COMP2012 Object-Oriented Programming and Data Structures

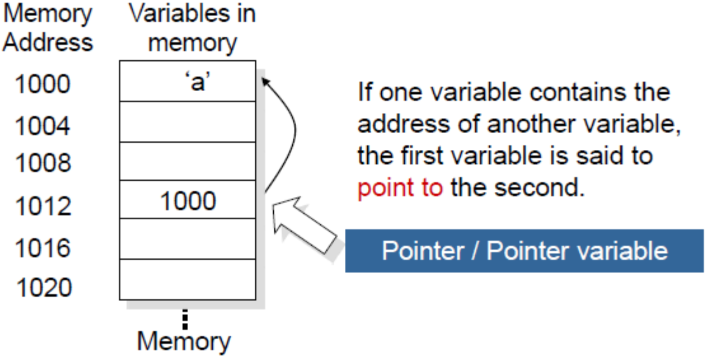
Review: Pointers

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

What are Pointers?

- A pointer or pointer variable is a variable that holds a memory address of another object (typically another variable) in memory



Declaration of Pointer Variables

- If a variable is going to hold an address of another variable, it must be declared as follows:

Syntax:

```
<type>* <variable name>;
<type> *<variable name>;
```

OR

where <type> is the type of the variable address that the pointer variable can store (e.g. int, char, double, user-defined type), <variable name> is the name of the pointer variable

- Actually, we can treat $<$type$>$ $*$ as a special type which is pointer type

- Recall the syntax for declaring a pointer variable:
  <type> * <variable name>;
- Examples:

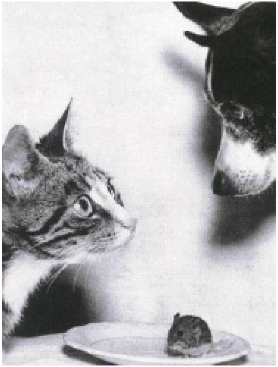
```
// Declare a pointer that points to an int variable
int* a;    // the value of a is garbage but it is NOT nullptr

// Declare a pointer that points to a double variable
double* b; // the value of b is garbage but it is NOT nullptr

// Declare a pointer that points to a char variable
char* c;   // the value of c is garbage but it is NOT nullptr

// It is no difference for you to put * close to type OR
// close to variable name
int* d;    // the value of d is garbage but it is not nullptr
int *d;    // same as above, no difference
```

We will talk a bit more about nullptr pointer later!

Pointer Operator & (Address-Of)

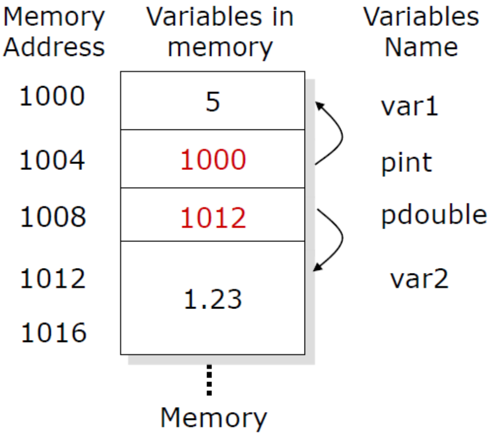
- There are two operators associated with pointers. They are & and * (Note: The * here doesn't mean multiplication)
- The first operator, & is a unary operator (i.e. with single operand) that returns the memory address of a variable
  - Usage: &<variable name>
- We can think of & as returning "the address of"

```
int var1 = 5;

// pint receives the address of var1
int* pint = &var1;

double var2 = 1.23;

// pdouble receives the address of var2
double* pdouble = &var2;
```



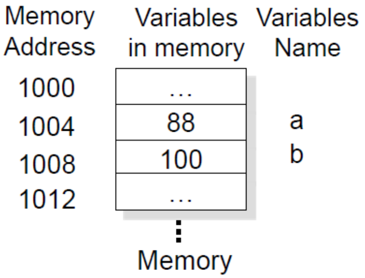
Pointer Operator & (Address-Of)(Cont'd)

- Graphical representation of last example



Example - & (Address-Of)

```
#include <iostream>
using namespace std;

int main() {
  int a, b;
  a = 88;
  b = 100;
  cout << "The address of a is " << &a << endl;
  cout << "The address of b is " << &b << endl;
  return 0;
}
```

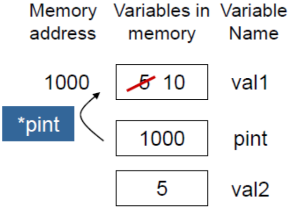


Output:

```
The address of a is 0x22ff74
The address of b is 0x22ff70
```

Pointer Operator * (Dereference)

- The second operator, *, is the complement of operator &
- It is also a unary operator but accesses the value located at the address that the pointer points to
- We can think of * as "at address"




```
int var1 = 5;
int* pint = &var1;

// var2 receives the value of the memory
// location pointed by pint
int var2 = *pint;

// Change the value of the memory location
// pointed by pint to 10, therefore var1 = 10 as well
*pint = 10;
```

The Different Uses of Operator *

- Do not confuse the use of operator * in declaring a pointer variable versus the use of operator * as the dereference operator
- Example

```
// This means to declare a pointer
// variable
int* p;

int i, j = 10;
p = &j;
```

```
// This means to dereference the pointer variable p
i = *p;
```

Pointer Assignments

- As with any variable, you may use a pointer variable on the right-hand side of an assignment statement to assign its value to another pointer variable placed on the left-hand side

```cpp
#include <iostream>
using namespace std;

int main() {
  int x;
  int *p1, *p2;

  p1 = &x; // Address of x is assigned to p1

  // Content of p1 (which is the address of x)
  // is assigned to p2
  p2 = p1;
  cout << "The address of x: " << p2 << endl;
  return 0;
}
```

Example of Pointers

```
#include <iostream>
using namespace std;

int main() {
  int value1 = 5, value2 = 15;
  int *p1, *p2; // Remember to add * before p2!
  p1 = &value1; // p1 = address of value1
  p2 = &value2; // p2 = address of value2
  *p1 = 10;     // value of variable pointed by p1 = 10
  *p2 = *p1;    // value of variable pointed by p2 =
                // value of variable pointed by p1
  p1 = p2;      // p1 = p2 (pointer value copied)
  *p1 = 20;     // value of variable pointed by p1 = 20
  cout << "value 1 = " << value1 << " / value2 = " << value2;
  return 0;
}
```

Output:

```
value1 = 10 / value2 = 20
```

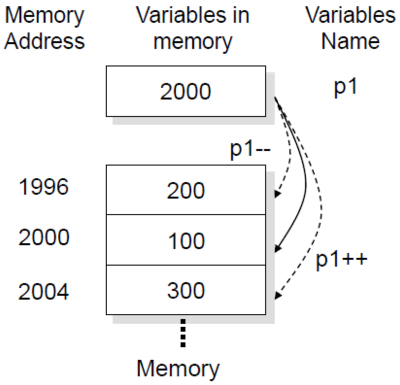
Pointer Arithmetic

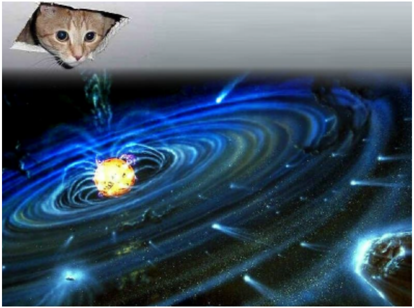
- ONLY TWO arithmetic operations are applicable on pointers. They are
  - Addition
  - Subtraction

  Therefore, C++ supports four operators for pointer arithmetic operations. They are $+$, $-$, $++$ and $--$
- To understand what occurs in pointer arithmetic, let `p1` be an int pointer with current value of 2000. Also, assume ints are 4 bytes long, after the expression `p1++`,
  - `p1` contains 2004, NOT 2001
- The same is true of decrements. For example, assuming that `p1` has the value 2000, after the expression `p1--`,
  - `p1` has the value 1996

Pointer Arithmetic

- Graphical representation of the last example





Pointer Arithmetic

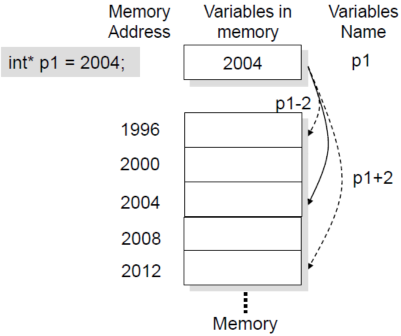
- Generalizing from preceding example, the following rules govern pointer arithmetic
  - Each time a pointer is incremented, it points to the memory location of the next element of its base type
  - Each time a pointer is decremented, it points to the memory location of the previous element of its base type
  - When applied to character pointers, this will appear as "normal" arithmetic because characters are always 1 byte long
  - All other pointers will increase or decrease by the length of the data type they point to

Pointer Arithmetic (Cont'd)

- You are not limited to the increment and decrement operators
- For example, you may add or subtract integers to or from pointers
  - The expression
    p1 = p1 + 2;
    makes p1 point to the second element of p1's type beyond the one it currently points to
  - The expression
    p1 = p1 - 2;
    makes p1 points to the second element of p1's type precede the one it is currently points to

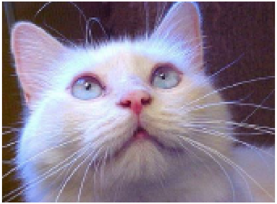
Pointer Arithmetic

- Graphical representation of the last example



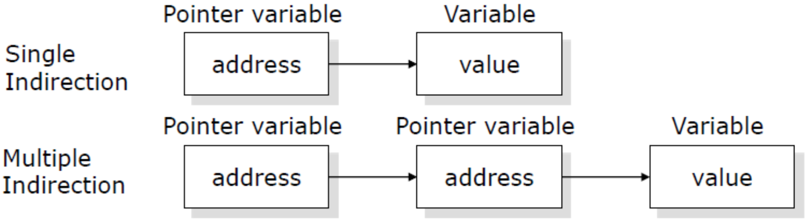
Pointer Comparisons

- We can compare two pointers in a relational expression
- For instance, given two pointers (i.e., pointer variables), p and q, the following statements are perfectly valid
  - ```
    if(p < q)
      cout << "p points to lower memory than q" << endl;
    ```
  - ```
    if(p > q)
      cout << "p points to higher memory than q" << endl;
    ```
  - ```
    if(p == q)
      cout << "p points to the same memory as q" << endl;
    ```
- Generally, pointer comparisons are used when two or more pointers point to a common objects


    ```

Multiple Indirection (Pointer to Pointer)

- You can have a pointer points to another pointer that points to the target value
- This is called "multiple indirection" or "pointer to pointer"
- Pointer to pointer can be confusing. The figure below helps clarify the concept of multiple indirection



Multiple Indirection (Pointer to Pointer) (Cont'd)

- As you can see, the value of a normal pointer is the address of the object that contains the value
- In the case of pointer to pointer, the first pointer contains the address of second pointer, which points to the object that contains the value desired
- A variable that is a pointer to pointer can be declared as:

Syntax:

<type>** <variable name>;

```
// An int variable i stores the value 10
int i = 10;

// A pointer variable ptr stores the address of i
int* ptr = &i;

// A pointer variable p_ptr stores the address
// of another pointer variable ptr
int** p_ptr = &ptr;
```

Multiple Indirection (Pointer to Pointer)

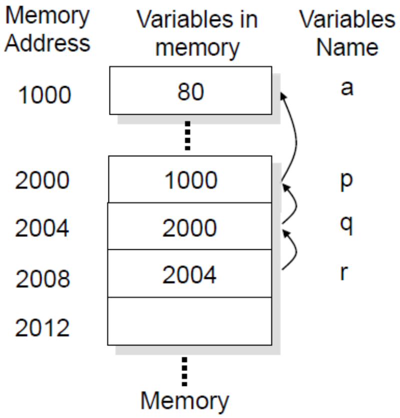
- Multiple indirection can be carried on to whatever extent required, but more than a pointer to a pointer is rarely needed
- In fact, excessive indirection is difficult to follow and prone to conceptual errors

Seldom use multiple indirections, i.e., more than pointer to pointer! :D

Example of Multiple Indirection

```
#include <iostream>
using namespace std;

int main() {
  int a = 80;
  int* p = &a;
  int** q = &p;
  int*** r = &q;
  int**** s = &r;
  cout << a << " ";
  cout << *p << " ";
  cout << **q << " ";
  cout << ***r << " ";
  cout << ****s << endl;
  return 0;
}
```



Output:

80 80 80 80 80

Arrays and Pointers

- There is a close relationship between pointers and arrays
- An array name is actually a constant pointer to the first element of the array
- A constant pointer means we cannot change the content of pointer variable

```
#include <iostream>
using namespace std;

int main() {
  int a[5];
  cout << "Address of a[0]: " << &a[0] << endl
       << "Name as pointer: " << a << endl;
  return 0;
}
```

Output:

```
Address of a[0]:  0x22ff50
Name as pointer:  0x22ff50
```

Question ;)

- Can we do something like the following?

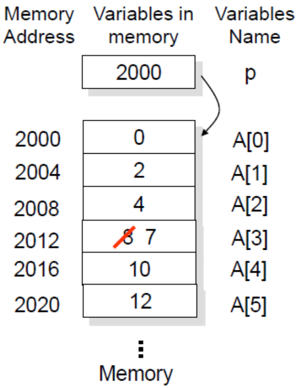
```
int a = 10;
int* p = &a;
int A[6] = { 0, 2, 4, 8, 10, 12 };
A = p; // Can we do this?
```

- No! Since A is a constant pointer

Arrays and Pointers

```
// Defines an array of ints
int a = 10;
int* p = &a;
int A[6] = { 0, 2, 4, 8, 10, 12 };
p = A; // Can we do this?
```

- Since array names and pointers are equivalent, we can also use p as the array name
- For example:
  `p[3] = 7;` or `*(p+3) = 7;`
  is equivalent to
  `A[3] = 7;`



Arrays and Pointers

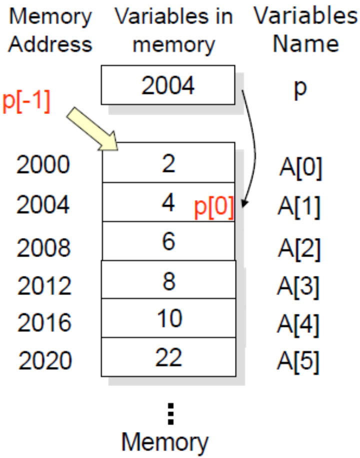
- Example:

```
#include <iostream>
using namespace std;

int main() {
  int A[6] = { 2, 4, 6, 8, 10, 22 };
  int* p = &A[1];
  cout << A[0] << " " << p[-1];
  cout << " ";
  cout << A[1] << " " << p[0];
  return 0;
}
```

Output:

2 2 4 4



Dereference Array Pointers

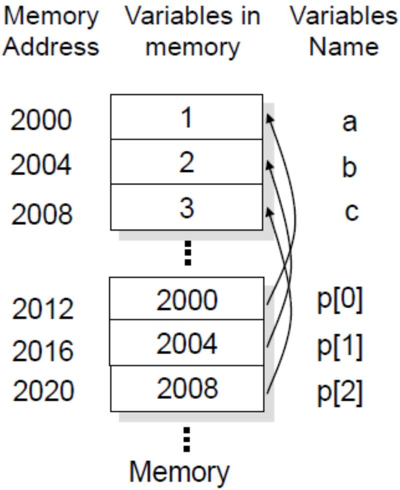
- As array name is a constant pointer, dereference operator (*) can be used on it
  - A[0] is same as *(A + 0)
  - A[1] is same as *(A + 1)
  - A[2] is same as *(A + 2)

    ⋮
  - In general, A[n] is equivalent to *(A + n)

Array of Pointers

- Pointers may be arrayed like any other data type
- Example:

```cpp
#include <iostream>
using namespace std;

int main() {
  int a = 1, b = 2, c = 3;
  int* p[3];
  p[0] = &a;
  p[1] = &b;
  p[2] = &c;
  return 0;
}
```



Pointer with nullptr literal

- A pointer with nullptr literal is a pointer that is currently pointing to nothing
- Often pointers are set to predefined pointer literal nullptr to make them null pointer
- Example:

```
#include <iostream>
using namespace std;

int main() {
  int* p = nullptr;
  if(!p)
    cout << "p is a nullptr pointer" << endl;
  return 0;
}
```

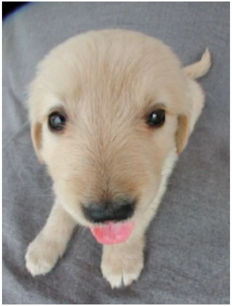
Pointer with nullptr literal (Cont'd)

- We will get an error if we try to access a nullptr pointer
- Example:

```
#include <iostream>
using namespace std;

int main() {
  int* p;
  p = nullptr;
  cout << p << endl;  // prints 0
  cout << &p << endl; // prints address of p
  cout << *p << endl; // runtime error!
}
```

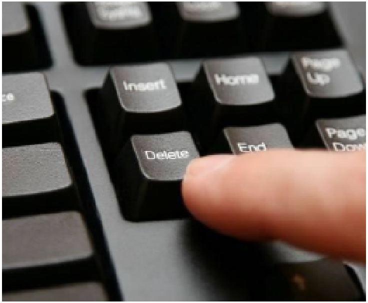
Memory Allocation

- If we know prior to the execution of the program, the amount and type of memory that we need, we can allocate memory statically prior to program start-up (i.e., compilation time)
  - We call this static memory allocation
- However, we cannot always determine how much memory we need before our programs run
  - For example: The length of an array or number of structures may not be known until your executing program determines what these values should be
  - So, what should we do?
    We need dynamic memory allocation

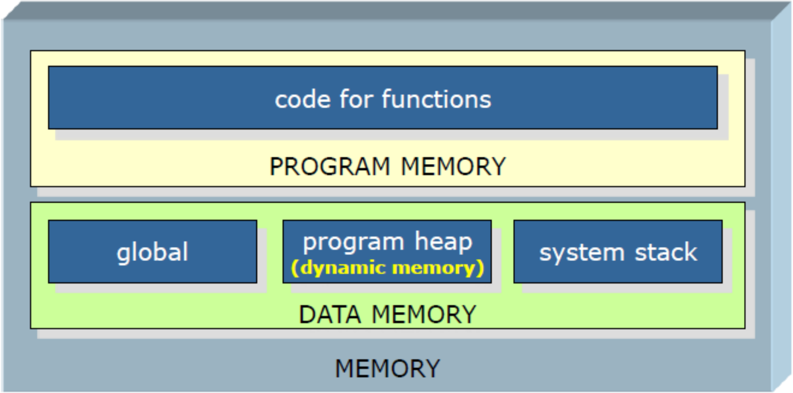


Memory Allocation (Cont'd)

- In C++, we can request memory from operating system at runtime and we call this dynamic memory allocation
  - An area of memory called the heap (or free store) is available in the run-time environment to handle dynamic memory allocation
  - In C++ programs, we can use operator new to allocate memory from heap and operator delete to release heap memory



Conceptual View of Memory



Heap is a special area of memory which is reserved for dynamic variables

Memory Allocation (Cont'd)

- Static Memory Allocation
  - Memory is allocated at compilation time
  - The following fragment allocates memory for x, y and p at compilation time
    - `int x, y; // x and y are integers`
    - `int* p; // p is an int pointer variable`
  - Memory is returned automatically when variable / object goes out of scope
- Dynamic Memory Allocation
  - Memory is allocated from heap at running time using new
  - Dynamic objects can exist beyond the function in which they were allocated
  - Memory is returned by a de-allocation request using delete operator

Dynamic Memory Allocation

Syntax:

```
<type>* <variable name> = new <type>;
```

where <type> is the type of the variable address that the pointer variable can store (e.g. int, char, double, user-defined type), <variable name> is the name of the pointer variable

- The new operator allocates memory from heap and returns a pointer to it
- If all memory is used up and new is unable to allocate memory, then it returns the value nullptr

Dynamic Memory Allocation (Cont'd)

- Example:

```
int* p;
p = new int;
```


```
// In a real programming situation, we should always
// check for this memory allocation error
if(p == nullptr) {
  cout << "Memory allocation not successful" << endl;
  exit(1); // Terminate the program with value = 1,
           // it means error!  :P
}
```

De-allocation of Memory

Syntax:

```
delete <pointer variable name>;
```

where <pointer variable name> is the variable name of a pointer variable stores an address of location in heap

- The system has a limited amount of space on the heap. In order to avoid using it up, it is a good idea to free UNUSED dynamic memory to the heap
  This is IMPORTANT!!!

new and delete

```
#include <iostream>
using namespace std;

int main() {
  int* p = new int; // allocate space from heap
  if(p == nullptr) { // or if(!p)
    cout << "Memory allocation not successful" << endl;
    exit(1);
  }
  *p = 100;
  cout << "At " << p << " ";
  cout << "is the value " << *p << endl;
  delete p;
  // Note that it DOES NOT modify p. After executing
  // delete p, the value of p is UNDEFINED
  return 0;
}
```

Output:

```
At 0x3d23f0 is the value 100
```

Allocating and De-allocating Dynamic Arrays

- The general forms of allocating dynamic array using new and delete are shown below

Syntax:

```
<type>* <pointer variable name> = new <type>[<size>];
delete [] <pointer variable name>;
```

where <type> is the type of data stored in an array,
<pointer variable name> is the variable name of a pointer variable, which stores an address of location in heap,
<size> is the number of elements needs to be allocated

- Note that <size> does not have to be a constant. It can be an expression evaluated at runtime
- The [] informs delete that an array is being released

Dynamic Array Example

```
#include <iostream>
using namespace std;

int main() {
  int* p;
  p = new int[10]; // allocate an array of a 10 ints
  if(p == nullptr) { // or if(!p)
    cout << "Memory application not successful" << endl;
    exit(1);
  }
  for(int i=0; i<10; ++i) {
    p[i] = i;
    cout << p[i] << " ";
  }
  delete [] p;    // release the array
  return 0;
}
```

Dynamic Array Example (Cont'd)

- Example: Need an array of unknown size

```
#include <iostream>
using namespace std;

int main() {
  int n;
  cout << "How many students? ";
  cin >> n;
  // The size of dynamic array is determined by user-input
  int* grades = new int[n];
  for(int i=0; i<n; ++i) {
    int mark;
    cout << "Input mark for student " << (i+1) << " : ";
    cin >> mark;
    grades[i] = mark;
  }
  // ...
  delete [] grades; // release the array
  return 0;
}
```

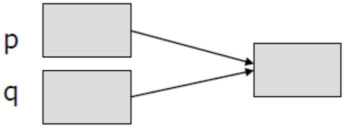
Dangling Pointer

- Dangling pointers are pointers which do not point to a valid object
- They arise when an object is deleted or de-allocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the de-allocated memory
- For example:

```
int* p;       // p is an int pointer variable
int* q;       // q is an int pointer variable
p = new int; // allocate memory from heap
q = p;
```

Dangling Pointer (Cont'd)

- The last example creates

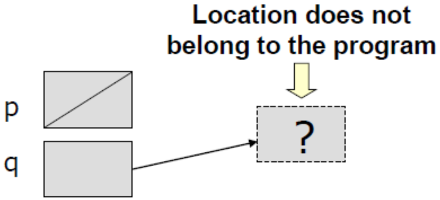


- But then executing

```
delete p;
p = nullptr;
```

leaves q dangling.

```
*q = 10; // illegal
```



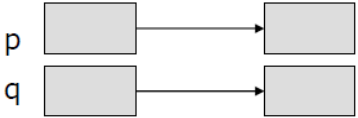
Memory Leakage

- A memory leak is what happens when we forgot to return a block of memory allocated with the new operator or make it impossible to do so, e.g., losing all pointers to an allocated memory location
- When this happens, the memory can never be de-allocated and is lost, i.e., never return to the heap
- For example

```
int* p;       // p is an int pointer variable
int* q;       // q is an int pointer variable
p = new int; // allocate memory from heap
q = new int; // allocate memory from heap
```

Memory Leakage (Cont'd)

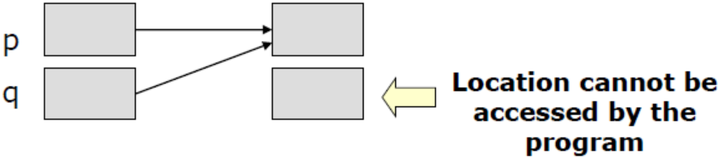
- The last example creates



- But then executing

  `q = p;`

  leaves the location previously pointed by q lost



Problem of Memory Leakage

- Memory leaks can seriously impact the ability of a program to complete its task
- It may be the case that subsequent dynamic memory requests cannot be satisfied because of insufficient heap memory
- For this reason, memory leaks should be avoided

Further Reading

- Read Chapter 8 of "C++ How to Program" or Chapter 4 of "C++ Primer" textbook

That's all!

Any question?