Topic 4

Memory Management

資料結構與程式設計

Data Structure and Programming

10/19/2016

Outline

- ◆ Memory related problems
  - ● Illegal memory address access
  - ● Memory leaks
  - ● Fragmentation
  - ● Performance issues
- ◆ Memory management
  - ● Basic concept
  - ● Categorization
  - ● How to implement

Memory Related Problems

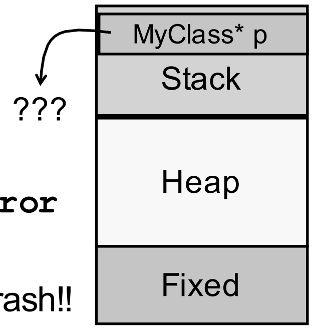
1. Illegal memory address access
2. Memory leaks
3. Fragmentation
4. Performance issues

Illegal Memory Address Access

1. Uninitialized memory read/write
   - Access to the content of a pointer variable that is not yet allocated

```
void f() {
  MyClass* p;
  ...
  int i
  = p->getData(); // error
}
```

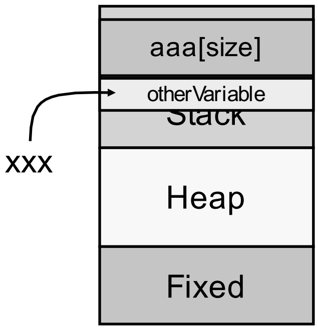
➔ Compilation OK; Runtime crash!!



Illegal Memory Address Access

2. Array bound read/write
   - Array index is greater than the bound

```
void f() {
  size_t size;
  ...
  size = ...;
  int aaa[size];
  ...
  size_t idx = ...;
  ...
  // error if idx >= size
  aaa[idx] = ...;
}
```

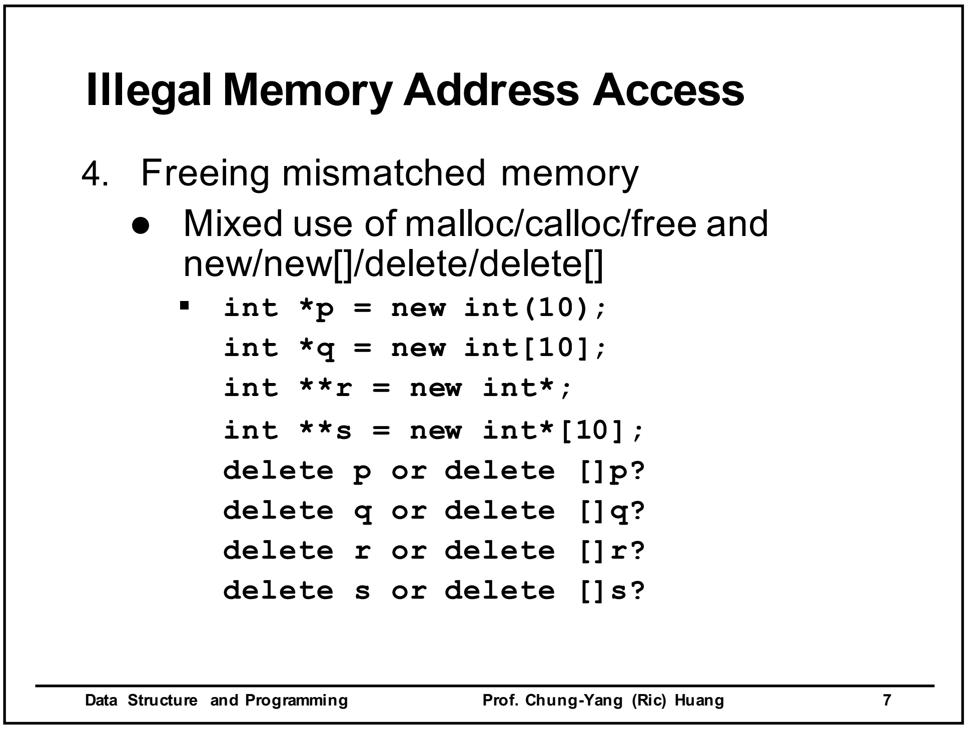


➔ Compilation OK,
but may get strange runtime bug

Illegal Memory Address Access

3. Freed memory read/write
   - Access to the just freed memory allocation
   - May still get the expected content, but will become garbage when reallocated by others

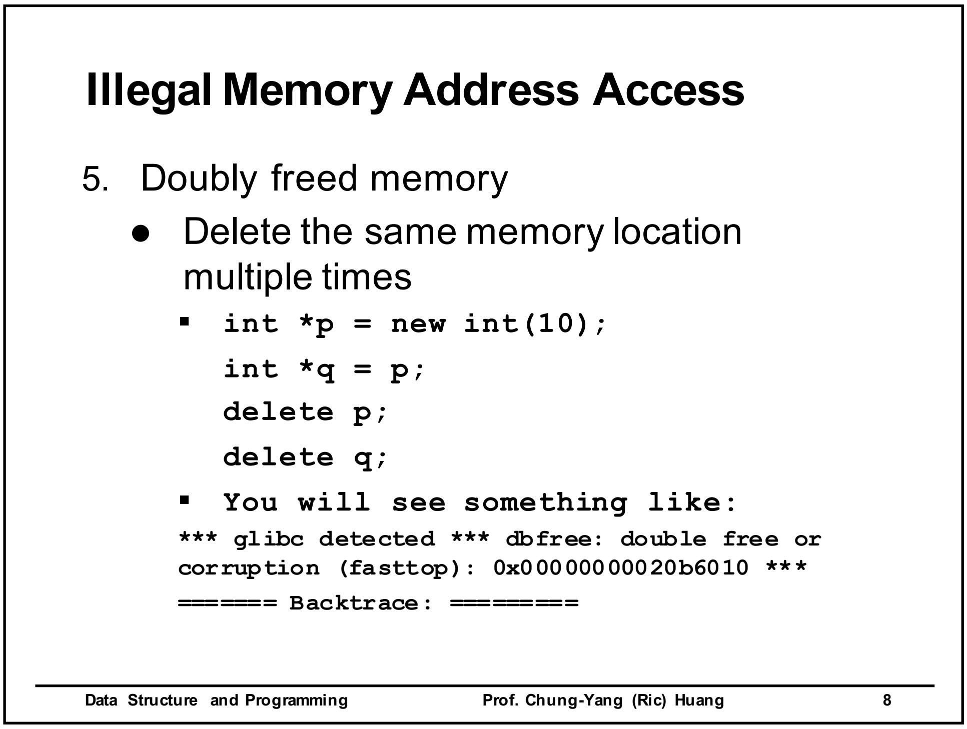
```
void f() {
   int* p = new int;
   cout << p << endl;
   delete p;
   // may print out the same address
   cout << p << endl;
   *p = 30;  // [NOTE] compilation & runtime OK;
   int j = *p;
   cout << j << endl;
   int* q = new int(20);
   int k = *p;
   cout << k << endl; // What's the value for k?
}
```



Illegal Memory Address Access

4. Freeing mismatched memory
   - Mixed use of malloc/calloc/free and new/new[]/delete/delete[]

     -
       ```
       int *p = new int(10);
       int *q = new int[10];
       int **r = new int*;
       int **s = new int*[10];
       delete p or delete []p?
       delete q or delete []q?
       delete r or delete []r?
       delete s or delete []s?
       ```



Illegal Memory Address Access

5. Doubly freed memory
   - Delete the same memory location multiple times

     -
       ```
       int *p = new int(10);
       int *q = p;
       delete p;
       delete q;
       ```
     - **You will see something like:**

```
*** glibc detected *** dbfree: double free or
corruption (fasttop): 0x000000000020b6010 ***
====== Backtrace: ========
```

How to avoid illegal memory access?

1. Allocate and free memory of data members in constructor and destructor
   - Use object to wrap the pointer variables

```
 class MyClass {
    A* _pp;
  public:
    MyClass(int i = 0) { _pp = new A(i); }
    ~MyClass() { delete _pp; }
  };
  void f() {
    MyClass o;  // o._pp is allocated
  }// o._pp is deleted automatically
```

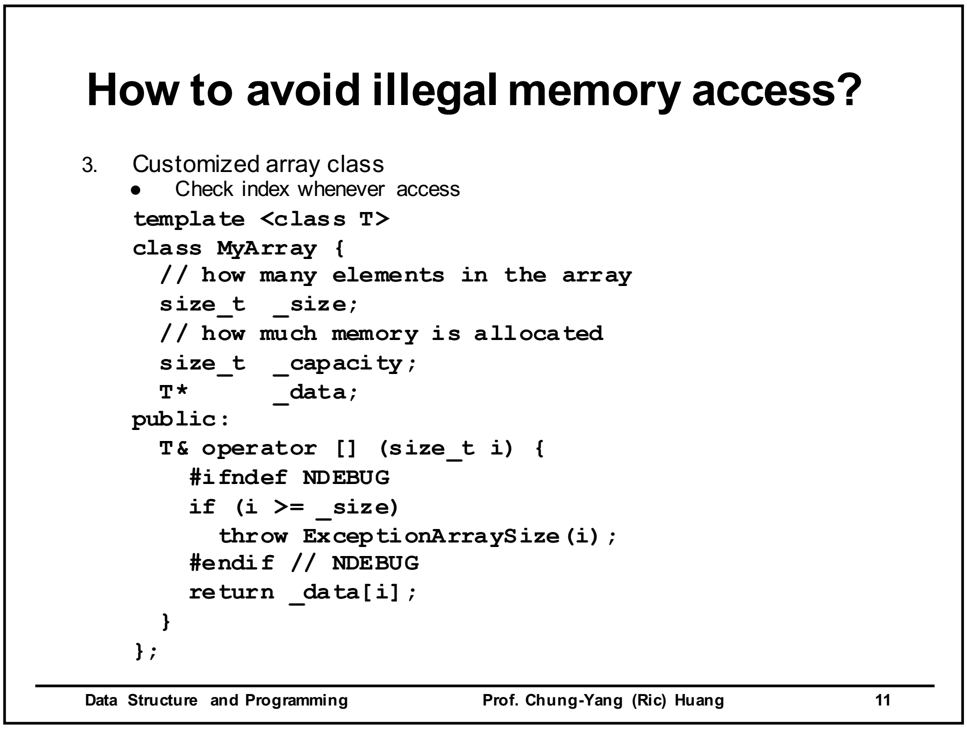
   - All the operations on _pp should go through class MyClass
     - Can make class A a private class to MyClass (by “friend”)
   - What about copy constructor or assignment operator?
     - May need “reference count” to avoid double-free error

How to avoid illegal memory access?

2. Paired memory allocation/deletion functions
   - Don’t allow too many functions to allocate and delete pointers

// [No good] hard to keep track of the memory allocation of _pp

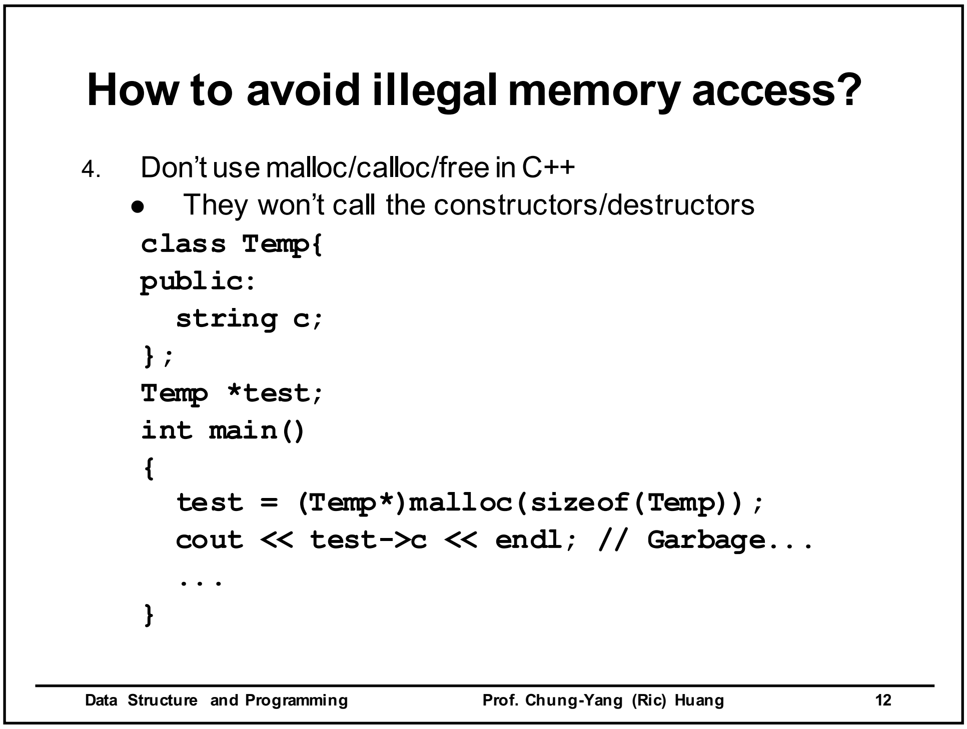
```
class MyClass {
  int* _pp;
public:
  void f1(int i) {
    ...; _pp = new int(i); ... }
  void f2() {
    ...; delete _pp; ... }
  void f3() {
    ...; _pp = new int(j);
    ...; delete _pp; ... }
};
```



How to avoid illegal memory access?

3. Customized array class
   - Check index whenever access

```
template <class T>
class MyArray {
  // how many elements in the array
  size_t  _size;
  // how much memory is allocated
  size_t  _capacity;
  T*      _data;
public:
  T& operator [] (size_t i) {
    #ifndef NDEBUG
    if (i >= _size)
      throw ExceptionArraySize(i);
    #endif // NDEBUG
    return _data[i];
  }
};
```



How to avoid illegal memory access?

4. Don’t use malloc/calloc/free in C++
   - They won’t call the constructors/destructors

```
class Temp{
public:
  string c;
};
Temp *test;
int main()
{
  test = (Temp*)malloc(sizeof(Temp));
  cout << test->c << endl; // Garbage...
  ...
}
```

How to avoid illegal memory access?

5. Correctly use of new/new[] and delete/delete[]
6. Memory management

In short, create your own style and strictly abide by your disciplines

What about the overhead generated by the above preventions?

- Minor overhead is OK; better than debugging tricky memory bugs
- Use “#ifndef NDEBUG” to bypass them in optimized mode compilation
  - “Debug build” --- for developer
    - g++ -g xxx.cpp

    “Optimized build” --- for tool release
    - g++ -O3 -DNDEBUG xxx.cpp

```
============
#ifndef NDEBUG
<codes for debug mode only>
#endif // NDEBUG
============
```

Memory Related Problems

1. Illegal memory address access
2. Memory leaks
3. Fragmentation
4. Performance issues

What is memory leak?

◆ Not freeing allocated memory, so as the program runs, the total occupied memory is increasing and cannot be reclaimed

- ➔ Performance degradation due to thrashing
- ➔ Program terminated due to memory out

Why do I have memory leaks?

1. Pointer data members not freed
   - class A { B *_b; …; A() { _b = new… } …};
     A a1, a2; … a2 = a1; …
2. Local pointers not freed
   - A *a; a = new …;
     if (xxx) { … return; }
     delete a;
3. Freeing memory mismatch
   - e.g. p = new MyClass[10]; …; delete p;
4. Overwrite on allocated pointer variables

```
void f() {
  int* p = new int;
  ...
  p = g();
  // original p cannot be deleted
  delete p;
}
```

How do I know if I have memory leak?

- Well, as the program runs longer, the memory usage is increasing and doesn't seem to saturate.

- To diagnose
  1. Code review
  2. Using tools
     - Commercial: purify
     - GNU: valgrind (http://valgrind.org/)
  3. "top" command

How to avoid memory leaks?

◆ Good practice makes it all !!

◆ Memory management
  - Block allocation and deletion

◆ Use “reference count” to keep track whether it is safe to delete an pointer
  - How??
  - A* p = q;
    // Who’s ref count is incremented by 1?
  - Constructor? Destructor? Object wrapper?

Memory Related Problems

1. Illegal memory address access
2. Memory leaks
3. Fragmentation
4. Performance issues

What is fragmentation?

◆ Like the fragmentation in your hard disk, the memory used in your program may have fragmentation too

DOGBERT'S TECH SUPPORT
YOU HAVE A BAD CASE OF COMPUTER ROT.
www.dilbert.com scottadams@aol.com
YOUR COMPUTER IS DESIGNED TO BECOME SLOWER AND MORE UNRELIABLE OVER TIME SO YOU HAVE TO UPGRADE.
5-28-05 © 2005 Scott Adams, Inc./Dist. by UFS, Inc.
BUT IF YOU'D LIKE SOME FALSE HOPE, I CAN TELL YOU HOW TO DEFRAGMENT YOUR DISK DRIVE.

© Scott Adams, Inc./Dist. by UFS, Inc.

What is fragmentation?

◆ Like the fragmentation in your hard disk, the memory used in your program may have fragmentation too

●
```
MyClass12Byte* a = new MyClass12Byte ;
MyClass16Byte* b = new MyClass16Byte;
MyClassWhatever* c = new MyClassWhatever;
delete a;
delete b;
MyClass16Byte* d = new MyClass16Byte;
MyClass16Byte* e = new MyClass16Byte;
```

➔ Memory fragmentation of 12 Bytes (where??)

How to avoid memory fragmentation?

- Memory fragmentation will make your program use more memory than necessary

- How to fix it?
  - Not easy, unless you use your own memory management and carefully allocate memory pieces with different sizes

Performance Issues

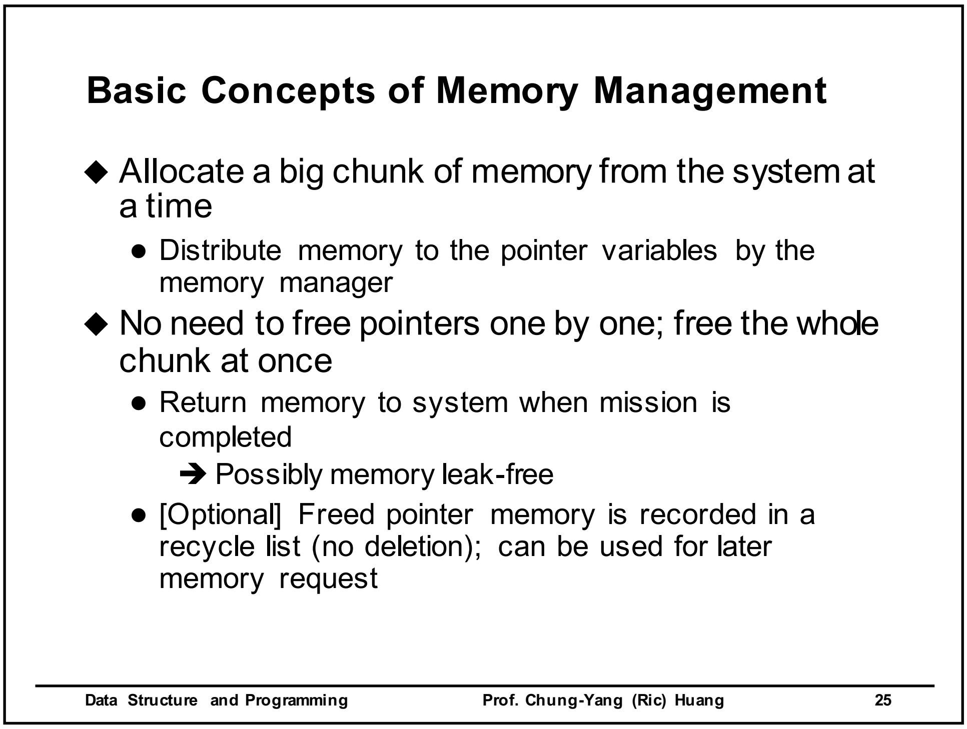
- Overhead in system calls of memory allocation / deletion

- What’s the runtime difference?

1.
```
A* a[1 << 20];
for (int i = 0; i < (1 << 20); i++) {
   a[i] = new A;
   *(a[i]) = i;
}
```

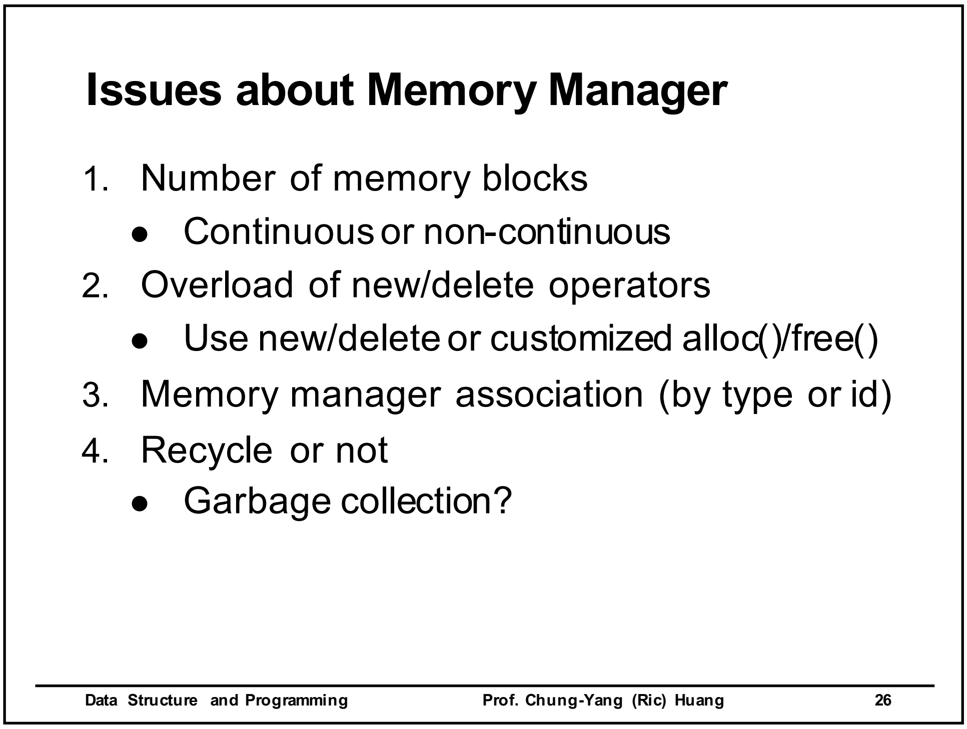
2.
```
A* a[1 << 20];
A* b = (A *)calloc(1 << 20, sizeof(A));
for (int i = 0; i < (1 << 20); i++) {
   a[i] = b + i;
   *(a[i]) = i;
}
```

➔ But, will A’s constructors be called in the second case?



Basic Concepts of Memory Management

- ◆ Allocate a big chunk of memory from the system at a time
  - ● Distribute memory to the pointer variables by the memory manager
- ◆ No need to free pointers one by one; free the whole chunk at once
  - ● Return memory to system when mission is completed
    - ➔ Possibly memory leak-free
  - ● [Optional] Freed pointer memory is recorded in a recycle list (no deletion); can be used for later memory request



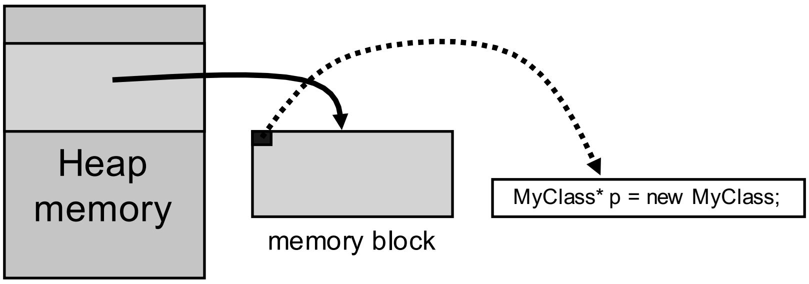
Issues about Memory Manager

1. Number of memory blocks
   - ● Continuous or non-continuous
2. Overload of new/delete operators
   - ● Use new/delete or customized alloc()/free()
3. Memory manager association (by type or id)
4. Recycle or not
   - ● Garbage collection?

Memory Blocks in Memory Manager

- How many memory should we claim from the system each time (i.e. 1 memory block)?
  - Too small: many system calls
  - Too big: waste of memory if not used up

  ➔ Depend on applications, usually 4K – 1MB



1. Continuous Memory Block

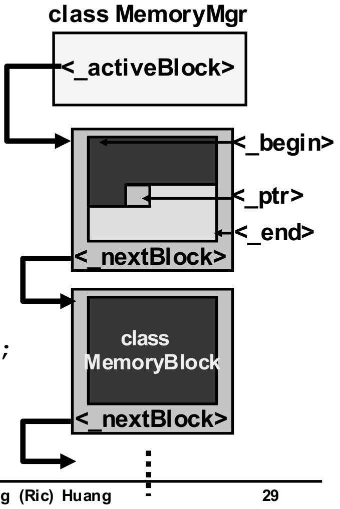
- Only 1 memory block
  - When _size >= _capacity, reallocate a bigger block and copy the original data over
  - Difficult to work with pointer variables (why?)
  - Addresses are continuous; can access by index

```
class MemoryBlock {
   #define S_SIZE_T sizeof(size_t)
public:
   MemoryBlock(size_t B) {  // block size = B Bytes
      _begin = _next = (void*)malloc(B);
      _end = _begin + numElm(B);_}
   void* alloc (size_t t) { // t is number of Bytes
      void* tmp = getNext(t);
      if (tmp >= _end) { /* allocate new memory and copy to it */ }
      void* ret = _next; _next = tmp;
      return ret;
   }
private:
   void    *_begin, *_next, *_end;
   void*   getNext(size_t t) const {
           size_t nt = numElm(t);  return (size_t*)_next + nt; }
   size_t  numElm(size_t t) const {
           return (t + S_SIZE_T - 1) / S_SIZE_T; }
};
```

2. Non-continuous Memory Block

- When size >= capacity, allocate a new memory block and link it to the previous one
  - Keep an active memory block

```
class MemoryBlock {
   char*          _begin;
   char*          _ptr;
   char*          _end;
   MemoryBlock*   _nextBlock;
};
class MemoryMgr {
   MemoryBlock*    _activeBlock;
};
```



Overload of new/delete operators

- We can overload the new and delete operators of a class
  - void* operator new (size_t t);
  - void* operator new[] (size_t t);
  - void operator delete (void* p);
  - void operator delete[] (void* p);

    (Can also be static functions)

  [Note] The parameters ‘t’ and ‘p’ are passed in by compiler with the “new/delete” calls
- Advantage
  - Memory manager is transparent to the programmer; can turn on and off easily
- For more information, please see (for example)
  - http://www.relisoft.com/book/tech/9new.html

Example: newOp.cpp

```
class A
{
  int       _a;
  int       _b;
  int       _c;
  short  _d;
  // sizeof(A) = 14 → 16 Bytes

public:
  A() {}
  ~A() {}

  static void* operator new(size_t t) {
     cout << "new (inside A): " << endl;
     cout << ">> size = " << t << endl;
     A* p = (A*)malloc(t);
     cout << ">> ptr  = " << p << endl;
     return p;
  }
  static void* operator new[](size_t t) {
     cout << "new[] (inside A): " << endl;
     cout << ">> size = " << t << endl;
     A* p = (A*)malloc(t);
     cout << ">> ptr  = " << p << endl;
     return p;
  }
  static void operator delete(void* p) {
     cout << "delete (inside A): " << endl;
     cout << ">> ptr  = " << p << endl;
     free(p);
  }
  static void operator delete[](void* p) {
     cout << "delete[] (inside A):  " << endl;
     cout << ">> ptr  = " << p << endl;
     free(p);
  }
};
```

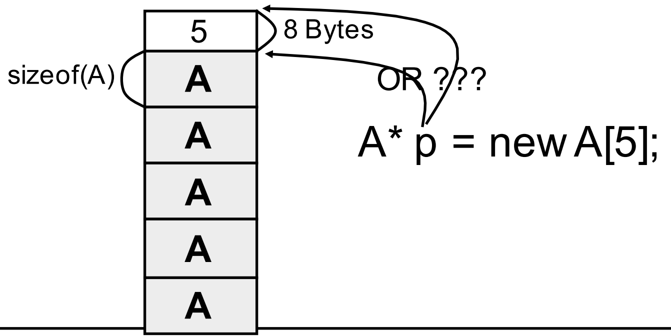
Example: newOp.cpp

```
int main()
{
  A* a = new A;
  cout << endl;
  cout << "new (in main): " << endl;
  cout << ">> ptr  = " << a << endl;
  cout << endl;

  A* b = new A[10];
  cout << endl;
  cout << "new[] (in main): " << endl;
  cout << ">> ptr  = " << b << endl;
  cout << endl;

  delete a;
  cout << endl;
  delete []b;
  cout << endl;
}
```

```
====== Sample output ======
new (inside A):
>> size = 16
>> ptr  = 0x502010

new (in main):
>> ptr  = 0x502010

new[] (inside A):
>> size = 168
>> ptr  = 0x502030

new[] (in main):
>> ptr  = 0x502038

delete (inside A):
>> ptr  = 0x502010

delete[] (inside A):
>> ptr  = 0x502030
```

What did “newOp.cpp” tell you?

1. Must have a destructor... somehow....
   - Try comment out the destructor in newOp.cpp...
2. Size of “new[]” = array size + sizeof(size_t)
   - How do we record the array size for delete?
   - ➔ i.e. delete [] p; // what’s the size?
3. Size of class A is promoted to 16 Bytes (multiple of sizeof(int))



What did “newOp.cpp” tell you?

1. Must have a destructor... somehow....
   - Try comment out the constructor in newOp.cpp...
2. Size of “new[]” = array size + sizeof(size_t)
3. Size of class A is promote to 16 Bytes (multiple of sizeof(int))
4. The ptr in new[] () points to the “-sizeof(size_t)” address
5. The ptr in new[] caller points to the array begin
6. The ptr in delete[] () points to the “-sizeof(size_t)” address
7. The ptr in delete[] caller points to the array begin

➔ In this example, memory is explicitly allocated by “malloc()” (or new char[numBytes])
(Will the constructor and destructor be called?)

➔ What if we want to use a memory manager (for chunk alloc and delete)?
(Who returns the pointers of new and delete?)

➔ Can the “new()”, “delete()” be non-static member functions?

Closer look at "new A" and "delete A"

- A *a = new A;
  1. A::operator new() is called
  2. Constructor of A is called
  3. The return pointer address is copied to 'a'
- A *a = new A[10];
  - Similar to "A *a = new A" except that 10 constructors are called
- delete a;
  1. Destructor of A is called
  2. A::operator delete() is called
- delete []a;
  - Similar to "delete a" except that several destructors are called

8/4-Byte Aligned

- In the previous example, the size of data members in A is 4 + 4 + 4 + 2 = 14. However...
  - sizeof(A) = 16
  - The parameters to new() and new[10] are 16 and 168
- But, if the class A is changed to:

```
class A {
   char   _data[14];
};
```

  - sizeof(A) = 14
  - The parameters to new() and new[10] are 14 and 148

  ➔ NOT 8-Byte aligned!!

Issues about Memory Manager

1. Number of memory blocks
   - Continuous or non-continuous
2. Overload of new/delete operators
   - Use new/delete or customized alloc()/free()
3. Memory manager association (by type or id)
4. Recycle or not
   - Garbage collection?

Memory Manager Association

- Remember we define a memory manage to distribute memory?
- Which memory manager to call when you allocate a memory in new/delete operator?
  (i.e. instead of calling "malloc()" and "free()" directly...)

```
➔ void* new(size_t t) {
        ... memMgr->alloc(t); ...}
➔ void operator delete (void* p) {
        ... memMgr->free((T*)p);
```

~~Is "memMgr" a data member?~~
Is "memMgr" a global variable?

1. Declared as “static” Data Member

```
class MyClass {
   static MemoryMgr *const  _mem_s;
public:
   void* operator new(size_t t) {
      _mem_s->alloc(t); }
};
```

➔ Each class is associated with an unique memory manager

➔ What if new/delete operators are not overloaded?

➔ What if we want to associate more than 1 memory managers for a class? (i.e. 1 class ➔ n memMgr)

- [Reason] Can have options to free portion of the memory
- Swap with other memory manager (bookkeeping needed)
- Who control this??

2. Use a Global Map(class id, MemManager);

```
class MyClass {
   static int const _mem_id_s;
public:
   void* operator new(size_t t) {
      ::globalMemMap[mem_id_s]
      ->alloc(t);
   }
};
```

➔ Memory manager association is controlled by a global function/class

Memory Manager Association

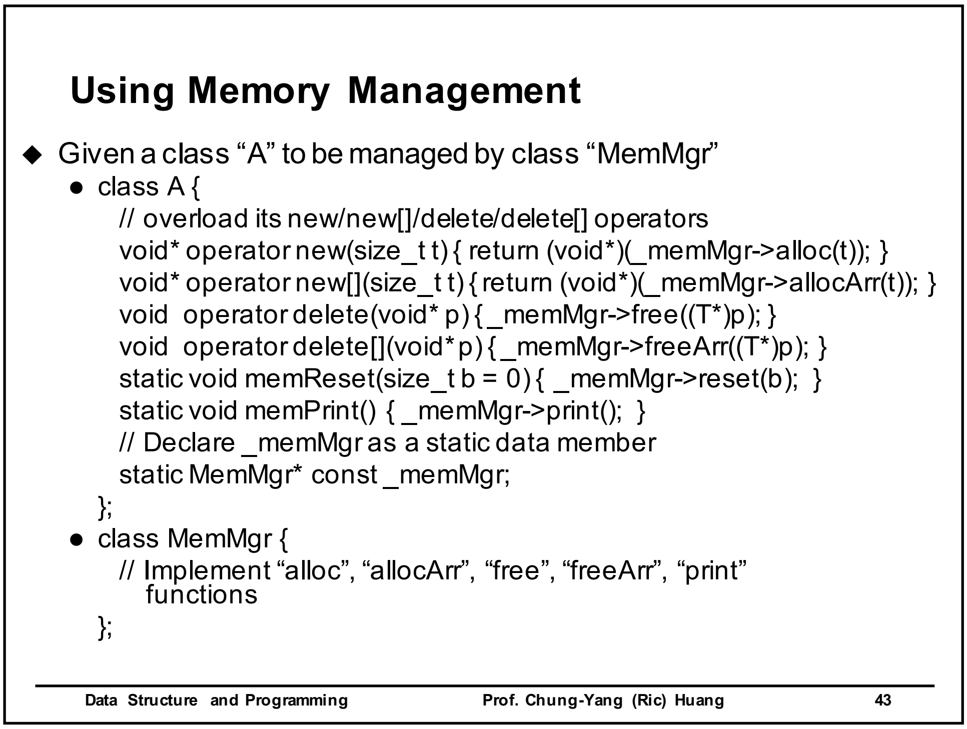
- We know...
  - Static data member must be initialized in .cpp code
  - e.g. MemoryMgr *const A::_mem_s = new ...
- Can we associate the memory managers of 2 different classes to the same one?

  (i.e. n classes → 1 memMgr)
  - i.e. Share the same memory manager
  - Any problem? (Answered later)

HW#4 Implementation of class MemMgr

```
template <class T>
class MemMgr {
private:
   size_t                 _blockSize;
   MemBlock<T>*       _activeBlock;
   MemRecycleList<T>
                    _recycleList[R_SIZE];
};

template <class T>
class MemBlock {
   friend class MemMgr<T>;
   char*                  _begin;
   char*                  _ptr;
   char*                  _end;
   MemBlock<T>*       _nextBlock;
};
```

```
template <class T>
class MemRecycleList {
friend class MemMgr<T>;
   // the array size of the recycled data
   size_t       _arrSize;
   // the first recycled data
   T*            _first;
   // next MemRecycleList
   //      with _arrSize + n*R_SIZE
   MemRecycleList<T>*  _nextList;
};
```



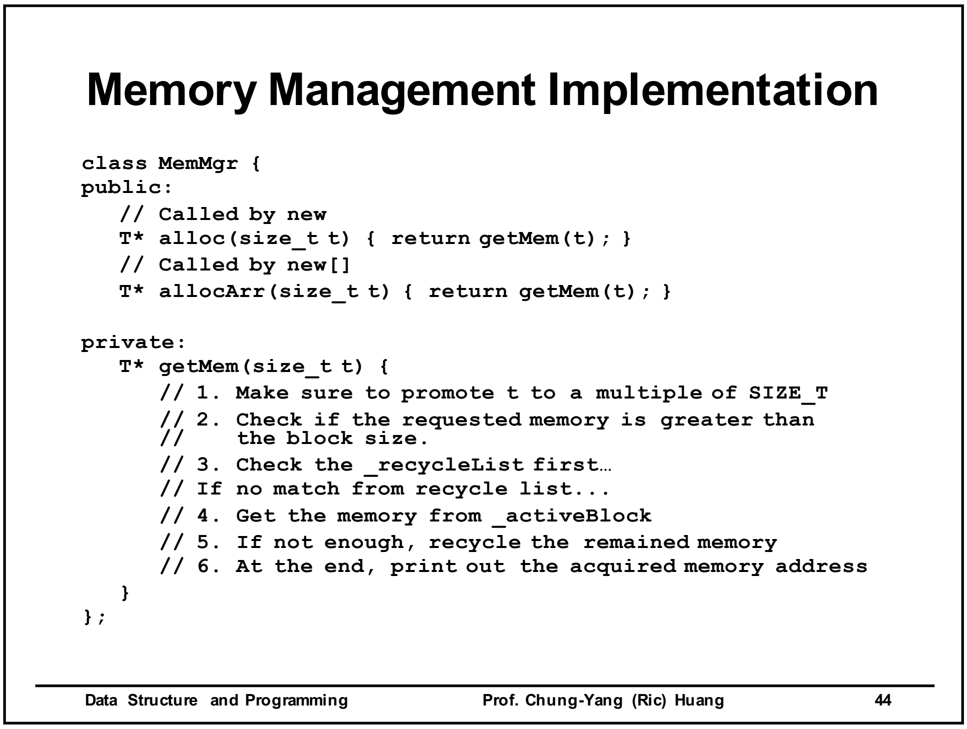
Using Memory Management

- Given a class "A" to be managed by class "MemMgr"
  - class A {

```
   // overload its new/new[]/delete/delete[] operators
   void* operator new(size_t t) { return (void*)(_memMgr->alloc(t)); }
   void* operator new[](size_t t) { return (void*)(_memMgr->allocArr(t)); }
   void  operator delete(void* p) { _memMgr->free((T*)p); }
   void  operator delete[](void* p) { _memMgr->freeArr((T*)p); }
   static void memReset(size_t b = 0) { _memMgr->reset(b);  }
   static void memPrint() { _memMgr->print();  }
   // Declare _memMgr as a static data member
   static MemMgr* const _memMgr;
};
```

  - class MemMgr {

```
   // Implement "alloc", "allocArr", "free", "freeArr", "print"
      functions
};
```



Memory Management Implementation

```
class MemMgr {
public:
   // Called by new
   T* alloc(size_t t) { return getMem(t); }
   // Called by new[]
   T* allocArr(size_t t) { return getMem(t); }

private:
   T* getMem(size_t t) {
      // 1. Make sure to promote t to a multiple of SIZE_T
      // 2. Check if the requested memory is greater than
      //    the block size.
      // 3. Check the _recycleList first…
      // If no match from recycle list...
      // 4. Get the memory from _activeBlock
      // 5. If not enough, recycle the remained memory
      // 6. At the end, print out the acquired memory address
   }
};
```

Do you remember.... mem manager is

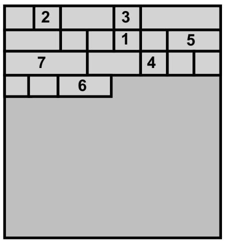
- Allocate a big chunk of memory from the system at a time
  - Distribute memory to the pointer variables by the memory manager
- No need to free pointers one by one; free the whole chunk at once
  - Return memory to system when mission is completed
    - ➔ Possibly memory leak-free
  - [Optional] Freed pointer memory is recorded in the recycle list (no deletion); can be used for later memory request

Do we call “delete”?
What if we don’t call “delete”?
(If so, will the destructor be called?)

Why do we need to overload “delete”?
What does it do?
Can we NOT overload “delete”?
(Either recycle or not recycle)

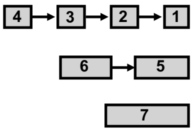
Recycling Memory

- If the usage is:
  new → new[] → new… → delete → delete[]…
  → new[] → delete… → …
  - That is, interleaving memory requests and frees
  - We should be able to “recycle” freed memory for later memory requests



Memory block

Recycle list



What if a “new” is called?

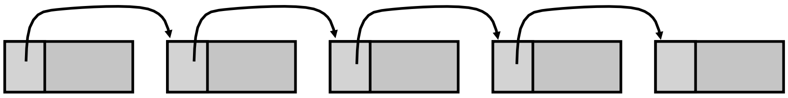
Recycling Memory

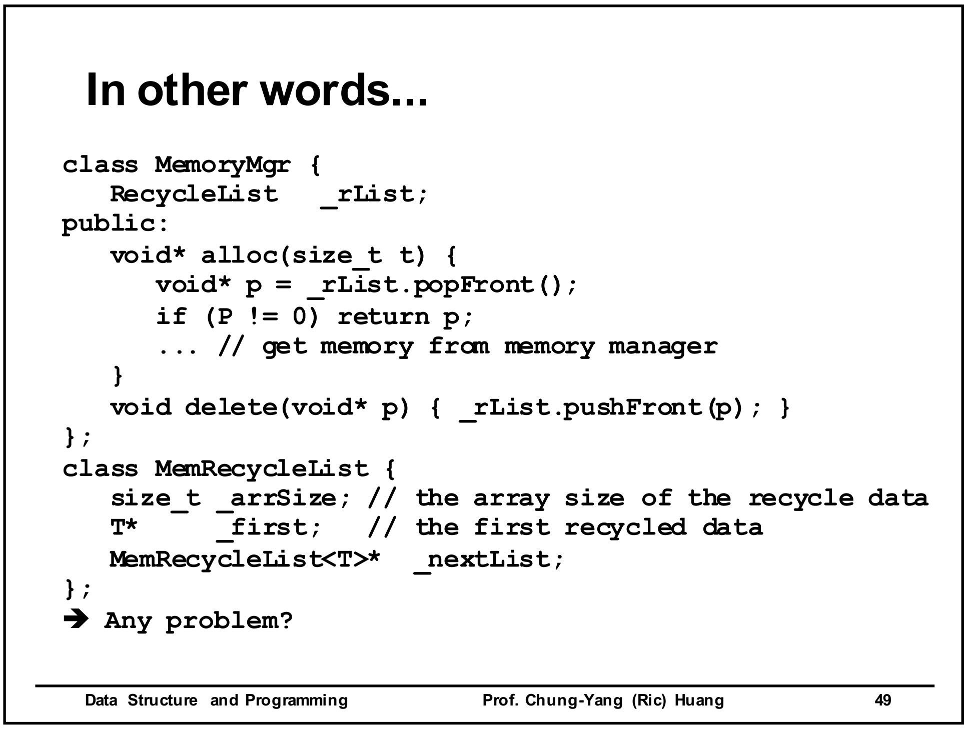
- Freed pointer memory is recorded in the recycle list (no deletion); can be used for later memory request

  How?? Use a “linked list” container class? (No!! extra memory)

1. In the memory manager, keep a recycleList that points to the first recycled memory
2. Reuse the first 4 or 8 Bytes (why?) of each recycled memory, pointing to the address of the next recycled memory

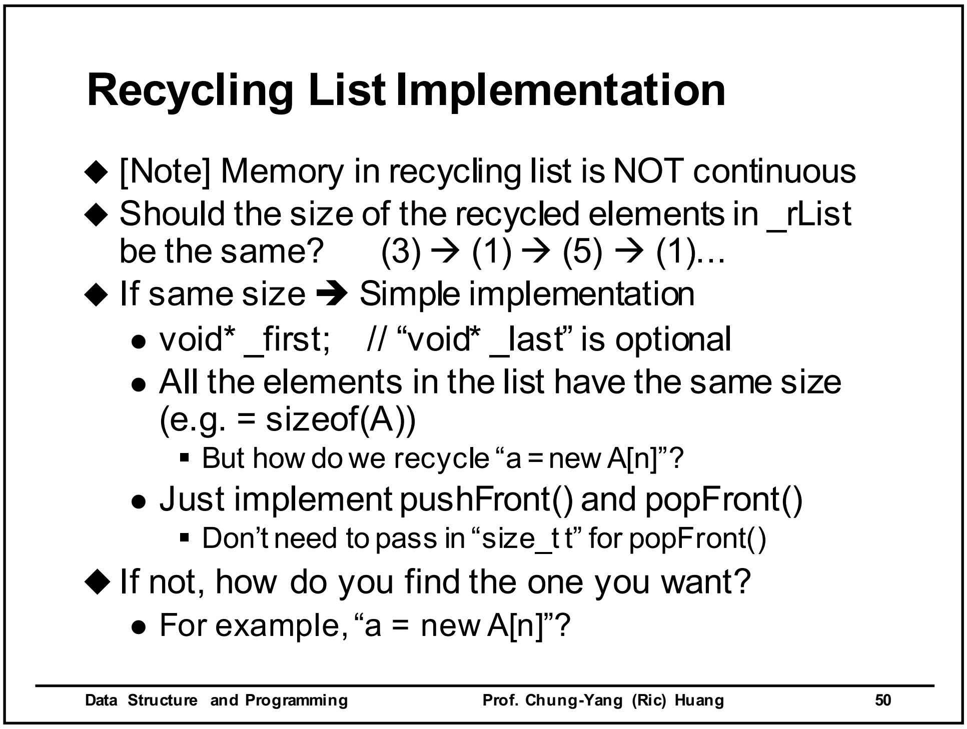
[Restriction]
The size of the managed class should be >= 4 (or 8) Bytes





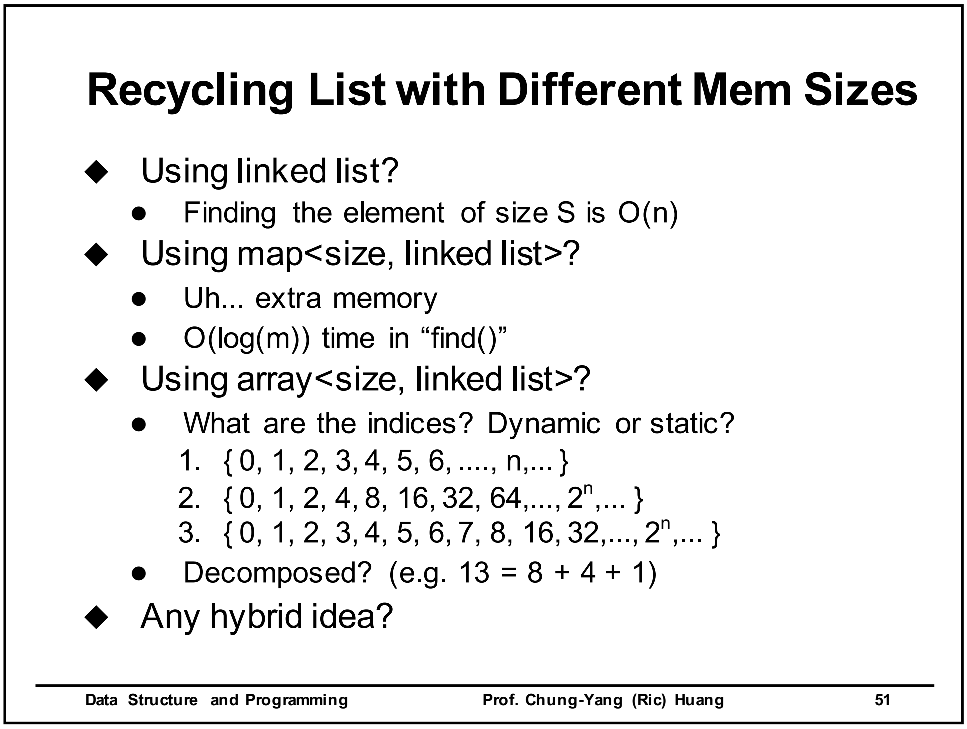
In other words...

```
class MemoryMgr {
   RecycleList   _rList;
public:
   void* alloc(size_t t) {
      void* p = _rList.popFront();
      if (P != 0) return p;
      ... // get memory from memory manager
   }
   void delete(void* p) { _rList.pushFront(p); }
};
class MemRecycleList {
   size_t _arrSize; // the array size of the recycle data
   T*     _first;   // the first recycled data
   MemRecycleList<T>*  _nextList;
};
➔ Any problem?
```



Recycling List Implementation

- [Note] Memory in recycling list is NOT continuous
- Should the size of the recycled elements in _rList be the same? (3) → (1) → (5) → (1)…
- If same size ➔ Simple implementation
  - void* _first; // “void* _last” is optional
  - All the elements in the list have the same size (e.g. = sizeof(A))
    - But how do we recycle “a = new A[n]”?
  - Just implement pushFront() and popFront()
    - Don’t need to pass in “size_t t” for popFront()
- If not, how do you find the one you want?
  - For example, “a = new A[n]”?



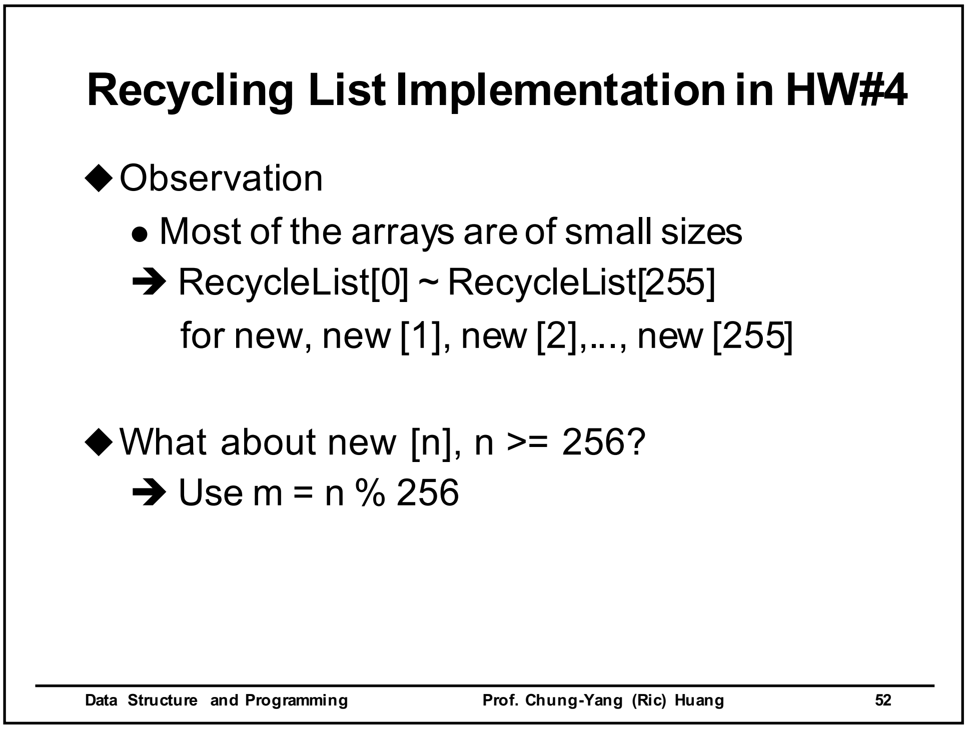
Recycling List with Different Mem Sizes

◆ Using linked list?
- Finding the element of size S is O(n)

◆ Using map<size, linked list>?
- Uh... extra memory
- O(log(m)) time in “find()”

◆ Using array<size, linked list>?
- What are the indices? Dynamic or static?
  1. { 0, 1, 2, 3, 4, 5, 6, ..., n,... }
  2. { 0, 1, 2, 4, 8, 16, 32, 64,..., 2^n,... }
  3. { 0, 1, 2, 3, 4, 5, 6, 7, 8, 16, 32,..., 2^n,... }
- Decomposed? (e.g. 13 = 8 + 4 + 1)

◆ Any hybrid idea?



Recycling List Implementation in HW#4

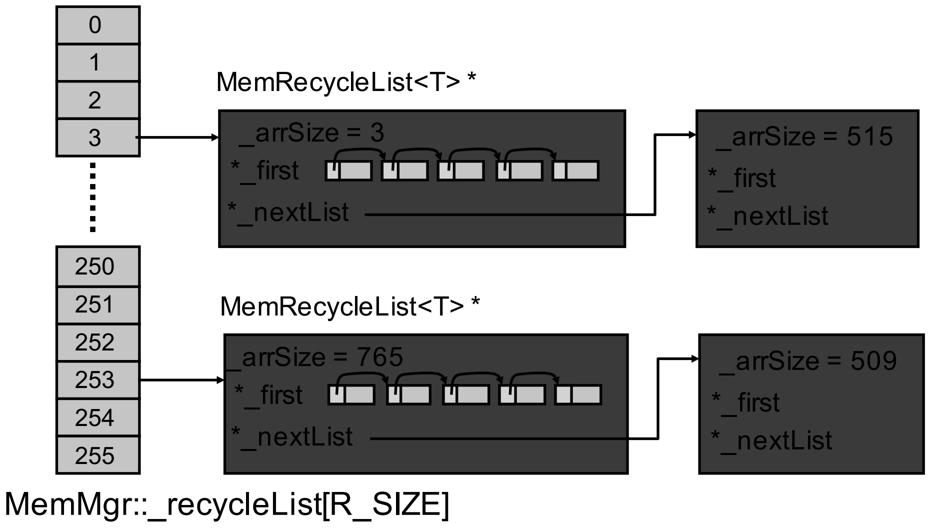
◆ Observation
- Most of the arrays are of small sizes

➔ RecycleList[0] ~ RecycleList[255]
for new, new [1], new [2],..., new [255]

◆ What about new [n], n >= 256?

➔ Use m = n % 256

Data Structure



Memory Management Implementation

```
class MemMgr {
public:
   // Called by delete
   void  free(T* p) { getMemRecycleList(0)->pushFront(p); }
   // Called by delete[]
   void  freeArr(T* p) {
      // Get the array size 'n' stored by system,
      // which is also the _recycleList index
      // add to proper recycle list...
   }
private:
   MemRecycleList<T>* getMemRecycleList(size_t n) {
      size_t m = n % R_SIZE;
      // Go through _recycleList[m], its _nextList, and etc,
      //    to find a recycle list whose "_arrSize" == "n"
      // If not found, create a new MemRecycleList with
      //    _arrSize = n and add to the last MemRecycleList
      // So, should never return NULL
   }
};
```

Recycling List with Different Classes

- ◆ What if we want to associate the same memory manager to different classes?
  (i.e. n classes → 1 memMgr)
  (e.g. class Inheritance?)
  - ● What would be the mem size in the recycling list? GCD of sizes of A & B? Multiple of sizeof(size_t)?

  ➔ More difficult to manage...!!

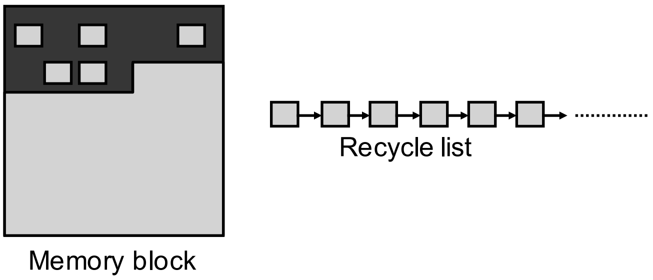
  ➔ Suggest to use memory management without recycling

Memory Management without Recycling

```
◆ Overload "new" to get memory from memMgr
◆ Overload "delete" to do nothing but calling destructor
   ➔ No recycle

class MemoryBlock {
   char*         _begin;
   char*         _ptr;
   char*         _end;
   MemoryBlock*  _nextBlock;
public:
   MemoryBlock(size_t B) { _begin = (char*)malloc(B); }
};
class MemoryMgr {
   MemoryBLock*   _activeBlock;
   void* alloc(size_t) {
      // get memory from _activeBlock;
      // If over the limit, new MemoryBlock as _activeBlock }
};
```

Garbage Collection

- After using memory management for a while, we may have many recycled memory pieces but not much required memory



- Can we rearrange the pointers so that the freed memory can be put together and even returned to system earlier?
  - Pointer value changes? How to keep the associations?
  - Index or pointer?
  - Too many to cover; beyond the scope of this class...

Conclusion

- Memory related problems are mostly runtime problems
  - You won’t see them during compilation
  - Crash during runtime ➔ difficult to debug
  - But please use debugger instead of “cout”
- Use memory manager to allocate a block of memory instead of piece by piece
  - Don’t need to worry about freeing individual memory ➔ no memory leak
  - Still need to properly issue “delete” if the callings of destructors are needed
  - Can achieve better memory locality and thus better performance