



Unit 3 (ch7, ch9)

Pointers, Arrays and Dynamic Memory

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electronics & Electrical Engr.
Nat'l Yang Ming Chiao Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nycu.edu.tw
<http://mseda.ee.nctu.edu.tw/jimmyliu>



Chien-Nan Liu, NYCUEE

Outline

- *3.1 Pointers*
- 3.2 Dynamic Memory Allocation
- 3.3 Pointer and Array

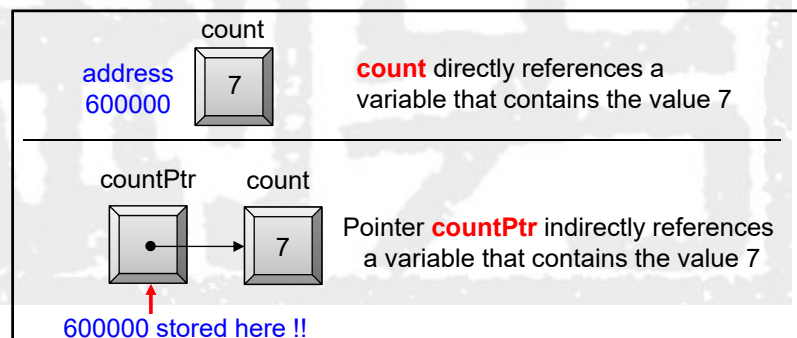


Pointers

- A pointer is the **memory address** of a variable
 - Pointers "point" to a variable by telling where the variable is located
- Memory addresses can be used to **access variables indirectly**
 - When a variable is used as a call-by-reference argument, its address is passed



Chien-Nan Liu, NYCUEE



3-3

Declaring Pointers

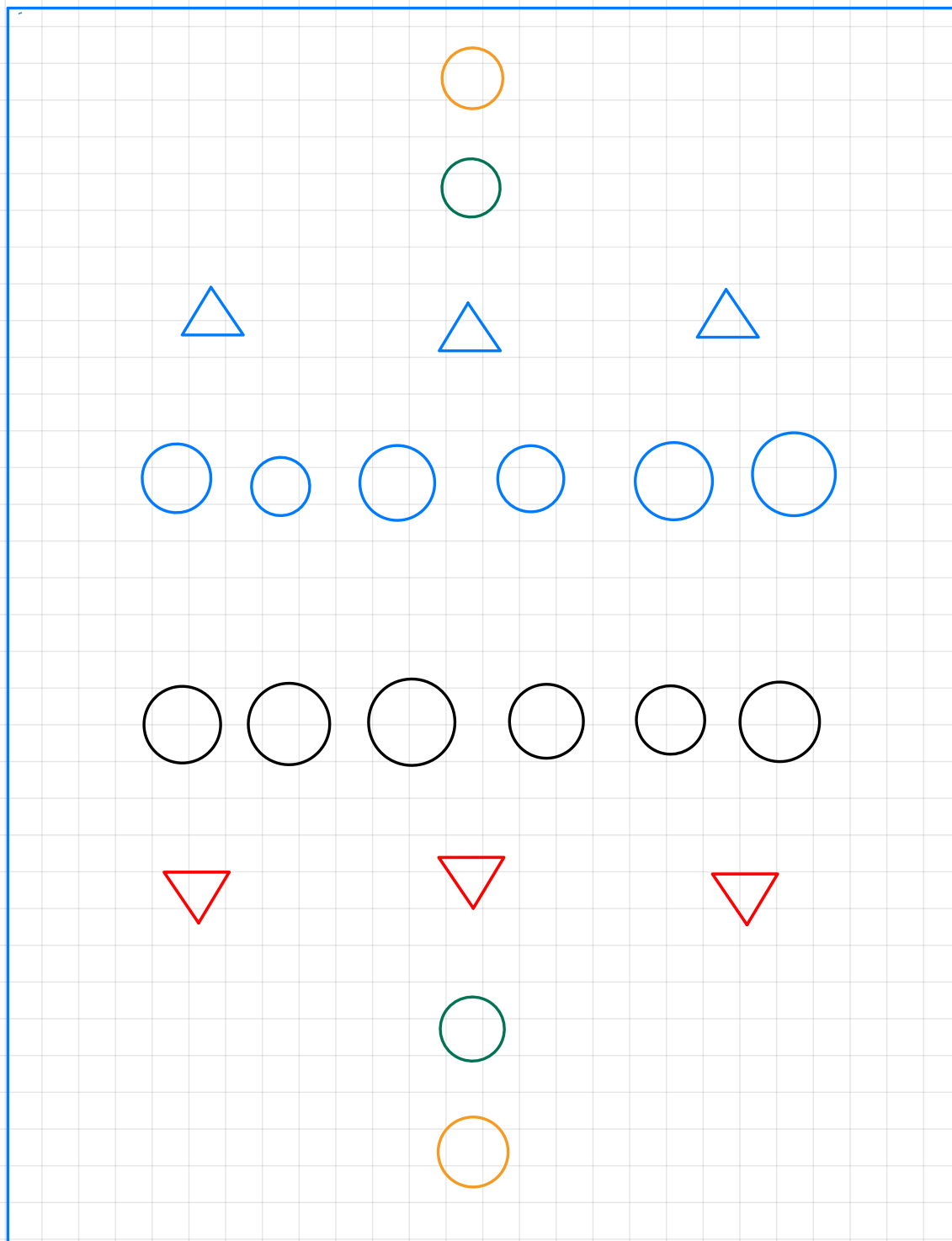
- Pointer variables must be declared to have a type
 - Define how to explain the data retrieved from memory
 - Ex: declare a pointer variable **p** that "point" to a double:
`double *p;`
 - The **asterisk(*)** identifies **p** as a pointer variable
- To declare multiple pointers in a statement, use the asterisk before each pointer variable
 - Pointer and non-pointer variables can be put together
 - Ex: `int *p1, *p2, v1, v2;`

p1 and **p2** **point to** variables of type *int*
v1 and **v2** are **variables** of type *int*



Chien-Nan Liu, NYCUEE

3-4



小隊員

小隊員

△ 藍隊旗點

▽ 紅隊旗點

○ 對手旗旗點

○ 黃金旗旗點

Operators on Pointer Variables

- The **& operator** can be used to obtain the address of a variable
 - Ex: `p1 = &v1;`
p1 is now a pointer to v1
v1 can be called v1 or
"the variable pointed to by p1"
- "The variable pointed to by p" is denoted as ***p** in C++
 - C++ uses the * operator in several different ways
 - With pointers, the * is dereferencing operator here
 - p is said to be dereferenced
 - p is the address, *p is the data



Chien-Nan Liu, NYCUEE

3-5

Example of Pointer Operations

- `v1 = 0;`
`p1 = &v1;` ←

v1 and *p1 now refer to the same variable

`*p1 = 42;`
`cout << "v1 = " << v1 << endl;`
`cout << "*p1 = " << *p1 << endl;`

output:

`v1 = 42`
`*p1 = 42`



Chien-Nan Liu, NYCUEE

3-6



Pointer Assignment

- The assignment operator = is used to assign the value of one pointer to another
 - Ex: If p1 still points to v1 (previous slide), then
`p2 = p1;`
causes *p2, *p1, and v1
be the same variable
- Some care is required making assignments to pointer variables
 - `p1 = p3;` // changes the **location** that p1 "points" to
 - `*p1 = *p3;` // changes the **value** at the location that
// p1 "points" to



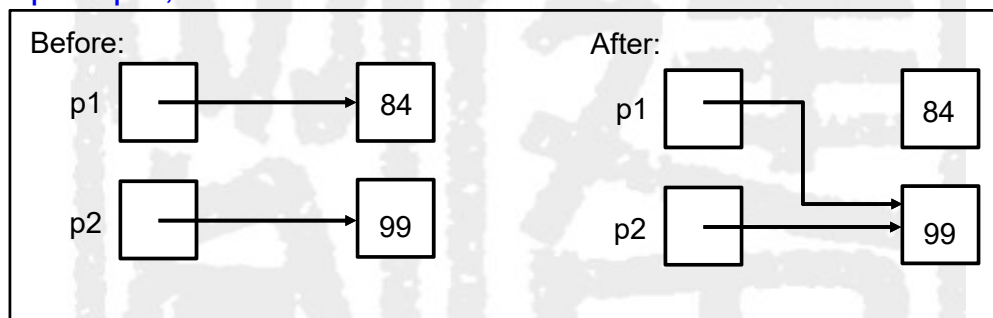
Chien-Nan Liu, NYCUEE

3-7

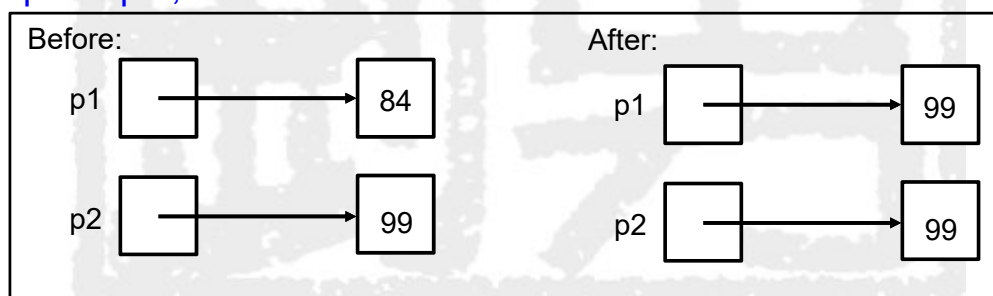


Uses of the Assignment Operator

`p1 = p2 ;`



`*p1 = *p2 ;`



Chien-Nan Liu, NYCUEE

3-8



Pass-by-Reference with Pointers

- In C++, we use **reference parameters** to pass arguments to a function by reference
- In old C, we implement pass-by-reference with **pointer** arguments
 - When calling a function with an argument that should be modified, the **address** of that argument is passed
 - Passing pointers to **large data objects** avoids the overhead of being passed by value
- Given the address of a variable, the **dereferencing operator (*)** form a synonym for it in the function
 - Used to modify the variable's value at that location in the caller's memory



Chien-Nan Liu, NYCUEE

3-9



Ex: Pass-by-Reference with Pointers

- In this example, the first function **cubeByValue** demonstrates typical pass-by-value mechanism
 - The local change has no affect on the original variable
- Another function **cubeByReference** passes the number by using pass-by-reference with a pointer
 - The address of that number is passed to the function
- The function **dereferences the pointer** and cubes the value to which nPtr points
 - This directly changes the value of *number* in main
- Graphical analysis for the execution of the programs is provided respectively



Chien-Nan Liu, NYCUEE

3-10

Comparison of Pass-by-Pointer

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

int cubeByValue( int );

int main( )
{
    int number = 5;
    cout << "The original value of number is "
        << number;
    // pass number by value
    number = cubeByValue(number);
    cout << "\nThe new value of number is "
        << number << endl;
} // end main

int cubeByValue ( int n )
{
    return n * n * n;
}
```

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

void cubeByReference( int * );

int main( )
{
    int number = 5;
    cout << "The original value of number is "
        << number;
    // pass the address of number
    cubeByReference(&number);
    cout << "\nThe new value of number is "
        << number << endl;
} // end main

void cubeByReference ( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
```

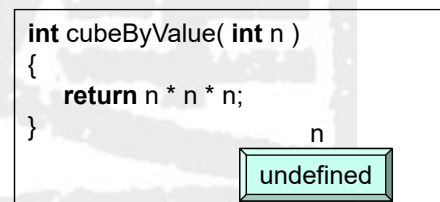
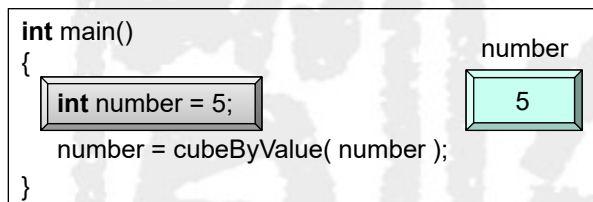


Chien-Nan Liu, NYCUEE

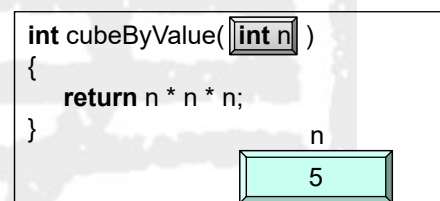
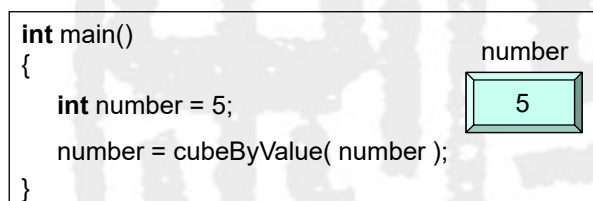
3-11

Pass-by-Value Analysis (1/2)

Step 1: Before **main** calls **cubeByValue**:



Step 2: After **cubeByValue** receives the call:



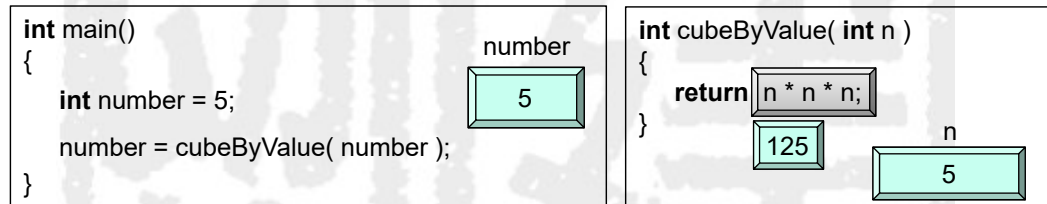
Chien-Nan Liu, NYCUEE

3-12

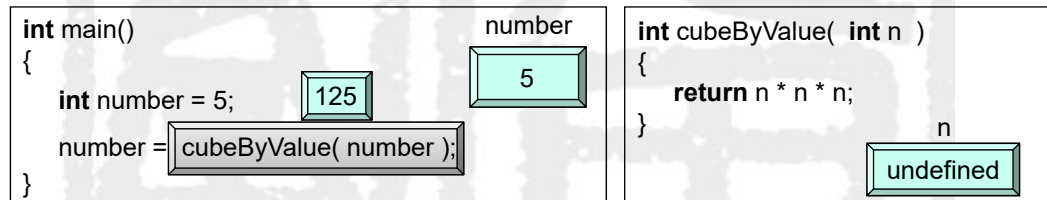


Pass-by-Value Analysis (2/2)

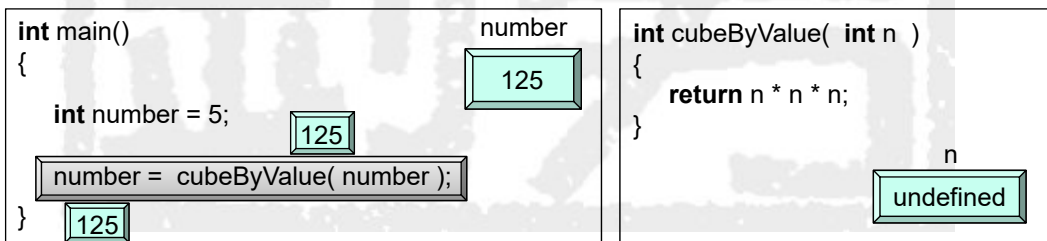
Step 3: After **cubeByValue** cubes parameter **n** and before **cubeByValue** returns to **main**:



Step 4: After **cubeByValue** returns to **main** and before assigning the result to **number**:



Step 5: After **main** completes the assignment to **number**:



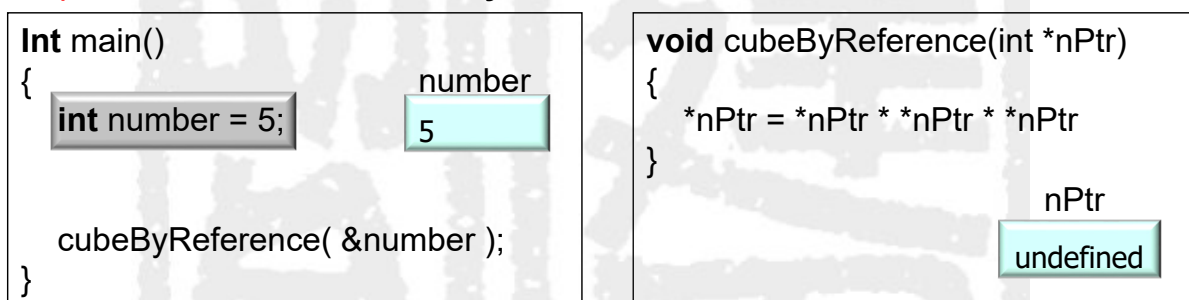
Chien-Nan Liu, NYCUEE

3-13

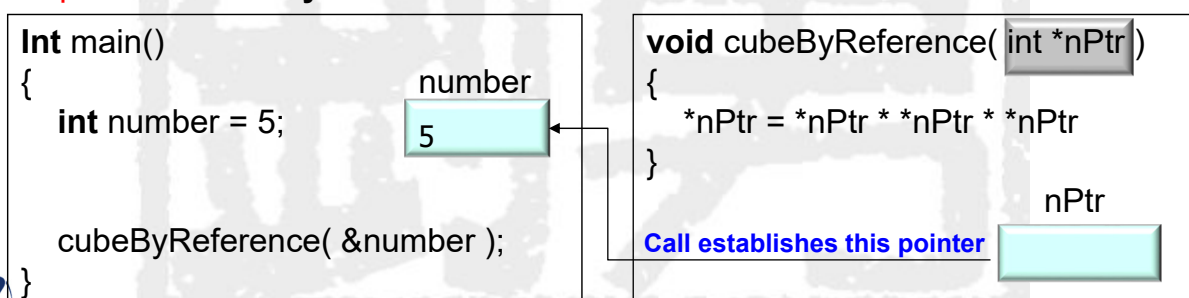


Pass-by-Reference Analysis

Step I: Before **main** calls **cubeByReference**:



Step II: After **cubeByReference** receives the call and before ***nPtr** is cubed:



Chien-Nan Liu, NYCUEE

3-14



Using const with Pointers

- There are 4 ways to pass a pointer to a function
- A **nonconstant pointer** to **nonconstant data**
ex: `int *myPtr = &x;`
 - Both the address and the data can be changed
- A **nonconstant pointer** to **constant data**
ex: `const int *myPtr = &x;`
 - Modifiable pointer to a **const int** (data are not modifiable)
- A **constant pointer** to **nonconstant data**
ex: `int *const myPtr = &x;`
 - Constant pointer to an **int** (data can be changed, but the address cannot)
- A **constant pointer** to **constant data**
ex: `const int *const Ptr = &x;`
 - Both the address and the data are not modifiable



Chien-Nan Liu, NYCU EE

3-15



Nonconstant Pointer to Constant Data

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

void f( const int * );

int main( )
{
    int y;
    // f attempts illegal modification
    f( &y );
} // end main
```

```
// xPtr cannot modify the value of constant variable
// to which it points
void f( const int *xPtr )
{
    *xPtr = 100;
} // end function f
```

Microsoft Visual C++ compiler error message:

```
c:\cpphttp7_examples\ch07\Fig07_10\fig07_10.cpp(17) :
error C3892: 'xPtr' : you cannot assign to a variable that is const
```

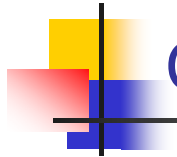
GNU C++ compiler error message:

```
fig07_10.cpp: In function `void f(const int*)':
fig07_10.cpp:17: error: assignment of read-only location
```



Chien-Nan Liu, NYCU EE

3-16



Constant Pointer to Nonconstant Data

```
int main( )
{
    int x, y;

    // The integer can be modified through ptr
    // But ptr always points to the same location
    int * const ptr = &x;

    *ptr = 7; // allowed: *ptr is not a constant
    ptr = &y; // error: ptr is a constant, cannot assign it a new address
} // end main
```

Microsoft Visual C++ compiler error message:

c:\cpphttp7_examples\ch07\Fig07_11\fig07_11.cpp(14) : error C3892: 'ptr' :
you cannot assign to a variable that is const

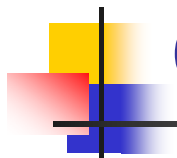
GNU C++ compiler error message:

fig07_11.cpp: In function `int main()':
fig07_11.cpp:14: error: assignment of read-only variable `ptr'



Chien-Nan Liu, NYCUEE

3-17



Outline

- 3.1 Pointers
- *3.2 Dynamic Memory Allocation*
- 3.3 Pointer and Array



Chien-Nan Liu, NYCUEE

3-18



Dynamic Variables -- new Operator

- Sometimes you need a **flexible** array or data structure to support dynamic requests
 - Traditional array requires a **fixed size** at compile time
 - Variables created using the **new** operator are called **dynamic variables**
 - Created and destroyed while the program is running
- Using **pointers**, variables can be manipulated even if there is no identifier for them
 - Ex: create a pointer to a "nameless" *int* variable
 - `int *p1 = new int;`
 - The new variable is referred to as ***p1**
 - *p1 can be used anyplace **an integer variable** can
 - Ex: `cin >> *p1; *p1 = *p1 + 7;`



Chien-Nan Liu, NYCU EE

3-19



Example: Pointer Manipulations (1/2)

```
//DISPLAY 9.2 Basic Pointer Manipulations
//Demonstrate pointers and dynamic variables.
```

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

```
int main( )
{
```

```
    int *p1, *p2;
```

```
    p1 = new int;
```

```
    *p1 = 42;
```

```
    p2 = p1;
```

```
    cout << "**p1 == " << *p1 << endl;
```

```
    cout << "**p2 == " << *p2 << endl;
```

```
    *p2 = 53;
```

```
    cout << "**p1 == " << *p1 << endl;
```

```
    cout << "**p2 == " << *p2 << endl;
```

```
    p1 = new int;
```

```
    *p1 = 88;
```

```
    cout << "**p1 == " << *p1 << endl;
```

```
    cout << "**p2 == " << *p2 << endl;
```

```
    cout << "Hope you got the point of this example!\n";
```

```
    return 0;
```

```
}
```

Sample Dialogue

```
*p1 == 42
```

```
*p2 == 42
```

```
*p1 == 53
```

```
*p2 == 53
```

```
*p1 == 88
```

```
*p2 == 53
```

```
Hope you got the point of this example!
```

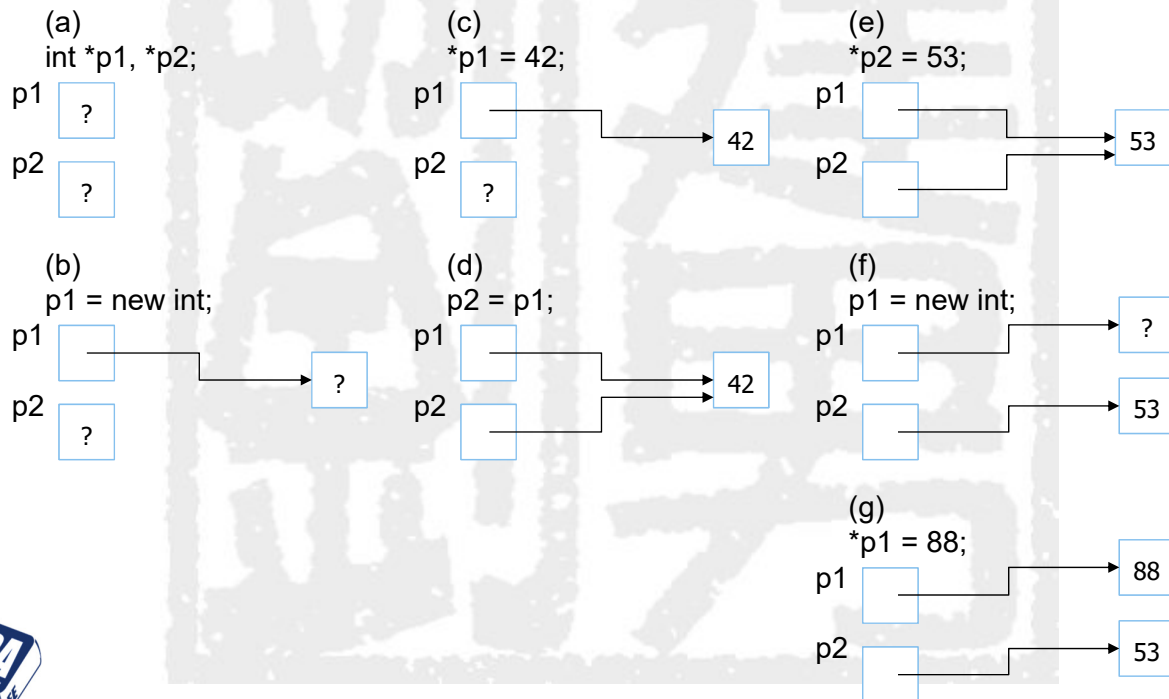


Chien-Nan Liu, NYCU EE

3-20

Example: Pointer Manipulations (2/2)

Explanation of Display 9.2



Chien-Nan Liu, NYCUEE

3-21

Memory Management -- delete Operator

- An area of memory called the **freestore** or the **heap** is reserved for dynamic variables
 - New dynamic variables use memory in the freestore
 - If all of the freestore is used, calls to *new* will **fail**
- Unneeded memory can be recycled
 - When dynamic variables are no longer needed, they can be deleted to return the memory they occupied
 - **Only dynamic memory** obtained through *new* operator can be recycled !!
- The **delete** operator returns the used memory
 - Ex: **delete p;**
 - The value of `p` is now **undefined** and the memory that `p` pointed to is **back in the freestore** (may be used later)



Chien-Nan Liu, NYCUEE

3-22



Dangling Pointers

- Using *delete* on a pointer variable destroys the dynamic variable pointed to
 - That memory location doesn't belong to you anymore
- If another pointer variable was pointing to the dynamic variable, that variable is also undefined
- Undefined pointer variables are called **dangling pointers**
 - Dereferencing a dangling pointer (*p) is usually a **disaster**
 - That address may be used by other variable → **illegal change**
 - Such runtime error is difficult to be caught



Chien-Nan Liu, NYCU EE

3-23



Automatic Variable v.s. Dynamic Variable

- Variables declared in a function are destroyed when the function ends
 - The creation and destruction of these **automatic variables** is **controlled automatically**
- Variable declared outside any function definition are **global variables**
 - Global variables are available **all the time**
- The programmer should **manually control** the creation and destruction of **dynamic variables**
 - Unless you delete the variable, that memory space is occupied even though you leave the function already



Chien-Nan Liu, NYCU EE

3-24



Type Definitions

- A name can be assigned to a type definition, then used to declare variables
 - Keyword *typedef* is used to define new type names
 - Syntax:
 - `typedef Known_Type_Definition New_Type_Name;`
 - `Known_Type_Definition` can be any type
- To avoid confusing between pointer variable and typical variable, define a pointer type name
 - Example: `typedef int* IntPtr;`
 - Defines a new type, `IntPtr`, for pointer variables containing pointers to `int` variables
 - `IntPtr p;` is equivalent to `int *p;`



Chien-Nan Liu, NYCUEE

3-25



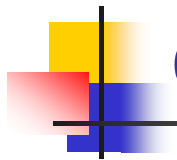
Confusing Pointer Declaration

- Type definition helps to prevent declaration error
 - `int *P1, P2;` // Only `P1` is a pointer variable
 - `IntPtr P1, P2;` // `P1` and `P2` are pointer variables
- A second advantage in using typedef to define a pointer type is seen in parameter lists
 - Example: `void sampleFunction(IntPtr& pointerVar);`
is less confusing than
`void sampleFunction(int*& pointerVar);`



Chien-Nan Liu, NYCUEE

3-26



Creating Dynamic Arrays

- Dynamic arrays are created using the new operator
 - Ex: create an array of 10 elements of type double

```
typedef double* DoublePtr;  
DoublePtr d;  
d = new double[10];
```



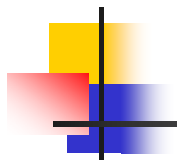
This could be an
integer variable!

- d can now be used as if it were an ordinary array!



Chien-Nan Liu, NYCUEE

3-27



Deleting Dynamic Arrays

- Pointer variable d is a pointer to a dynamic array
- When finished with the array, it should be deleted to return memory to the freestore
 - Syntax: `delete [] d;`
 - The brackets tell C++ a dynamic array is being deleted
→ need to know how many elements to remove
- Forgetting the brackets while deleting a dynamic array is not legal
 - It tells the computer to remove only one variable (first array element)



Chien-Nan Liu, NYCUEE

3-28



Example of a Dynamic Array

//Sorts a list of numbers entered at the keyboard.

```
#include <iostream>
```

```
#include <cstdlib>
```

```
#include <cstring>
```

```
typedef int* IntArrayPtr;
```

```
void fill_array(int a[], int size);
```

```
void sort(int a[], int size); // ascending order
```

```
int main( )
```

```
{
```

```
    using namespace std;
```

```
    cout << "This program sorts numbers  
            from lowest to highest.\n";
```

```
    int array_size;
```

```
    cout << "How many numbers will be sorted? ";
```

```
    cin >> array_size;
```

```
    IntArrayPtr a;
```

```
    a = new int[array_size];
```

```
    fill_array(a, array_size);
```

```
    sort(a, array_size);
```

```
    cout << "In sorted order the numbers are:\n";
```

```
    for (int index = 0; index < array_size; index++)
```

```
        cout << a[index] << " ";
```

```
    cout << endl;
```

```
    delete [] a;
```

```
    return 0;
```

```
}
```

```
void fill_array(int a[], int size)
```

```
{
```

```
    using namespace std;
```

```
    cout << "Enter " << size << " integers.\n";
```

```
    for (int index = 0; index < size; index++)
```

```
        cin >> a[index];
```

```
}
```

```
void sort(int a[], int size)
```

```
//Any implementation of sort may be used.
```



Chien-Nan Liu, NYCU

3-29



Dynamic Memory Allocation in C

- The functions to obtain and release memory during execution are put in library `<stdlib.h>`
- **malloc**
 - Takes number of bytes to allocate
 - Use *sizeof* to determine the size of an object
 - Returns pointer of type **void ***
 - A **void *** pointer may be assigned to any pointer
 - If no memory available, returns **NULL**
 - Example:

```
newPtr = malloc( n*sizeof(int) ); //n-element array
```
- When using *malloc*, test the return value for the **NULL** pointer
 - Give an error message if the memory allocation is failed



Chien-Nan Liu, NYCU

3-30



The sizeof Operator

- The unary operator *sizeof* determines the total size of a variable in bytes at compilation time
 - Can also be applied on an array, a constant, or a data type name
 - When applied to the name of an array, the *sizeof* operator returns the memory size of **total elements**
- When applied to a **pointer** parameter, the *sizeof* operator returns the size of the pointer in bytes
 - If it cannot recognize the array, only a single pointer variable is counted
- The number of elements in an array also can be determined using two sizeof operations
 - `sizeof realArray / sizeof(realArray[0])`



Chien-Nan Liu, NYCU EE

3-31



Return Dynamic Memory in C

- **free**
 - Deallocates memory allocated by `malloc`
 - Takes a pointer as an argument
 - `free (newPtr);`
- Freeing memory not allocated dynamically with `malloc` is an error
 - In C, manage memory by `malloc` \leftrightarrow `free`
 - In C++, manage memory by `new` \leftrightarrow `delete`
 - You cannot mix the two systems (ex: `malloc` + `delete`)
- Referring to memory that has been freed is a runtime error
 - You don't know whether the memory content is modified



Chien-Nan Liu, NYCU EE

3-32



Common Programming Error

- Not returning dynamically allocated memory when it is no longer needed can cause the system to run out of memory
 - This is sometimes called a “**memory leak**”
- **When memory that was dynamically allocated is no longer needed, use `free` to return the memory to the system **immediately****



Chien-Nan Liu, NYCU EE

3-33



Error-Prevention Tip

- After deleting dynamically allocated memory, set the pointer that referred **to that memory to 0**
 - This **disconnects the pointer** from the previously allocated space on the free store
- By setting the pointer to 0, the program **loses any access** to that free-store space
 - This space in memory could still contain information, despite having been deleted
- In fact, that space could have already been reallocated for a different purpose
 - If you didn't set the pointer to 0, your code could **inadvertently access this new information**
 - **Cause extremely subtle, nonrepeatable logic errors**



Chien-Nan Liu, NYCU EE

3-34



Outline

- 3.1 Pointers
- 3.2 Dynamic Memory Allocation
- *3.3 Pointer and Array*



Chien-Nan Liu, NYCUEE

3-35



Introduction to Arrays

- An array is used to process a **collection of data** of the **same type**
 - Examples: A list of names
 A list of temperatures
- Why do we need arrays?
 - Imagine keeping track of 5 test scores, or 100, or 1000 in memory
 - How would you name all the variables?
 - How would you process each of the variables?



Chien-Nan Liu, NYCUEE

3-36

Declaring an Array

- An array, named `score`, containing five variables of type *int* can be declared as
`int score[5];`
- This is like declaring 5 variables of type *int*:
`score[0], score[1], ... , score[4]`
- The value in brackets is called a **subscript** or an **index**
 - The index starts from 0, not 1 ...
- The variables making up the array are referred to as
 - **Indexed variables** or **elements of the array**
- The number of indexed variables in an array is the **size** of the array
 - The largest index is one less than the size
 - The first index value is zero



Chien-Nan Liu, NYCUEE

3-37

Using [] With Arrays

- In an array declaration, []'s enclose the size of the array
 - Ex: for an array of 5 integers → `int score [5];`
- When referring to one indexed variable, the []'s enclose a number identifying the indexed variable
 - Ex: `score[3]` is one of the indexed variables
 - The value in the []'s can be any expression that evaluates to one of the integers 0 to (size -1)
- To assign a value to an indexed variable, use the assignment operator:

`int n = 2;`

`score[n + 1] = 99;`

variable `score[3]`
is assigned 99



Chien-Nan Liu, NYCUEE

3-38

Loops And Arrays

- **for-loops** are commonly used to step through arrays

First index is 0

Last index is (size - 1)

- Example:


```
for (i = 0; i < 5; i++)
{
    cout << score[i] << " off by "
        << (max - score[i]) << endl;
}
```

could display the difference between each score and the maximum score stored in an array

- Enumeration can help to think about the behavior

- $i = 0 \rightarrow \text{cout} \ll \text{score}[0] \ll \text{" off by " } \ll \text{max} - \text{score}[0]$
- $i = 1 \rightarrow \text{cout} \ll \text{score}[1] \ll \text{" off by " } \ll \text{max} - \text{score}[1]$



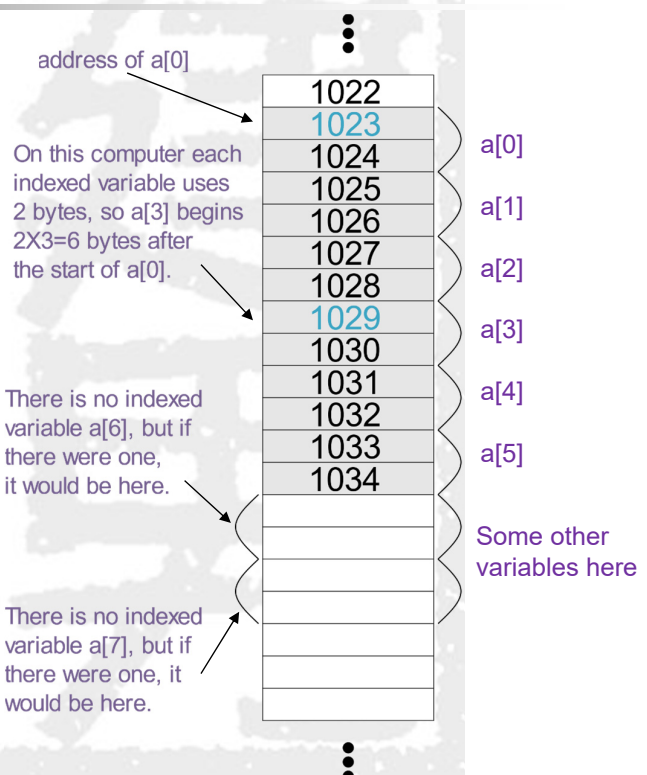
Chien-Nan Liu, NYCUEE

3-39

Arrays and Memory

- Declaring the array **int a[6]**

- Reserves memory for **six variables** of type **int**
- The variables are stored one after another (**consecutive locations**)
- Only the **address of a[0]** is remembered
- To determine the address of a[3]
 - Start at a[0]
 - Count the memory for three integers
 - Past enough memory to find a[3]



Chien-Nan Liu, NYCUEE

3-40

Out of Range Problems

- A common error is using a **nonexistent index**
 - Index values for `int a[6]` are the values 0 through 5, not 1 to 6
 - An index value not allowed by the array declaration is out of range, ex: using `a[7]` ??
- Using an out of range index value does not produce an error message!!
 - However, this address could be where some other variable is stored
 - May cause some unpredictable errors!!



Chien-Nan Liu, NYCUEE

3-41

Array and Pointer Variables

- In C/C++, only the **first element** of an array is remembered
 - Actually, array variables are **pointer variables** that **point to the first indexed variable** (ex: `a[0]`)
 - Example:

```
int a[10];
typedef int* IntPtr;
IntPtr p;
```

 - Variables `a` and `p` are the same kind of variable
 - `a` is a pointer variable that points to `a[0]` 即便宣告時沒用到*
- `p = a;` causes `p` to point to the same location as `a` → **point to `a[0]`**
 - Using `a` and `p` has the same effects



Chien-Nan Liu, NYCUEE

3-42

Pointer Variables as Array Variables

- In previous example, pointer variable *p* can be used as if it were an array variable
 - Example: *p*[0], *p*[1], ...*p*[9]
are all legal ways to use *p*
- Variable *a* can also be used as a **constant** pointer variable
 - But the pointer value in *a* cannot be changed
 - This is not legal:

```
IntPtr p2;  
...           // p2 is assigned a value  
a = p2; // attempt to change a
```



Chien-Nan Liu, NYCUEE

3-43

Example: Arrays and Pointers (1/2)

//Program to demonstrate that an array variable is a kind of pointer variable.

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

```
typedef int* IntPtr;
```

```
int main( )  
{
```

```
    IntPtr p;  
    int a[10];  
    int index;
```

```
    for (index = 0; index < 10; index++)  
        a[index] = index;
```

```
    p = a;
```

```
    for (index = 0; index < 10; index++)  
        cout << p[index] << " ";  
    cout << endl;
```

```
        for (index = 0; index < 10; index++)  
            p[index] = p[index] + 1;
```

```
        for (index = 0; index < 10; index++)  
            cout << a[index] << " ";  
        cout << endl;
```

```
    return 0;  
}
```

Output

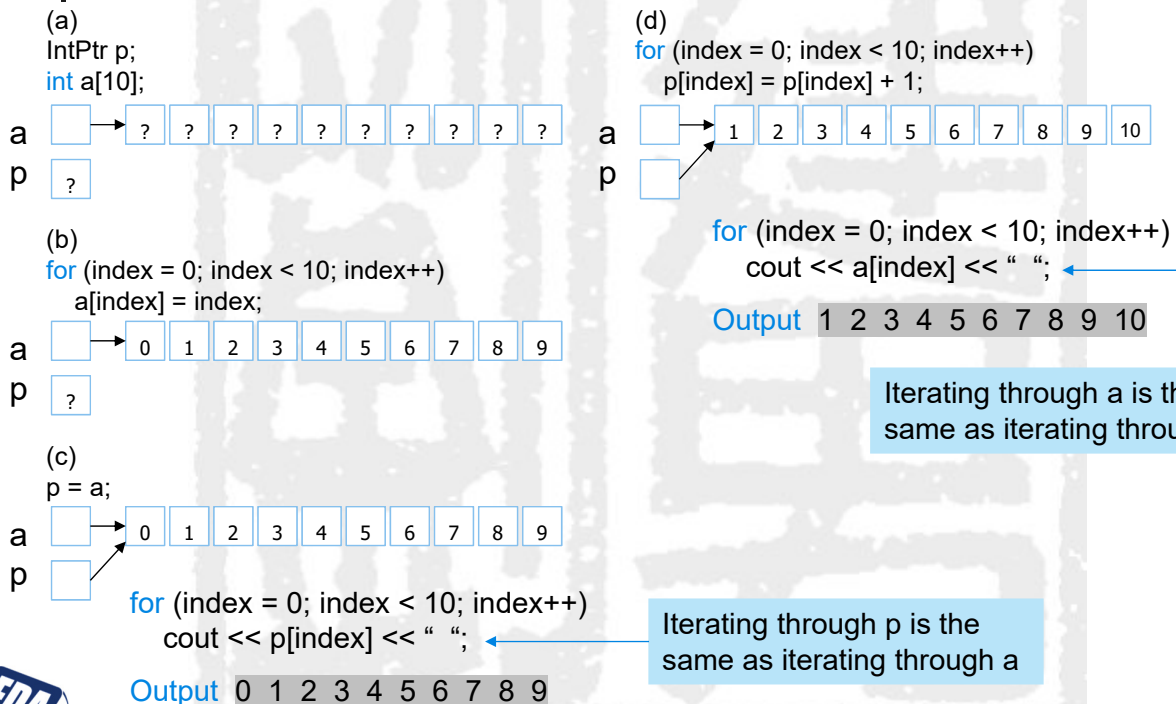
```
0 1 2 3 4 5 6 7 8 9  
1 2 3 4 5 6 7 8 9 10
```



Chien-Nan Liu, NYCUEE

3-44

Example: Arrays and Pointers (2/2)

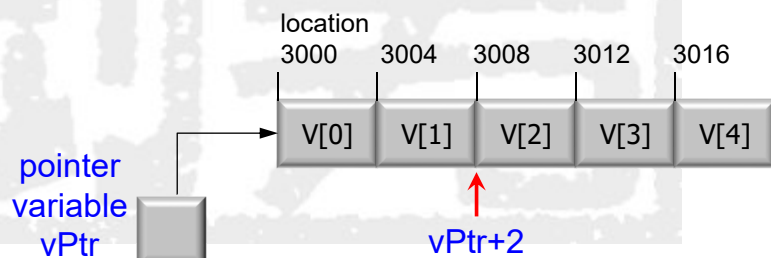


Chien-Nan Liu, NYCUEE

3-45

Pointer Arithmetic

- Arithmetic can be performed on the addresses contained in pointers
 - Given a dynamic array of integers, `v`, as shown below
 - If `int *vptr = v`, `vptr` points to `v[0]`
 - The expression `vptr+2` evaluates to the address of `v[2]`
 - Adding one adds enough bytes for one element in the array
 - This is NOT the arithmetic on the memory address !!
 - If `vPtr = 3000`, `vptr+2 = 3008`, not 3002 ...



Chien-Nan Liu, NYCUEE

3-46



Pointer Arithmetic Operations

- You can **add** and **subtract** with pointers
 - The **++** and **--** operators can be used
 - Two pointers of the same type can be subtracted to obtain the number of indexed variables between
 - The pointers should be **in the same array**!
 - Ex: If vPtr contains the address 3000 and v2Ptr contains the address 3008, **v2Ptr - vPtr = 2** in previous example
- Pointers can be compared using **equality** and **relational** operators
 - **Relational** operators apply on the pointers in the **same array**
 - Using equality operators, the pointers can be compared with the addresses stored in pointers
 - Ex: compare with **NULL** pointer → it points to nothing !!



Chien-Nan Liu, NYCU EE

3-47



Relationship Between Pointers and Arrays

- The array name (without a subscript) is a **constant pointer** to the first element of the array.
 - This pointer **cannot be modified** as a normal variable
- Pointers can do operations involving array indexing
- Assume the following declarations:
 - `int b[5]; // create 5-element int array b`
`int *bPtr; // create int pointer bPtr`
- We can set bPtr to the address of the first element in array b with the statement
 - `bPtr = b; // assign address of array b to bPtr`
- This is equivalent to
 - `bPtr = &b[0]; // also assigns address of array b to bPtr`



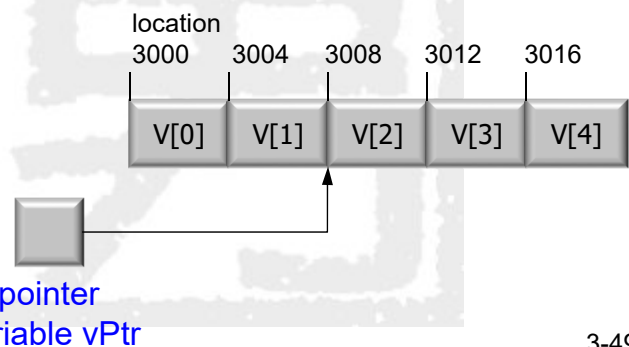
Chien-Nan Liu, NYCU EE

3-48

Relationship Between Pointers and Arrays

- Array element $b[2]$ can alternatively be referenced with the pointer expression
 - $*(bPtr + 2)$
- The 2 in the preceding expression is the **offset** to the pointer
- This notation is referred to as **pointer/offset notation**.
 - The parentheses are necessary, because the precedence of $*$ is higher than that of $+$

括號



Chien-Nan Liu, NYCUEE

3-49

Relationship Between Pointers and Arrays

- The address $\&b[3]$ can be written with the pointer expression $bPtr + 3$
 - Array elements can also be referenced with pointers
- The array name (which is implicitly const) can be treated as a pointer and used in pointer arithmetic
- For example, the expression $*(b + 3)$ also refers to the array element $b[3]$
- In general, all subscripted array expressions can be written with **a pointer and an offset**
 - For clarity, use array notation instead of pointer operation when manipulating arrays



Chien-Nan Liu, NYCUEE

3-50

Relationship Between Pointers and Arrays

- Pointers can be subscripted exactly as arrays can
- For example, the expression `bPtr[1]` refers to the array element `b[1]`; this expression uses **pointer/subscript notation**
- In summary, **four notations are discussed** for referring to array elements:
 - Array subscript notation,
 - Pointer/offset notation with the array name as a pointer,
 - Pointer subscript notation, and
 - Pointer/offset notation with a pointer



Chien-Nan Liu, NYCU EE

3-51

Code: Using Array Names and Pointers

```
//Using subscripting and pointer notations with arrays.
#include <iostream>
using namespace std;

int main()
{
    int b[] = { 10, 20, 30, 40 }; //create 4-element array b
    int *bPtr = b; // set bPtr to point to array bPtr

    // output array b using array subscript notation
    cout << "Array b printed with:\n\nArray subscript notation\n";

    for( int i = 0; i < 4; i++)
        cout << "b[" << i << "]" << b[i] << "\n";

    //output array b using the array name and pointer/offset notation
    cout << "\nPointer/offset notation where "
        << "the pointer is the array name\n";

    for( int offset1 = 0; offset1 < 4; offset1++)
        cout << "*(b+ " << offset1 << ") = " << *( b + offset1) << "\n";
```



Chien-Nan Liu, NYCU EE

3-52

Code: Using Array Names and Pointers

```
//output array b using bPtr and array subscript notation
cout << "\nPointer subscript notation\n";

for ( int j = 0; j < 4; j++)
    cout << "bPtr[" << j << "]" << bPtr[j] << "\n";

cout << "\nPointer/offset notation\n";

//output array b using bPtr and pointer/offset notation
for ( int offset2 = 0; offset2 < 4; offset2++)
    cout << "(bPtr+ " << offset2 << ") = "
        << *( bPtr + offset2) << "\n";

} // end main
```

Array b printed with:

Array subscript notation

b[0] = 10

b[1] = 20

b[2] = 30

b[3] = 40



Chien-Nan Liu, NYCUEE

3-53

Program Results

Pointer/offset notation where the pointer is the array name

*(b + 0) = 10

*(b + 1) = 20

*(b + 2) = 30

*(b + 3) = 40

Pointer subscript notation

bPtr[0] = 10

bPtr[1] = 20

bPtr[2] = 30

bPtr[3] = 40

Pointer/offset notation

*(bPtr + 0) = 10

*(bPtr + 1) = 20

*(bPtr + 2) = 30

*(bPtr + 3) = 40



Chien-Nan Liu, NYCUEE

3-54

Multi-Dimensional Arrays

- C++ allows arrays with multiple index values
 - `char page [30] [100];`
declares an array of characters named page
 - This array has two index values:
 - The first ranges from 0 to 29
 - The second ranges from 0 to 99
 - Each index is enclosed in its own brackets
 - Page can be visualized as an array of 30 rows and 100 columns
- C++ supports two dimensions only
 - For more dimensions, use index transformation
 - Ex: 2D transformation



Chien-Nan Liu, NYCUEE

	x=0	x=1	x=2	x=3
y=0	0	1	2	3
y=1	4	5	6	7
y=2	8	9	10	11

- x, y: reference addr.
- real index = $y*4 + x$

3-55

Multidimensional Dynamic Arrays

- Multidimensional arrays are **arrays of arrays**
 - In other words, they are actually **arrays of pointers**
- To create a 3x4 multidimensional dynamic array
 - First create a one-dimensional dynamic array

- Start with a new definition:

`typedef int* IntArrayPtr;`

- Now create a dynamic array of pointers named m:

`IntArrayPtr *m = new IntArrayPtr[3];` or `Int** m = new`

- For each pointer in m, create a dynamic array of int's

`for (int i = 0; i < 3; i++)
m[i] = new int[4];`

↑ return a pointer to the dynamic array

int* 指定 type

Int** m = new

第二个 * 表示
取值, 也就是
指向 size=3 的阵列

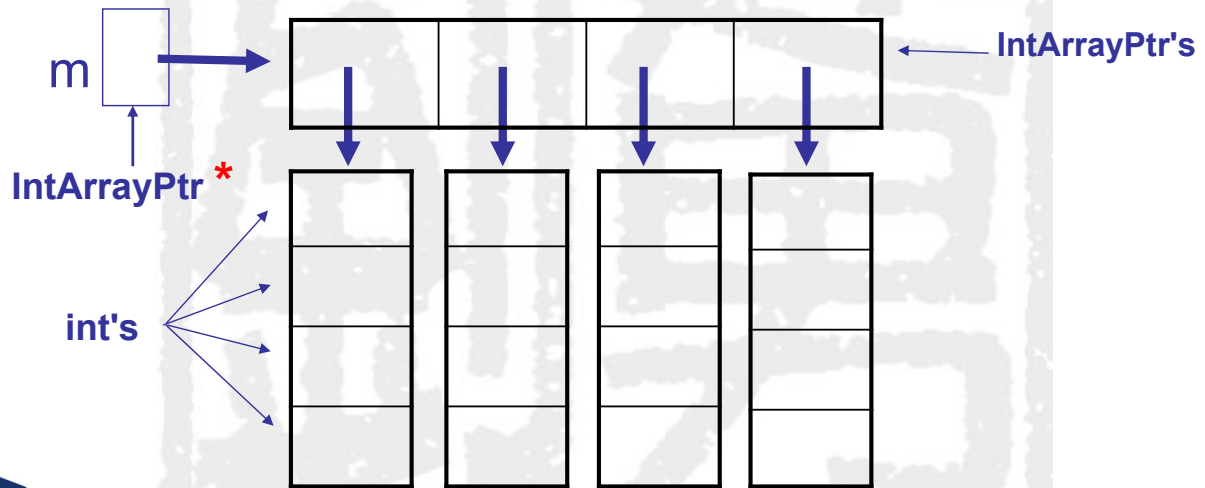


Chien-Nan Liu, NYCUEE

3-56

A Multidimensional Dynamic Array

- The dynamic array created on the previous slide could be visualized like this:



Chien-Nan Liu, NYCUEE

3-57

Deleting Multidimensional Arrays

- To delete a multidimensional dynamic array
 - Each call to `new` that created an array must have a corresponding call to `delete[]`
 - Example: To delete the dynamic array created on a previous slide:

for (i = 0; i < 3; i++)
 delete [] m[i] //delete the arrays of 4 int's
 delete [] m; // delete the array of IntArrayPtr's

(Handwritten notes in Chinese):
 - 要刪的東西是陣列 (The thing to be deleted is an array)
 - 該東西的名字 (The name of the thing)
 - Arrows point from the notes to the `m[i]` and `m` in the code.



Chien-Nan Liu, NYCUEE

3-58

Example of 2-D Dynamic Array

```
#include <iostream>
using namespace std;
typedef int* IntArrayPtr;

int main( )
{
    int d1, d2;
    cout << "Enter the row and column
        dimensions of the array:\n";
    cin >> d1 >> d2;

    IntArrayPtr *m = new IntArrayPtr[d1];
    int i, j;
    for (i = 0; i < d1; i++)
        m[i] = new int[d2];
    //m is now a d1 by d2 array.

    cout << "Enter " << d1 << " rows of "
        << d2 << " integers each:\n";

    for (i = 0; i < d1; i++)
        for (j = 0; j < d2; j++)
            cin >> m[i][j];
```



Chien-Nan Liu, NYCUEE

```
    cout << "Echoing the two-dimensional array:\n";
    for (i = 0; i < d1; i++)
    {
        for (j = 0; j < d2; j++)
            cout << m[i][j] << " ";
        cout << endl;
    }
    for (i = 0; i < d1; i++)
        delete[] m[i];
    delete[] m;
    return 0;
}
```

Sample Dialogue

```
Enter the row and column dimensions of the array:
3 4
Enter 3 rows of 4 integers each:
1 2 3 4
5 6 7 8
9 0 1 2
Echoing the two-dimensional array:
1 2 3 4
5 6 7 8
9 0 1 2
```

3-59

Demo Dynamic Array of Strings in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef char* StringPtr;

int main( void )
{
    int i, d1, d2, idx;
    char temp[80];
    printf("Enter the number of strings: ");
    scanf("%d", &d1);

    StringPtr *m = malloc(d1 * sizeof(StringPtr));
    for (i = 0; i < d1; i++)
    {
        printf("Enter string %d: ", i+1);
        scanf("%s", temp);
        d2 = strlen(temp);
        m[i] = malloc( (d2+1) * sizeof(char));
        strcpy(m[i], temp);
    }
    //m is now an array with d1 strings.
```



Chien-Nan Liu, NYCUEE

```
    printf("\nRandomly show 10 strings:\n");
    for (i = 0; i < 10; i++)
    {
        idx = rand()%d1;
        printf("%s\n", m[idx]);
    }

    for (i = 0; i < d1; i++)
        free( m[i] );
    free(m);

    return 0;
}
```

```
Enter the number of strings: 3
Enter string 1: test1
Enter string 2: test2
Enter string 3: test3

Randomly show 10 strings:
test2
test3
.....
```

3-60