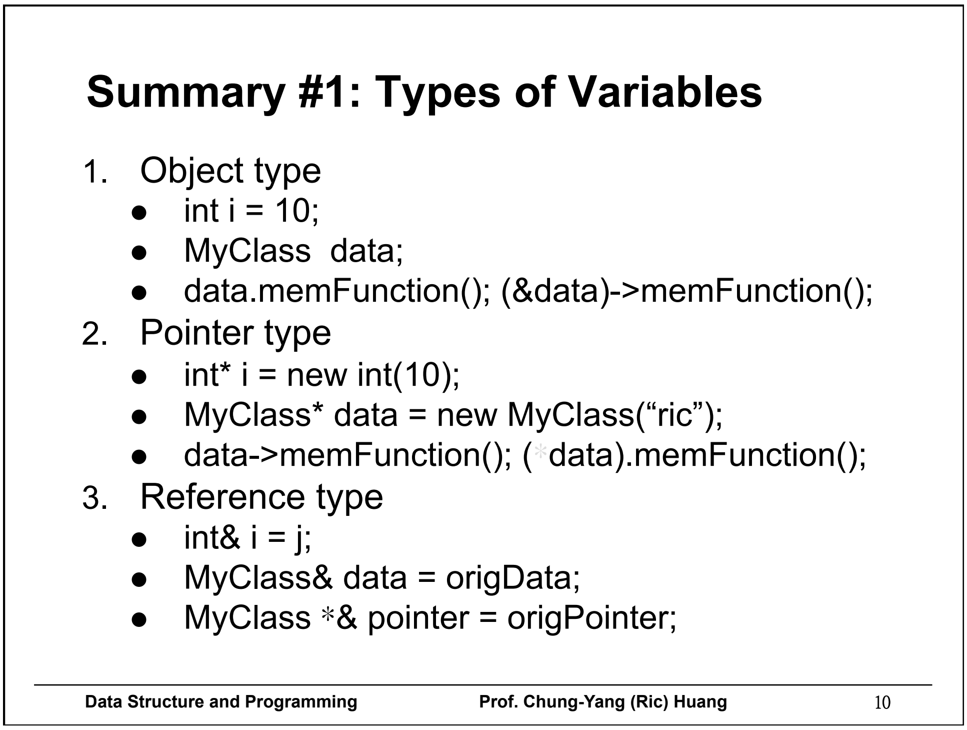
Key Concept #4: Pointer Variables

- Pointers are also variables
  - int a;
    The memory location of “a” stores an integer value.
  - int *p;
    The memory location of “p” stores a memory address, which points to an integer memory location.
- “a” vs. “p”
  - Both are variables
  - Different types: “int” vs. “int *”



Key Concept #5: 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

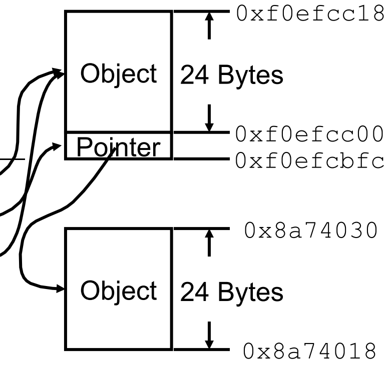
1. Object type
   - int i = 10;
   - MyClass data;
   - data.memFunction(); (&data)->memFunction();
2. Pointer type
   - int* i = new int(10);
   - MyClass* data = new MyClass(“ric”);
   - data->memFunction(); (*data).memFunction();
3. Reference type
   - int& i = j;
   - MyClass& data = origData;
 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

Object, Pointer, Reference?

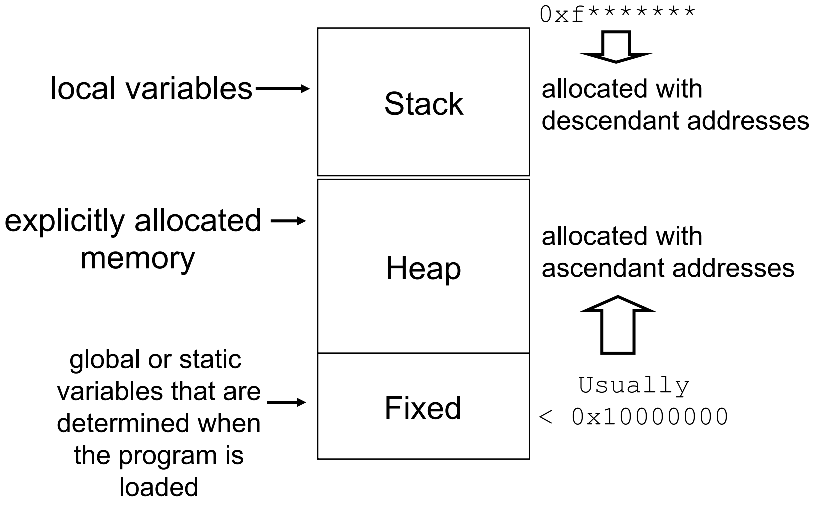
-
```
void goo(){
  MyClass   aaa;  // Object(Let size = 24Byte)
  MyClass*  ppp;  // Pointer
  MyClass&  rrr = aaa;  // Reference
   ...
}
```

- Symbol table

name	address
aaa	0xf0efcc00
ppp	0xf0efcbfc
rrr	0xf0efcc00

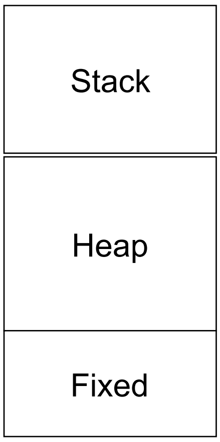


Key Concept #6: Types of Memory Allocations



Scope and Visibility

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



Key Concept #7:
Every variable that is NOT global, is local.

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

Address vs. Content

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

Key Concept #8:
int *p1 = &i;   vs.   int *p2 = new int;

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

Key Concept #9: “new” and “delete” operators

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

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”

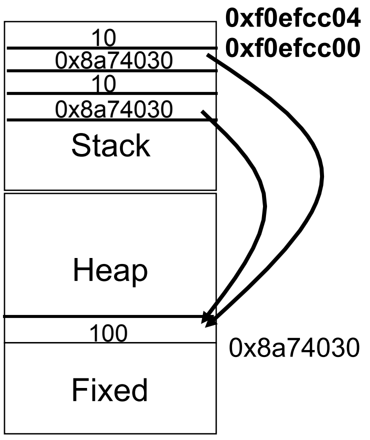
Copy the content, but, what is the content?

-
  ```
  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?
  ```

Summary: A Simple Example

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



0xf0efcc04

0xf0efcc00

10

0x8a74030

10

0x8a74030

Stack

Heap

100

0x8a74030

Fixed

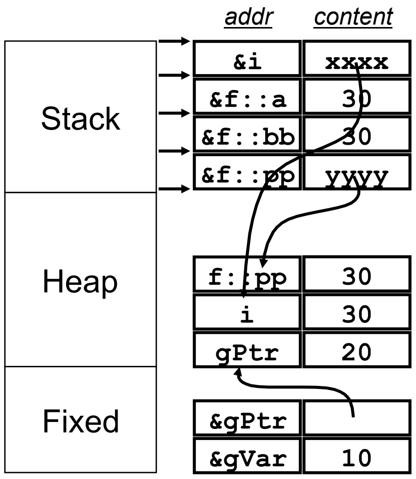
- Symbol table

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



addr	content
&i	xxxx
&f::a	30
&f::bb	30
&f::pp	yyyy
f::pp	30
i	30
gPtr	20
&gPtr	
&gVar	10

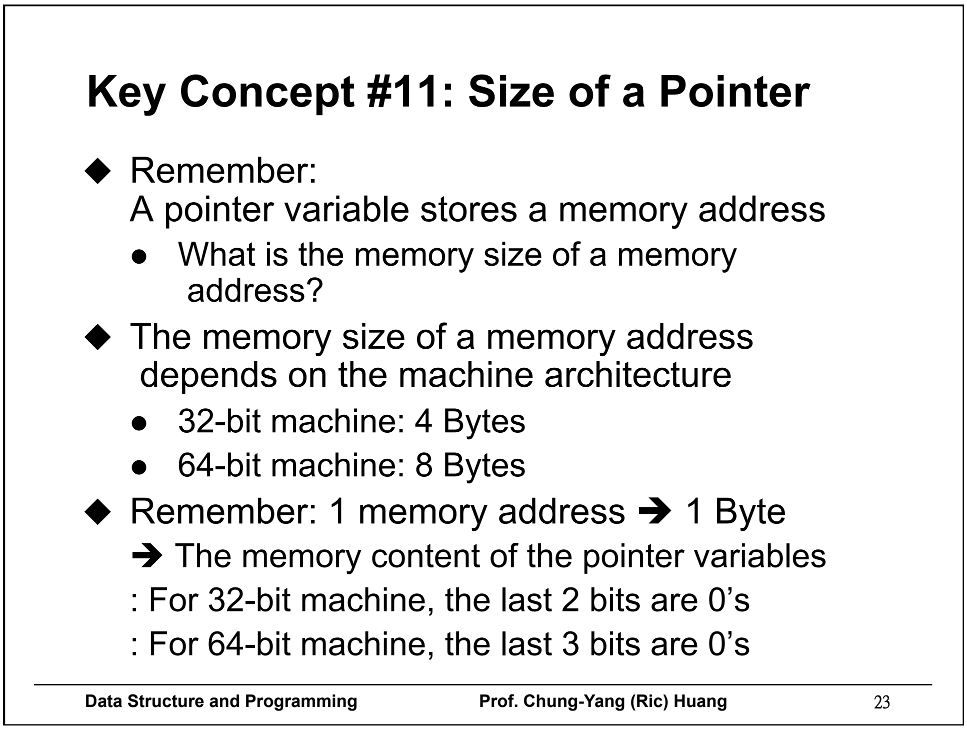
```
int  gVar = 10;
int* gPtr = new
    int(20);

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);
}
```

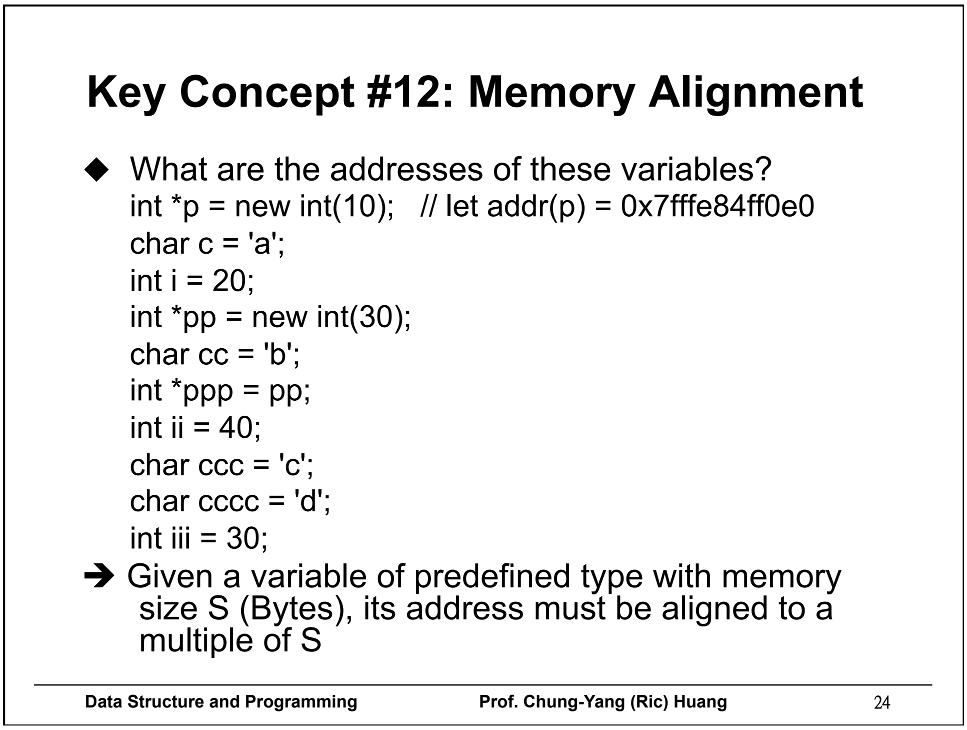
Key Concept #10: Memory Sizes

- 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 #11: 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 #12: Memory Alignment

- ◆ What are the addresses of these variables?

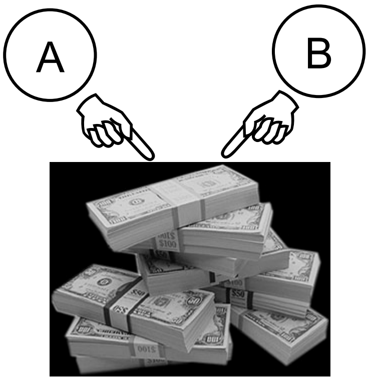
```
int *p = new int(10);   // let addr(p) = 0x7fffe84ff0e0
char c = 'a';
int i = 20;
int *pp = new int(30);
char cc = 'b';
int *ppp = pp;
int ii = 40;
char ccc = 'c';
char cccc = 'd';
int iii = 30;
```

➔ Given a variable of predefined type with memory size S (Bytes), its address must be aligned to a multiple of S

Can you answer this...

- Why do we need “pointer” in C/C++?

“Share” !!



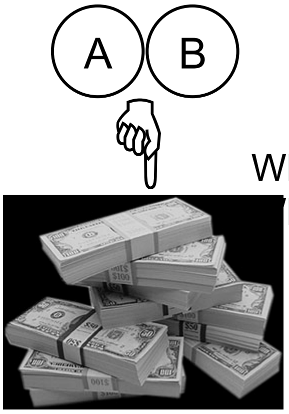
```
compared:
int a = 10;
int b = a;
b += 10;
```

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

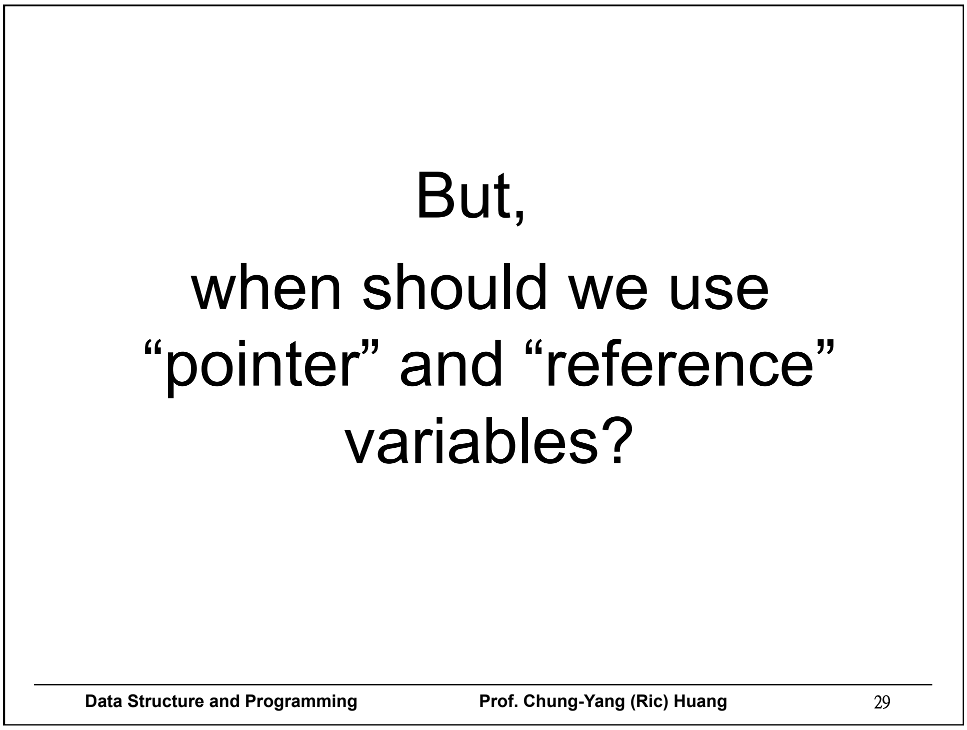
Can you answer this...

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

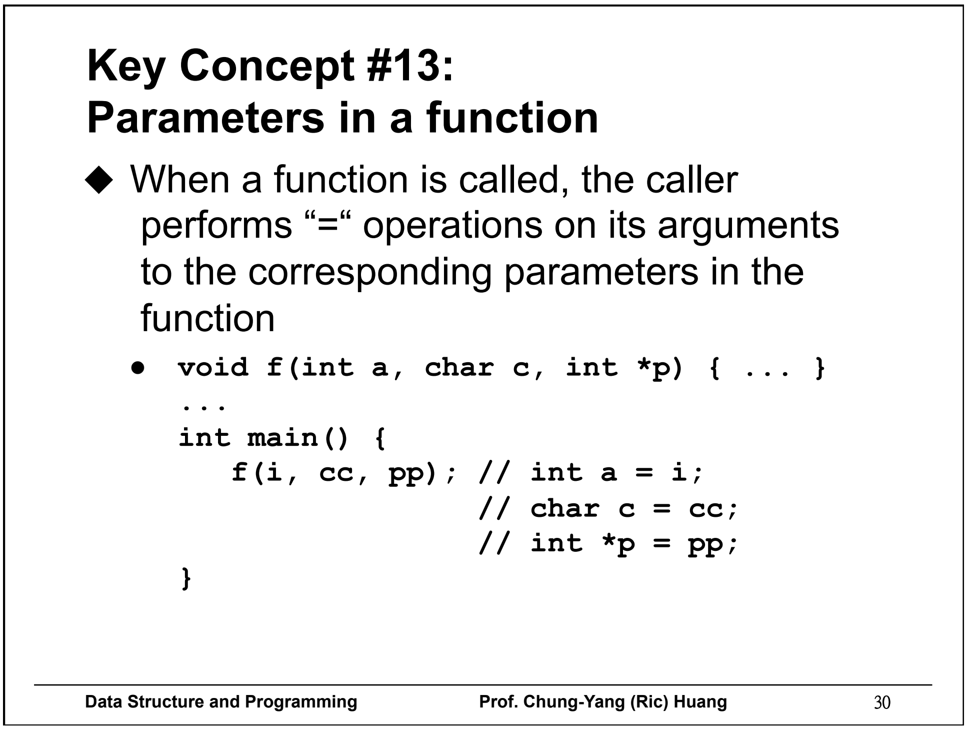
"Share" vs. "Clone"!!



Why should we share?
Why should we clone?



But, when should we use "pointer" and "reference" variables?



Key Concept #13: 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); }
```

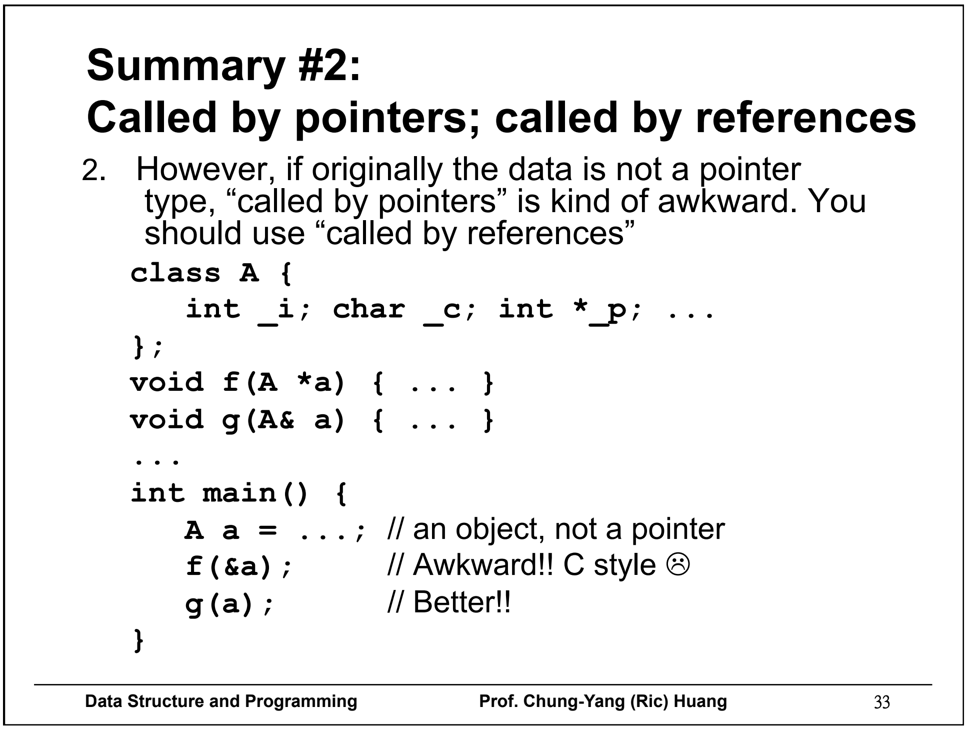
```
main()
{
   int a = 10;
   int* p = &a;
   int a1,a2,a3,a4,a5;
   f1(a); a1 = a;
   f2(a); a2 = a;
   f3(p); a3 = *p;
   f4(p); a4 = *p;
   f5(p); a5 = *p;
}
```

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”

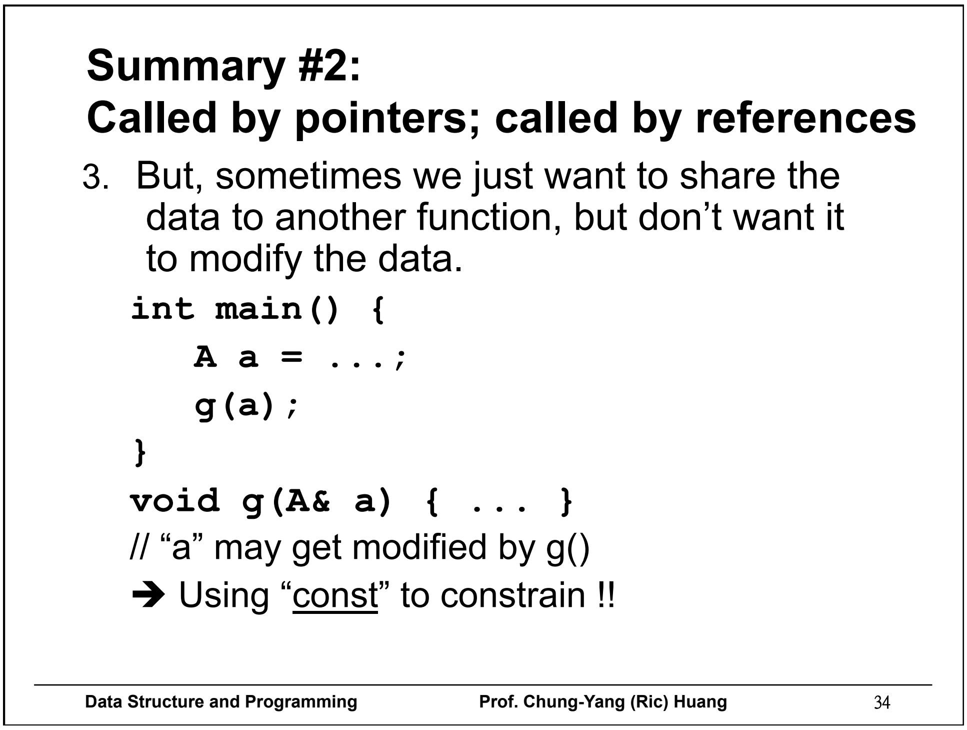
```
class A {
   int _i; char _c; int *_p; ...
};
void f(A *a) { ... }
...
int main() {
   A *a = ...;
   f(a);
}
```



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!!
}
```



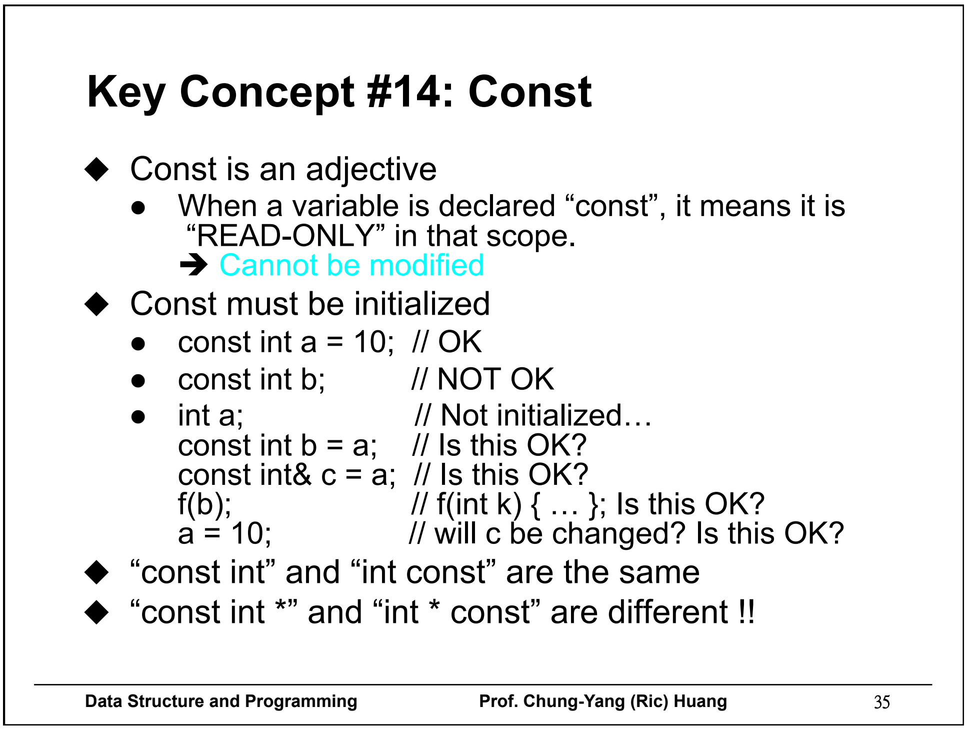
Summary #2: Called by pointers; called by references

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 = ...;
   g(a);
}
void g(A& a) { ... }
```

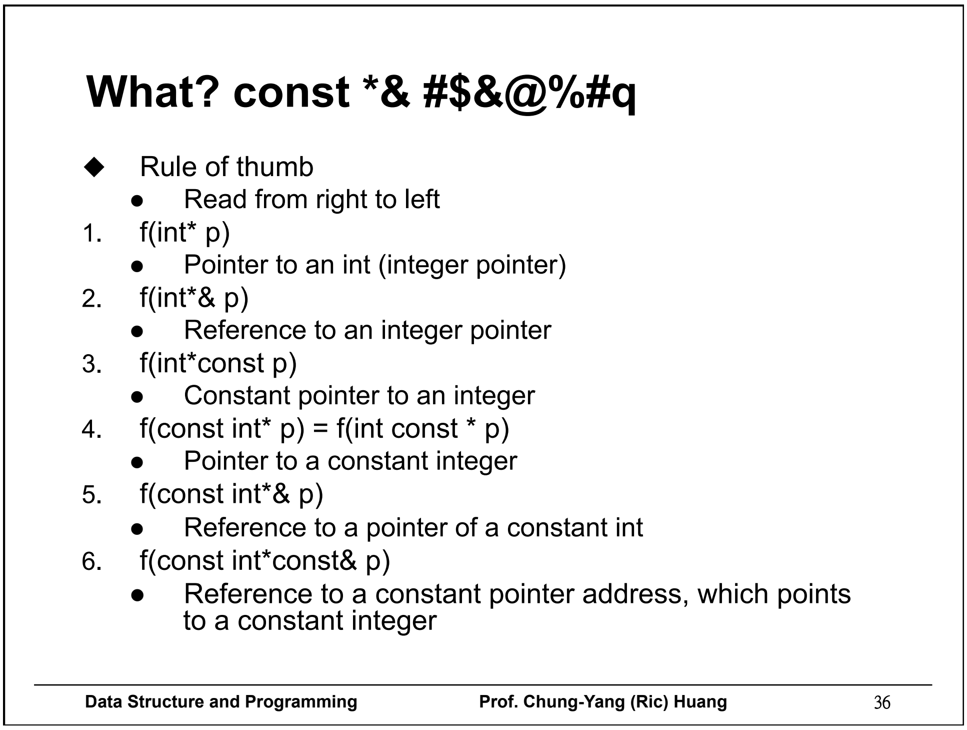
// “a” may get modified by g()

➔ Using “const” to constrain !!



Key Concept #14: Const

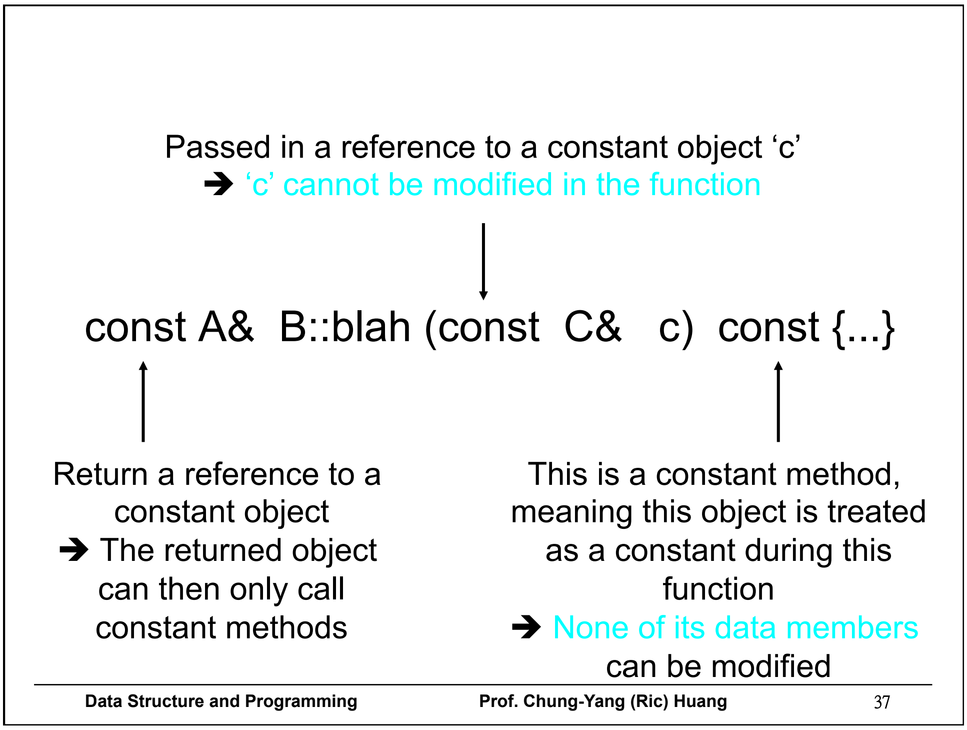
- Const is an adjective
  - When a variable is declared “const”, it means it is “READ-ONLY” in that scope.
    ➔ Cannot be modified
- Const must be initialized
  - const int a = 10; // OK
  - const int b;   // NOT OK
  - int a;   // Not initialized…
    const int b = a;   // Is this OK?
    const int& c = a; // Is this OK?
    f(b);   // f(int k) { … }; Is this OK?
    a = 10;   // will c 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

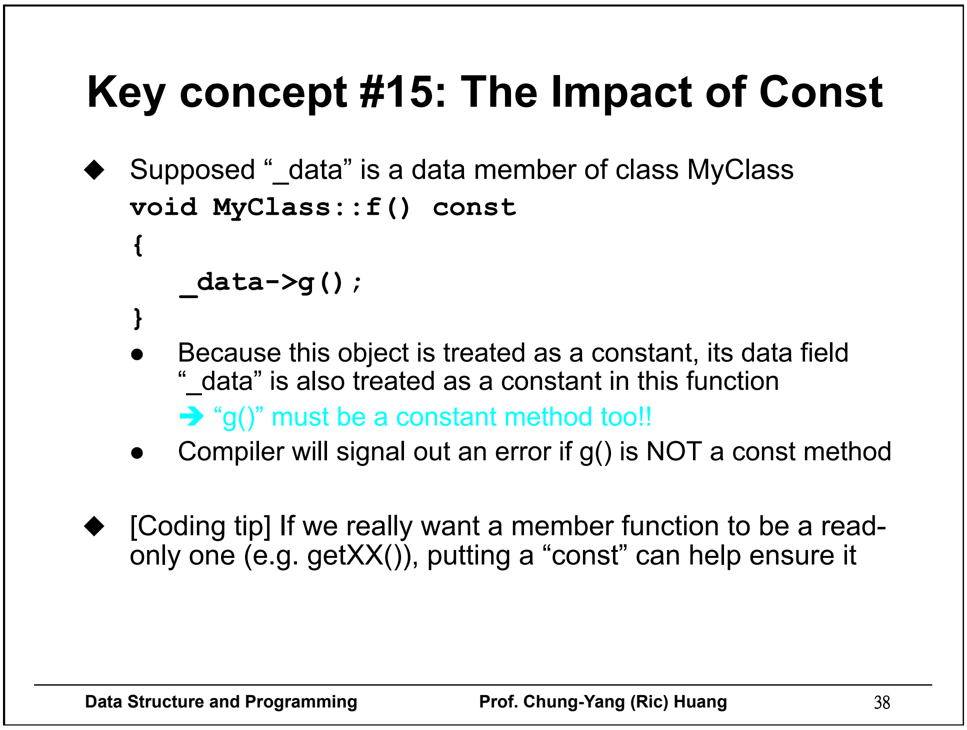


Passed in a reference to a constant object ‘c’
➔ ‘c’ cannot be modified in the function

```
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



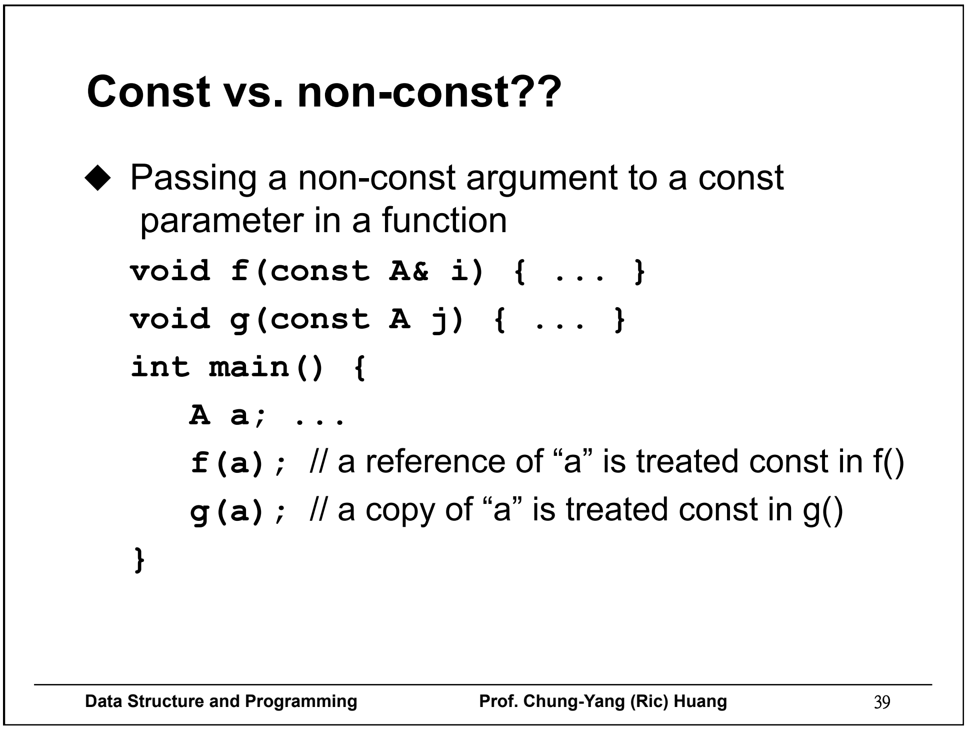
Key concept #15: The Impact of Const

- 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

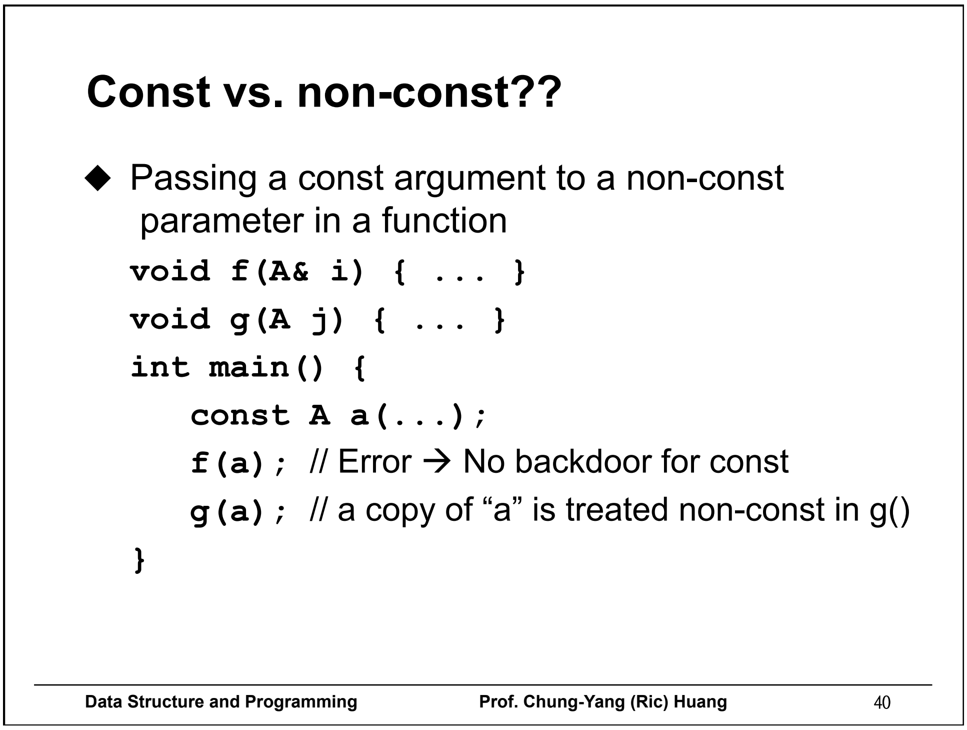
- [Coding tip] If we really want a member function to be a read-only one (e.g. getXX()), putting a “const” can help ensure it



Const vs. non-const??

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

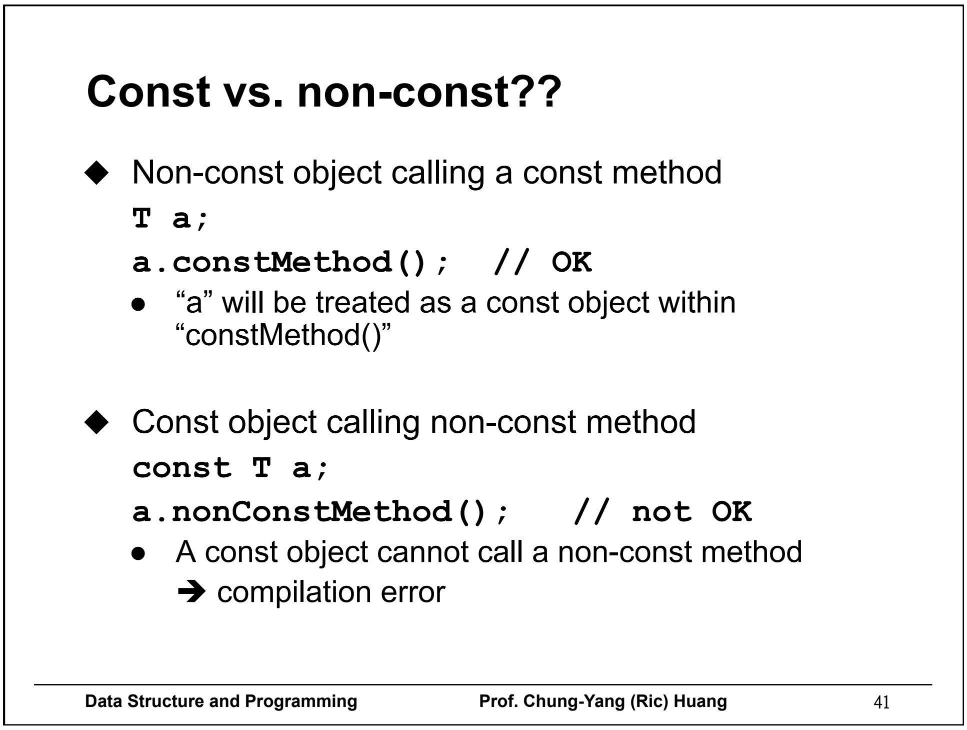
```
void f(const A& i) { ... }
void g(const A j) { ... }
int main() {
   A a; ...
   f(a); // a reference of “a” is treated const in f()
   g(a); // a copy of “a” is treated const in g()
}
```



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

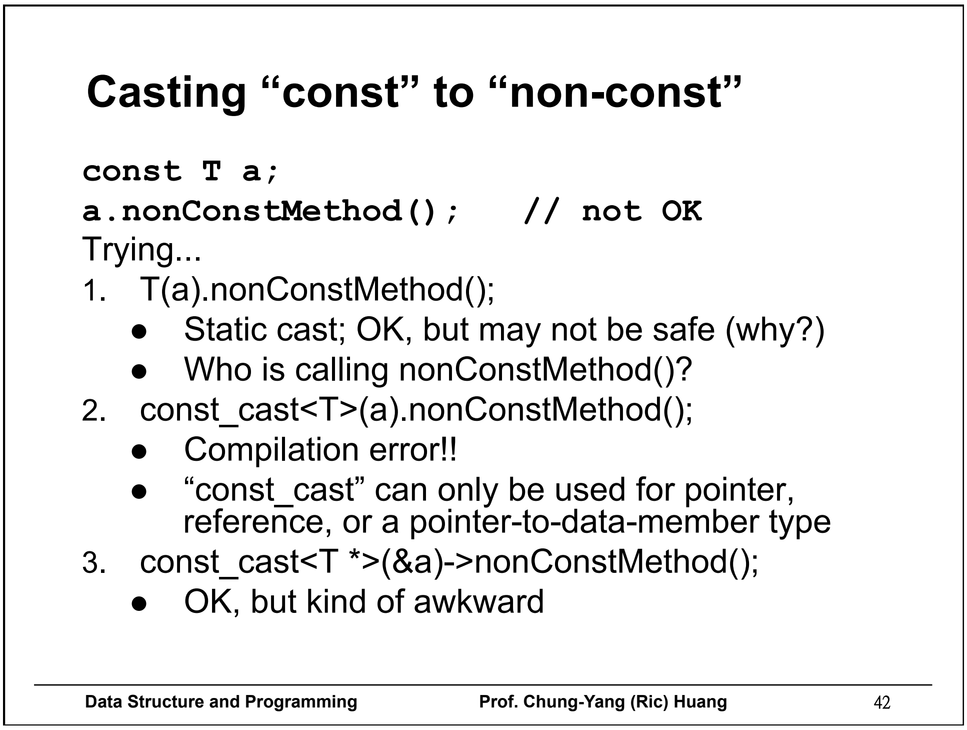
```
T a;
a.constMethod();  // OK
```

  - "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

Key Concept #16:
“mutable” --- a back door for const method

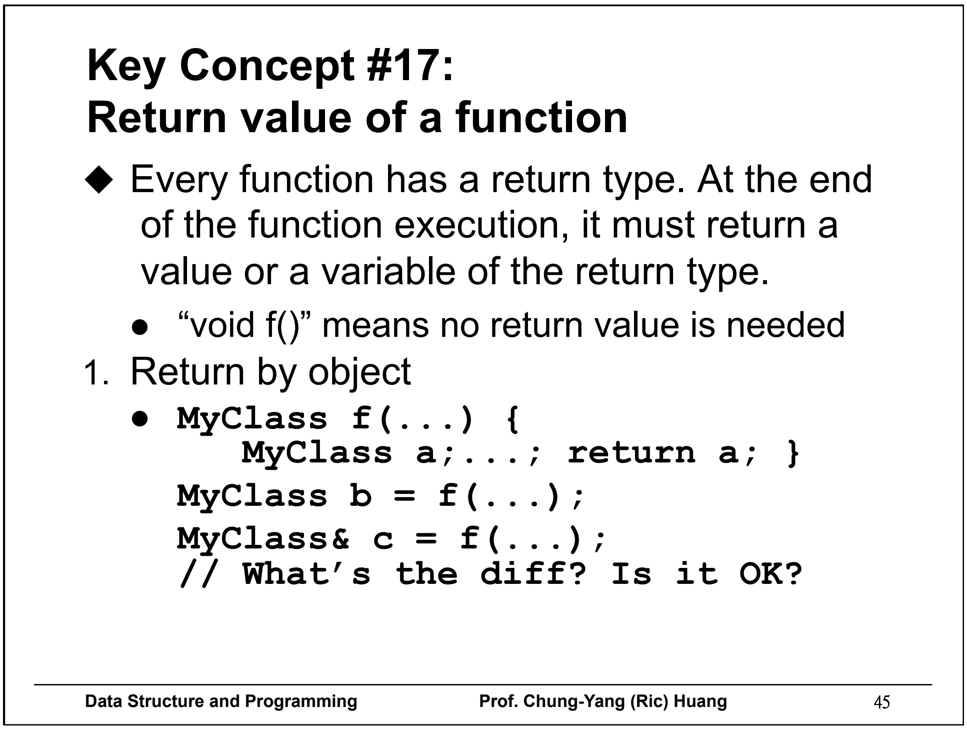
◆ However, sometimes we MUST modify the data member in a const method

- void MyClass::f() const

```
{
   _flags |= 0x1;  // setting a bit of the _flags
}
```

- In such case, declare “_flag” with “mutable” keyword
  - e.g.

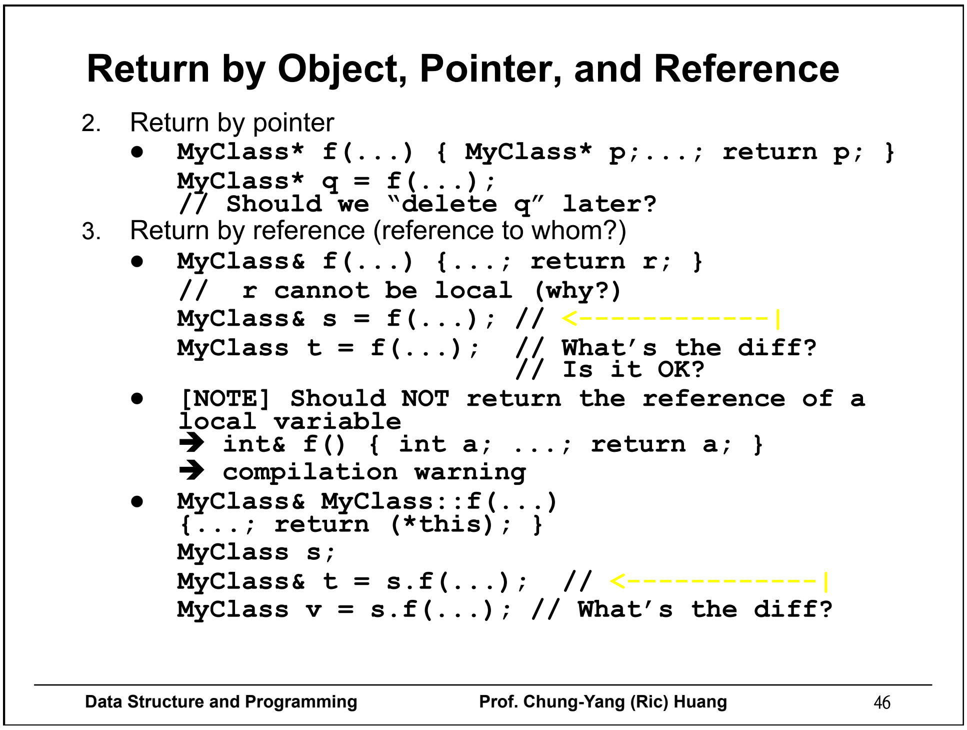
```
mutable unsigned  _flag;
```



Key Concept #17: 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?
```



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;"?
  ```

A quick review:
Understanding “Variables”

- Variable and memory
- Content vs. address
- What is a pointer variable?

  What is a reference variable?
- Why do we need “const”?
- Parameters and return value of a function