# UEE1303
# Objective-Oriented Programming

C++_Lecture 02:

Pointers, References, and Dynamic Arrays

**C: How to Program 7th ed.**

**C++: How to Program (Late Objects Version) 7th ed.**

# Agenda

- Fundamentals of C/C++ pointer
  - difference between C and C++ pointers
- References
  - as a reference variables
  - pass to functions
- Using references and pointer with constants
- Dynamic memory allocation
  - dynamic arrays
- pass/return array to/from function

# Introduction

- C pointers are very powerful
  - but prone to error if not properly used
  - including system crashes
- C++ enhances C pointers and provides increased security because of its rigidity
  - by providing a new kind of pointer *reference*
- References have advantages over regular pointers when *passed to functions*

# C/C++ Pointer

- A pointer is a variable that is used to store a memory address
  - can be a location of variable, pointer, function
- Major benefits of using pointers in C/C++:
  - support *dynamic memory allocation*
  - provide the means by which functions can modify their *actual arguments*
  - support some types of *data structures* such as linked lists and binary trees
  - improve the *efficiency* of some program

# Pointer

- A pointer variable is declared using

```
int *ptr; //most suggested by textbook
int* ptr; //most convenient practically
int * ptr;
```

  - `'*'` must be located before each variable
- Data type that the pointer points can be any valid C/C++ type including `void` type and *user-defined* types

```
int *iPtr1;
double *dPtr2;
void *vPtr3; //can point to a variable
             //of any datatype
Robot *rPtr4; //user-defined datatype
```

# Operators for Pointers

- Indirection operator (*) precedes a pointer and returns the *value* of a variable
  - Address of the variable is stored in the pointer
  - *dereferencing*: access the value that the pointer points to
- Address-of operator (&) returns the memory *address* of its operand

```
float x = 1.23, y;
float *pt;    //point to any float variable
pt = &x;      //place x’s address into pt
cout << *pt; //print x’s value 1.23
```

# Pointer Expressions

- Pointers can be used as operands in assignment, arithmetic (only + and -), and comparison expressions

The address of f is 1000
The size of float is 8 bytes

```
float f = 13.3, *p1, *p2;
p2 = p1 = &f;    //p1 and p2 point to f
p1--;            //decrementing pl
cout << pl;      //pl is 992=1000-8
p2 += 5;         //add 5 and assign to p2
cout << p2;    //p2 is 1040=1000+5*8
if (p1 == p2){  //compare two addresses
    cout << "Two addresses are the same"
         << endl;
}
```

# Assignment of Pointers

- Pointer variables can be assigned:

```
int *p1, *p2;
p1 = p2; //ex: address of p2 is 5678
```

  - assign one pointer to another
  - make `p1` point to where `p2` points

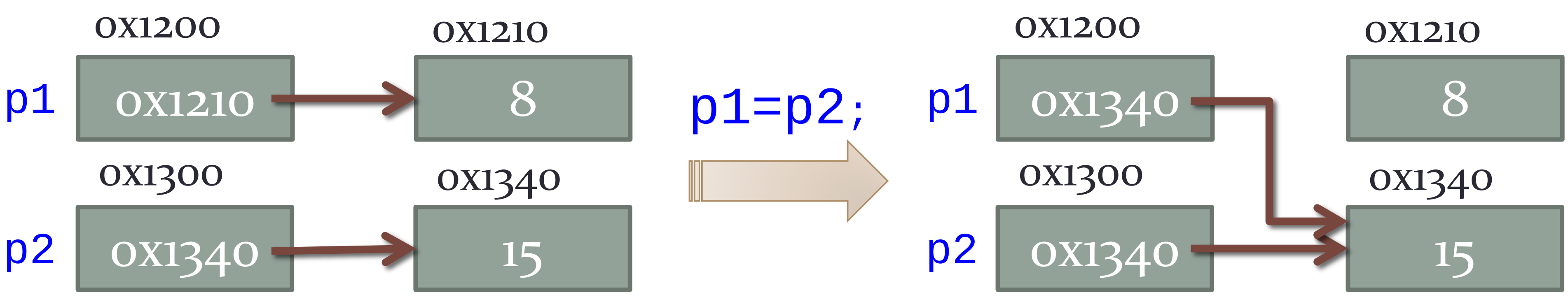
  => p1 is assigned the same address as p2

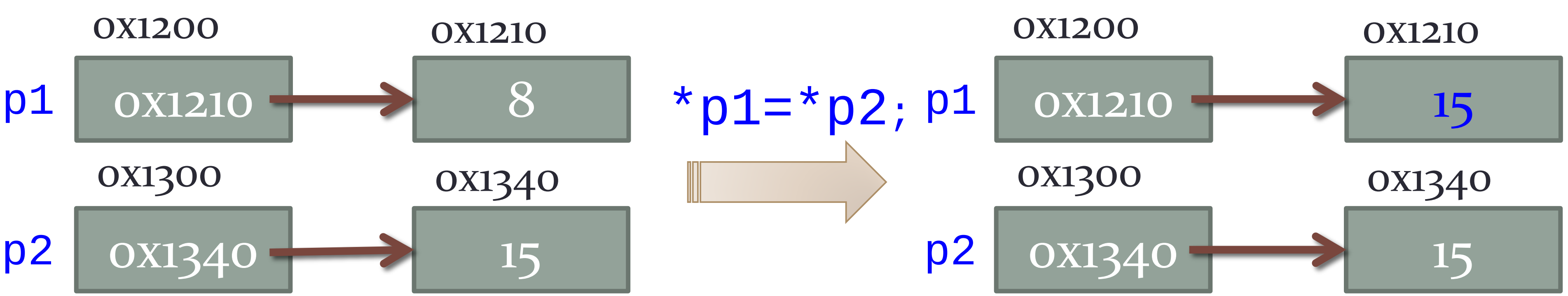
- How about this one?

```
*p1 = *p2;
```

  - assign the value pointed to by `p1` to the value pointed to by `p2`
  - copy the content that `p2` points to the content that `p1` points

# Assignment of Pointers (cont.)

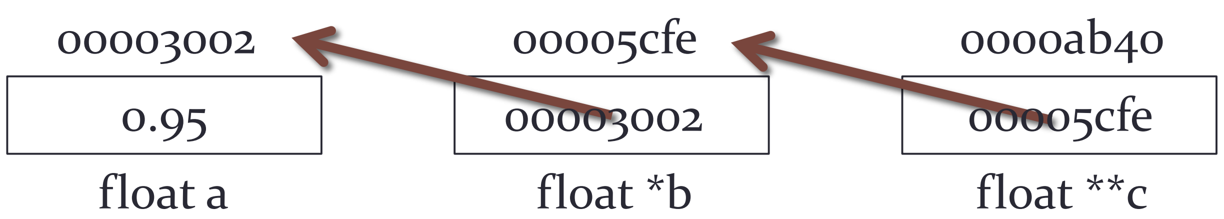


# Pointer to Another Pointer

- When declaring a pointer that points to another pointer, two asterisks (`**`) must precede the pointer name

```
float a = 0.95, *b, **c;
b = &a;  //pointer b points to variable
c = &b;  //pointer c points to pointer
cout << **c; //dereferencing pointer c
             //two times to access var a
```



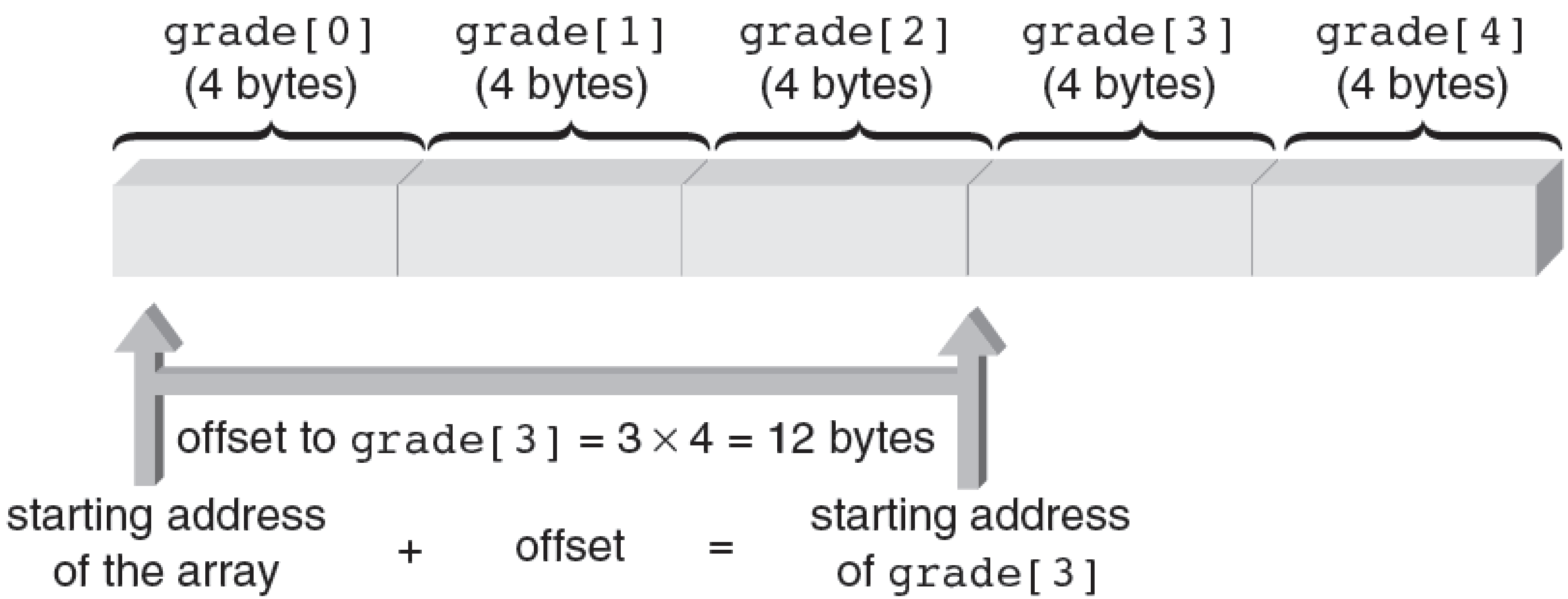
# Access Array by Pointer

- An array name
  - returns the *starting address* of the array
  - the address of the *first element* of the array
  - can also be used as a *pointer to the array* ⇨ faster than indexing

```
int grade[5] = {90, 80, 70, 60, 50};
```

| array indexing | pointer notation |
|---|---|
| `grade[0]` | `*grade` |
| `grade[1]` | `*(grade + 1)` |
| `grade[2]` | `*(grade + 2)` |
| `grade[3]` | `*(grade + 3)` |
| `grade[4]` | `*(grade + 4)` |

# Access Array by Pointer (cont.)



# Pointers & Arrays Example

```
//pointers & Arrays
#include <iostream>
using namespace std;
int main ()
{
    int numbers[5];
    int *p;
    p = numbers; *p = 10;
    p++; *p = 20;
    p = &numbers[2]; *p = 30;
    p = numbers + 3; *p = 40;
    p = numbers; *(p+4) = 50;
    for (int n = 0; n < 5; n++)
        cout << numbers[n] << ", ";
    return 0;
}
```

# Reference Variables

- A reference variable is an alternative name for a variable
  - a reference variable **must be initialized** to reference another variable
  - once the reference is initialized you can treat it just like any other variable
- To declare a reference variable you precede the variable name with a "**&**":

```
int &foo = intA;
double &cost = doubleB;
```

# Examples of Reference Variables

```
…
int count;
// blah is the same variable as count
int &blah = count;
count = 3;
cout << "blah is " << blah << endl;
blah++;
cout << "count is " << count << endl;
…
```

- screen output

```
blah is 3
count is 4
```

# References and Pointers

- Example of automatic dereference:

```
int b;        // b is an integer variable
int &a = b; // a is a reference variable that
              // stores b's address
a = 10;       // this changes b's value to 10
```

  - `int &a = b;` ⇨ compiler assigns address of `b` (not the contents of `b`)
  - `a = 10;` ⇨ compiler uses address stored in `a` to change the value stored in `b` to 10

# References and Pointers (cont.)

- Repeat the example using pointers instead of automatic dereferencing

```
int b;  // b is an integer variable
int *a = &b; // a is a pointer - store
             // b's address in a
*a = 10;     // this changes b's value to 10
```

- a is a pointer initialized to store address of b
  - pointer a can be altered to point to a different variable
  - reference variable a (from previous example) cannot be altered to refer to any variable except one to which it was initialized

# Passing References to Function

- C++ supports the three methods for passing values to functions:
  - pass by value
  - pass by address
  - pass by reference
- Passing references to function
  - code is *cleaner*, it is not necessary to use the * operator
  - *no copy* of function arguments
  - not remember to *pass the address*

# Example of Passing Pointers

```
//swap.cpp
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

```
//main() in main.cpp
int main() {
    int a = 4, b = 10;
    cout << a << " " << b << endl;
    swap(&a, &b);
    cout << a << " " << b << endl;
    return 0;
}
```

# Example of Passing References

```
//swap.cpp
void swap(int &x, int &y) {
    int temp = x;
    x = y;
    y = temp;
}
```

```
//main() in main.cpp
int main() {
    int a = 4, b = 10;
    cout << a << " " << b << endl;
    swap(a, b);
    cout << a << " " << b << endl;
    return 0;
}
```

# Constant Reference Parameters

- Reference arguments inherently "dangerous"
  - Caller's data can be changed
  - Often this is desired, sometimes not
- To "protect" data, and still pass by reference:
  - Use `const` keyword
  - EX:

```
void sendConstRef(const int &par1,
                  const int &par2);
```

    - Make arguments "read-only" by function
    - No changes allowed inside function body

# Reference/Pointers with Constants

- If the `const` keyword is applied to references and pointer, one of the following four types can be created:
  - a reference to a constant
  - a pointer to a constant (Fig. 7.10)
  - a constant pointer (Fig. 7.11)
  - a constant pointer to a constant (Fig. 7.12)

# A Reference to a Constant

- A *read-only* alias
  - cannot be used to change the value it references
- However, a variable that is referenced by this reference can be changed

```
int x = 8;
const int & xref = x; //a ref to a const.
x = 33;
cout << xref ;
//ERROR! cannot modify a ref. to a const.
xref = 15;
x = 50; //OK
```

# A Pointer to a Constant

- A nonconstant pointer to constant data
  - A pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer
- Might be used to receive an array argument to a function that will process each array element, but should not be allowed to modify the data
- Any attempt to modify the data in the function results in a compilation error
- Sample declaration:

```
const int *countPtr;
```

  - Read from right to left as "countPtr is a pointer to an integer constant"

# A Pointer to a Constant (cont.)

```
// Fig 7.10: fig07_10.cpp
// Attempt to modify data through a
// nonconstant pointer to constant data
#include <iostream>
using namespace std;

void f( const int *); // prototype

int main ()
{
    int y;
    f(&y)

    return 0;
}
```

# A Pointer to a Constant (cont.)

```
void f( const int *xPtr)
{
    *xPtr = 100; // error: cannot modify a const object
}
```

- screen output

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

# A Pointer to a Constant (cont.)

```
int x = 4, y = 7;
const int *pt = &x; //a pointer to a const.
cout << *pt;        //print 4 on screen
pt = &y;
cout << *pt;        //print 7 on screen
*pt = 11;           //ERROR! cannot modify
```

- The pointer `pt` to a constant used in this example can store different addresses
  - can pointer to different variable, `x` or `y`
- However, cannot change the dereferenced value that `pt` points to

# A Constant Pointer

- A constant pointer to nonconstant data is a pointer that always points to the same memory location; the data at that location can be modified through the pointer
- An example of such a pointer is an array name, which is a constant pointer to the beginning of the array
- All data in the array can be accessed and changed by using the array name and array subscripting

# A Constant Pointer (cont.)

- A constant pointer is a kind of pointers that its content is constant and cannot be changed
  - cannot be changed to point to another variable
  - but can change the value it points to

```
int var1 = 15, var2 = 8;
//a constant pointer to a declared variable
int * const cpt = &varl;
*cpt = 34;//change the value cpt points to
cout << var1; //print 34 on screen
//ERROR! a const. pointer cannot be changed
cpt = &var2;
```

# A Constant Pointer to a Constant

- The minimum access privilege is granted by a constant pointer to constant data
  - Such a pointer always points to the same memory location, and the data at that location cannot be modified via the pointer
- This is how an array should be passed to a function that only reads the array, using array subscript notation, and does not modify the array
- Ex: `const int * const ptr = &x`
  - This declaration is read from right to left as "`ptr` is a constant pointer to an integer constant."

# A Constant Pointer to a Constant (cont.)

```
// Fig 7.12: fig07_12.cpp
// Attempt to modify a constant pointer to constant data
#include <iostream>
using namespace std;

int main ()
{
    int x = 5, y;
    const int *const ptr = &x;

    cout << *ptr << endl;

    *ptr = 7;  // error: *ptr is const
    ptr = &y;  // error: ptr is const

    return 0;
}
```

# Dynamic Allocation

- Static memory allocation
  - uses the explicit variable and fixed-size array declarations to allocation memory
  - reserves an amount of memory allocated when a program is loaded into the memory
  - a program could fail when lacking enough memory
  - or reserve an excessive amount of memory so that other programs may not run
- What if the size can be known until the program is running?

  ⇨ *dynamic* memory allocation

# Dynamic Memory Allocation

- Only allocate the amount of memory need at run-time
- Heap (a.k.a. freestore)
  - reserved for dynamically-allocated variables
  - all new dynamic variables consume memory is freestore

- C: `malloc()`, `calloc()`, `realloc()`, `free()`
- C++: `new` and `delete`

# The new Operator

- Since pointers can refer to variables…
  - no real need to have a standard identifier
- Can dynamically allocate variables

  => operator `new` creates variables
  - no identifiers to refer to them
  - just a pointer!
- Example: `p1 = new int;`
  - creates a new *nameless* variable, and assigns `p1` to *point to* it
  - can access with `*p1`
  - use just like ordinary variable

# Example of Pointer Manipulations

```
#include <iostream>
using namespace std;
int main()
{
    int *p1, *p2;
    p1 = new int;
    *p1 = 45;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;
    *p2 = 23;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;
    p1 = new int(101); //initialize as well
    *p2 = 77;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;
    return 0;
}
```

# Checking new Success

- Older compilers:

```
int *p;
p = new int;
if (p == NULL)
{
    cout << "Insufficient memory.\n";
    exit(1);
}
```

  - test if null returned by call to new:
  - if new succeeds, program continues

# Checking new Success (cont.)

- For newer compilers, if new operation fails:
  - Program terminates automatically
  - Produces error message
- Still good practice to use NULL check

# The `delete` Operator

- De-allocate dynamic memory
  - when no longer needed
  - return memory to freestore
- Example:

```
int *p;
p = new int(5);
… //some processing…
delete p; //delete space that p points to
```

  - *de-allocate* dynamic memory pointed to by pointer `p`
  - literally *destroy* memory space

# Dangling Pointers

- `delete p;`
  - destroy dynamic memory
  - but `p` still points the original address => called *dangling pointer*
  - if `p` is then *dereferenced* (`*p`)

    => unpredictable results! often disastrous!
- Avoid dangling pointers
  - assign pointer to `NULL` after delete:

```
delete p;
p = NULL;
```

# Standard vs. Dynamic Arrays

- Standard array limitations
  - must specify size first ⇨ estimate maximum
  - may not know until program runs
  - waste memory
- Example:

```
const int MAX_SIZE = 100000;
int iArray[MAX_SIZE];
```

  - what if the user only need 100 integers?
- Dynamic arrays
  - can grow and shrink as needed

# Creating Dynamic Arrays

- Use `new` operator
  - dynamically allocate with pointer variable
  - treat like standard arrays
- Example:

```
int size = 0;
cin >> size;
double *ptr;
ptr = new double[size]; //size in brackets
```

  - create a dynamical array variable `ptr`
  - contain `size` elements of type **`double`**

# Deleting Dynamic Arrays

- Allocated dynamically at run-time
  - so should be destroyed at run-time
- Continue the previous example:

```
ptr = new double[size]; //size in brackets
… //Processing
delete [] ptr; //delete array that p points
```

  - de-allocate all memory for dynamic array
  - brackets [] indicate array is there
  - note that `ptr` still points there. => dangling!
  - should add "`ptr = NULL;`" immediately

# Dynamic Multi-dimensional Arrays

- Multi-dimensional arrays are *arrays of arrays*
  - various ways to create dynamic multidimensional arrays.
- Example:

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

  - declare one array `m` of 3 IntArrayPtr pointers
  - make each allocated array of 4 integers
  - create one 3x4 dynamic array

# Two-dimensional Dynamic Arrays

- Example 1:

```
int *Mat1[4]; //fix 2nd dimension at 4
for (int r = 0; r < 4; r++)
    Mat1[r] = new int[6]; //create 6 columns
```

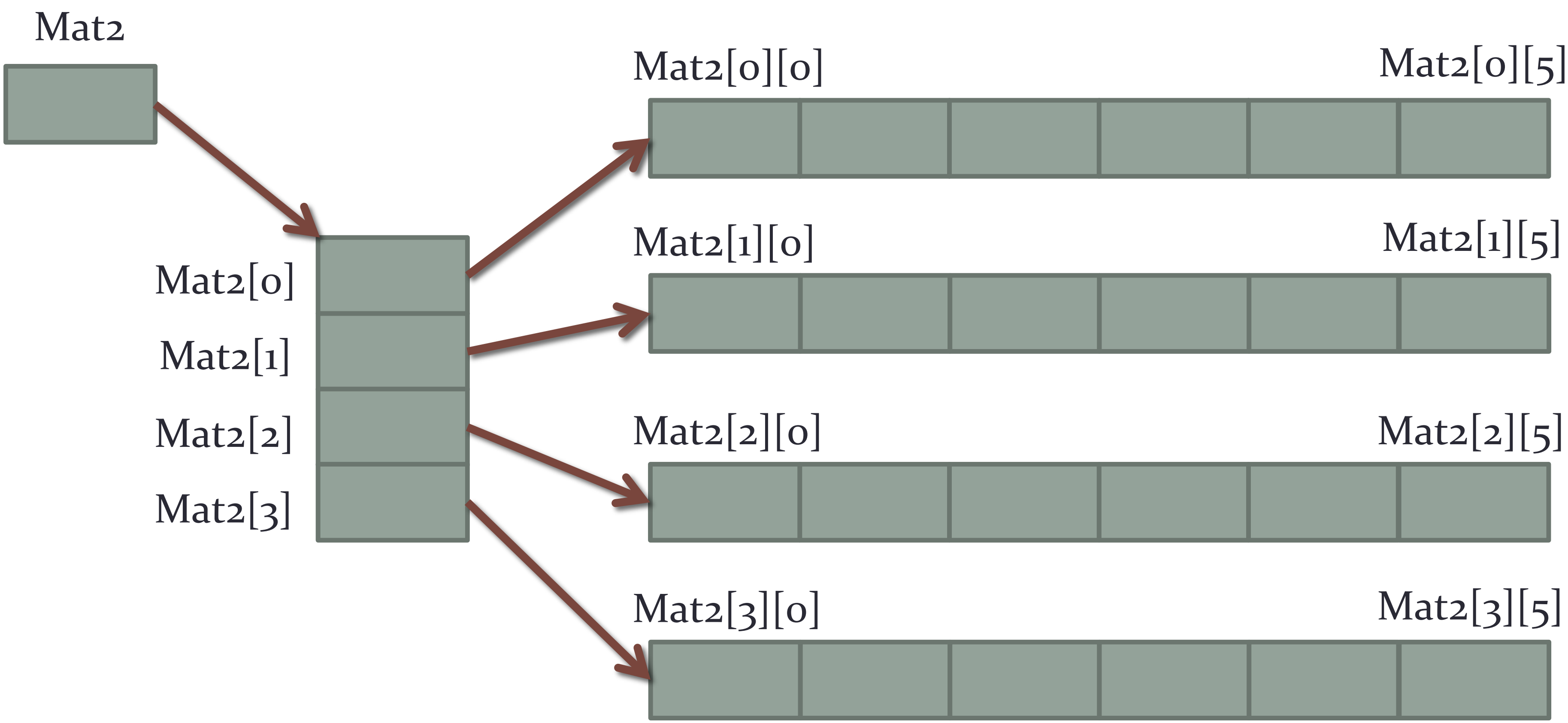
  - 4 row `Mat1[0]`, `Mat1[1]`, `Mat1[2]` and `Mat1[3]` are declared
  - each row has 6 columns to be created

- Example 2: (most common)

```
int **Mat2; //2-level pointer
Mat2 = new int *[4]; //create 4 rows
for (int r = 0; r < 4; r++)
    Mat2[r] = new int[6]; //create 6 columns
```

  - Both `Mat2` and `*Mat2` are pointers

# Two-dimensional Dynamic Arrays (cont.)



# Delete Dynamic Arrays

- After a dynamic array is of no use any more, de-allocate the memory by `delete` operation
  - Clean reversely from last allocated memory
- Example: //re-allocate a dynamic 5x9 matrix

```
int** Mat = new int *[5]; //create 5 rows
for (int r = 0; r < 5; r++)
    Mat[r] = new int[9]; //create 9 columns
... //some processing
for (int r = 0; r < 5; r++) //clean columns
    delete [] Mat[r];
delete [] Mat; //clean rows
Mat = NULL;
```

# Expand Dynamic Arrays

- A program can start with a small array and then expands it only if necessary
- Example: //initially `MAX` is set as `10`

```
int * ivec = new int [MAX];
while (cin >> ivec[n]) {
    n++;
    if (n >= MAX) {
        MAX *= 2;
        int * tmp = new int [MAX];
        for (int j = 0; j < n; j++)
            tmp[j] = ivec[j];
        delete [] ivec;
        ivec = tmp;
    }
}
```

# Shallow vs. Deep Copies

- Shallow copy (copy-by-address)
  - two or more pointers point to the same memory address
- Deep copy (copy-by-value)
  - two or more pointers have their own data

```
int *first, *second;
first = new int[10];
second = first; //shallow copy
second = new int[10];
//deep copy
for (int idx = 0;idx < 10; idx++)
    second[idx] = first[idx];
```

# Common Programming Errors

- Using a pointer to access nonexistent array elements
- Incorrectly apply address and indirect operators

```
int *ptr1 = &45;
int *prt2 = &(miles+10);
```

  - Illegal to take the address of a value
- Taking addresses of pointer constants

```
int nums[25];
int * pt;
pt = &nums;
```

  - Correct form: `pt = nums;`

# Common Programming Errors (cont.)

- Initializing pointer variables incorrectly

```
int *pt = 5;
```

  - `pt` is a pointer to an integer
  - must be a valid address of another integer variable or NULL
- Forgetting to the bracket set, `[]`, after the `delete` operator when dynamically de-allocating memory

# Pass Arrays to Function

- When array is passed to a function, only pass the *address* of the first element
- Example: in main function

```
int max = FindMax(array, size);
```

in function declaration section

```
int FindMax(int *array, int size) {
    ...
}
```

- parameter receives the address of array
- Another form:

```
int FindMax(int val[], int size) {}
```

# Return Array from Function

- Array type pointers are not allowed as return-type of function
- Example:

```
int [] someFun(...); //illegal
```

- Instead return pointer to array base type

```
int * someFun(...); //legal
```

- Return a integer pointer after function call
  - in main (or caller) function

```
int * pt = someFun(...);
```

  - only one array (address) can be returned!

# Return Array from Function (cont.)

- One more example:

```
int *display();

int main() {
    cout << *display() << endl;
}

int *display() {
    int *pt = new int(0);
    int b[2] = {10,20};
    for (int i = 0; i < 2; i++)
        *pt += b[i];
    return pt;
}
```

# Summary

- Fundamentals of C++ pointer
  - operators and expressions for pointers
  - point to another pointer/array

- References
  - as a reference variables
  - pass to functions including constant references

# Summary (cont.)

- Using references and pointer with constants
  - a reference to a constant
  - a pointer to a constant
  - a constant pointer
  - a constant pointer to a constant

- Dynamic memory allocation
  - C/C++ memory allocation
  - `new/delete` operators
  - memory leaking/dangling pointers
  - multi-dimensional dynamic arrays
  - pass/return array to/from function

# References


- Paul Deitel and Harvey Deitel, “C How to Program” Sixth Edition
  - Chapter 7
  - Chapter 15.7: Reference Variable
- Paul Deitel and Harvey Deitel, “C++ How to Program (late objects version)” Seventh Edition
  - Chapter 5.15: Reference Variable
  - Chapter 6: Array
  - Chapter 7: Pointer
- W. Savitch, “Absolute C++,” Fourth Edition
  - Chapter 10