

Pointers, References, and Dynamic Memory Allocation

Jiun-Long Huang
National Chiao Tung University

Introduction

- C pointers are very powerful and dangerous.
 - If they are not used properly they are prone to producing unpredictable results, including system crashes.
- C++ provides a new kind of pointer called a **reference**.
 - References have advantages over regular pointers when passed to functions.
 - Call by value vs. call by reference

Pointer

- A pointer is a variable that is used to store a **memory address**.
 - This address can be a location of variable, pointer, or function.
- The major benefits of using pointers are
 - To support dynamic memory allocation
 - To provide the means by which functions can modify their actual arguments
 - To support some types of data structures such as linked lists and binary trees
 - To improve the efficiency of some programs

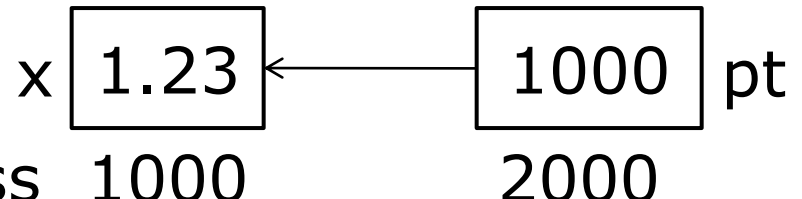
Pointer (contd.)

- ❑ A pointer variable is declared using
 - `data_type *variable_name;`
- ❑ Data type the pointer points can be any valid C/C++ type including void and user-defined types
 - `int *ptr1;`
 - `double *ptr2;`
 - `void *ptr3;` //can point to a variable of any type
 - `Robot *ptr4;` //user defined

Operators

- ❑ Indirection operator (*) precedes a pointer and returns the value of a variable, the address of which is stored in the pointer.
- ❑ Address-of operator (&) returns the memory address of its operand.

```
float x=1.23;
float *pt;
pt=&x;
/* places the memory address of x into pt */
cout << pt << ' ' << *pt;
//prints the address of x: ???
//prints the value of x: 1.23
```



The diagram illustrates the memory state after the code execution. It shows two memory locations: one for variable `x` at address 1000 containing the value 1.23, and another for pointer variable `pt` at address 2000 containing the value 1000. An arrow points from the box containing 1000 (under `pt`) to the box containing 1.23 (under `x`), indicating that `pt` stores the address of `x`.

Operators (contd.)

□ The * operator performs on a pointer (accessing the value the pointer points to) is also described as dereferencing the pointer.

□ A void pointer **cannot** be dereferenced.

```
void *pt1; //pt1 is a void pointer
int *pt2;  //pt2 is an integer pointer
int x=3,y,z;
pt1=pt2=&x; //pt1 and pt2 point to x
y=*pt1;     //ERROR! pt1 cannot be dereferenced.
z=*pt2;     //CORRECT! pt2 is dereferenced and
            //the value of x is placed into z.
```

Examples

```
float f[]={1.1,2.2,3.3,1.1};
```

```
float *ptr1,*ptr2;
```

```
ptr1=&f[1];
```

1.1	2.2	3.3	1.1
-----	-----	-----	-----

```
ptr2=ptr1; //assigning pointers: ptr1 and  
           //ptr2 point to f
```

```
ptr1--; //decrementing ptr1
```

```
cout << ptr1; //The value of ptr1 is 996 (=1000-4)
```

```
ptr2=ptr2+2; //adding 5 to ptr2 and assigning  
            //result to ptr2
```

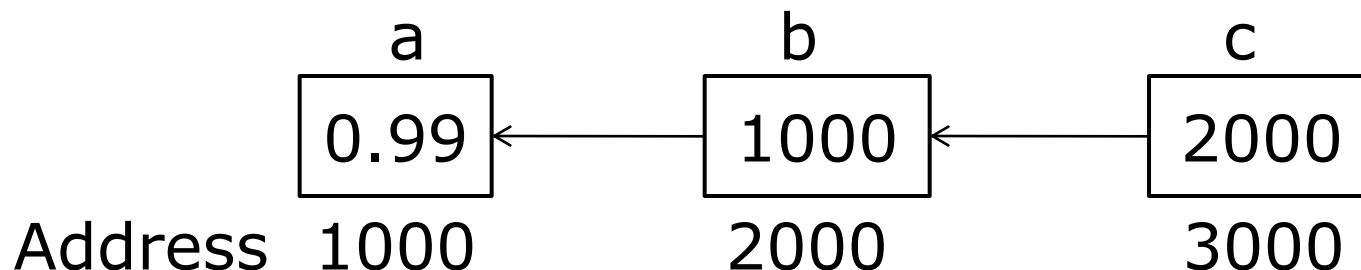
```
cout << ptr2; //The value of ptr2 is 1008 (=1000+2*4)
```

```
if (ptr1==ptr2) //comparing pointers by equality
```

```
    cout << "Both pointers contain the same memory  
           address.";
```

Pointing to Another Pointer

- ❑ `float a=0.99, *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 a`



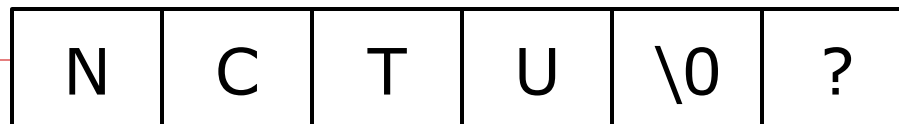
Access Array by Pointer

- An array name returns the starting address of the array (the address of the first element of the array), an array name can also be used as a pointer to the array.

Array indexing	Pointer notation
array[0]	*array
array[1]	*(array+1)
array[2]	*(array+2)
array[3]	*(array+3)

String

- ❑ String is equivalent to a character pointer.
- ❑ Operations with strings are often performed by using pointers.
 - `char school[6]="NCTU";`
 - `char *cptr;`
 - `cptr = school;` //cptr is set to the address of school
 - `cout << *(cptr+2);` //prints the third character 'T'
 - `cout << cptr;` //prints the entire string "NCTU"



A **void pointer** can point to a variable of any type.

```
int ivalue=13;
float fvalue=8.3;
int *iptr;
float *fptr;
void *vptr;
fptr=&ivalue;      //ERROR! A float pointer cannot
                   //point to an integer variable.

iptr=&ivalue;       //CORRECT
vptr=&ivalue;       //CORRECT
vptr=&fvalue;       //CORRECT
iptr=fptr;         //ERROR! A float pointer cannot
                   //be assigned to an integer pointer.

vptr=fptr;         //CORRECT
vrptr=iptr;        //CORRECT
```

Differences in C and C++ (contd.)

- ❑ Both C and C++ do not permit the direct assigning pointers of different types.
- ❑ However, C can accomplish that indirectly, through a void pointer.

```
mytype1 *ptr1;  
mytype2 *ptr2;  
void *vptr;  
//Assume all pointers have been initialized at  
//this point.  
vptr = ptr1;  
ptr2 = vptr; // No problem in C; Invalid in C++.
```

Differences in C and C++ (contd.)

- Pointers of different types can be assigned to each other by using a type cast as follows:

```
ptr2 = (mytype2 *) ptr1;
```

References

- ❑ A reference is an implicit pointer that is automatically dereferenced.
- ❑ References also act as alternative names for other variables.
- ❑ A reference can be used in three different ways:
 - Created as an independent variable
 - Passed to a function
 - Returned by a function

References as Independent Variables

- ❑ To create a reference variable, the & operator has to be put before the variable name when it is declared.
- ❑ The type of reference variable should be the same as the type of the variable to which it points.

References as Independent Variables (contd.)

- ❑ The values of reference variables **have to be initialized when declared.**

```
data_type & reference-name = variable-name;
```

```
float num = 7.3;
```

```
float & refnum = num;
```

```
//The refnum reference is initialized to num
```

```
//and it acts as an alias for num
```


PROGRAM CODE

```
int x = 13;
int & ref = x;
```

```
int *ptr;
```

```
ref = 8;
```

```
ptr = &x;
```

MEMORY CONTENT

address: 1040₁₀

13

x/ref

address: 1100₁₀

?

ptr

address: 1040₁₀

8

x/ref

address: 1100₁₀

1040

ptr

OUTPUT

```
cout<<x<<endl;
```

8

```
cout<<*ptr<<endl;
```

8

```
cout<<ref<<endl;
```

8

```
cout<<ptr<<endl;
```

1040

```
cout<<&ref<<endl;
```

1040

```
cout<<&ptr;
```

1100

Differences Between Pointers and References

RESTRICTIONS	Reference	Pointer
It reserves a space in the memory to store an address that it points to or references.	NO	YES
It has to be initialized when it is declared.	YES	NO
It can be initialized at any time.	NO	YES
Once it is initialized, it can be changed to point to another variable of the same type.	NO	YES
It has to be dereferenced to get to a value it points to.	NO	YES
It can be a NULL pointer/reference.	NO	YES
It can point to another reference/pointer.	NO	YES
An array of references/pointers can be created.	NO	YES

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

```
int main()
{
```

```
    int i = 13;
    int &iref = i; //declaring a reference variable
    cout<<"The value is => "<<iref;
    i--;
    cout<<"\nAfter decrementing => "<<iref<<endl;
    iref = 99;
    cout<<"The value is now => "<<i;
    return 0;
}
```

The value is => 13
After decrementing => 12
The value is now => 99

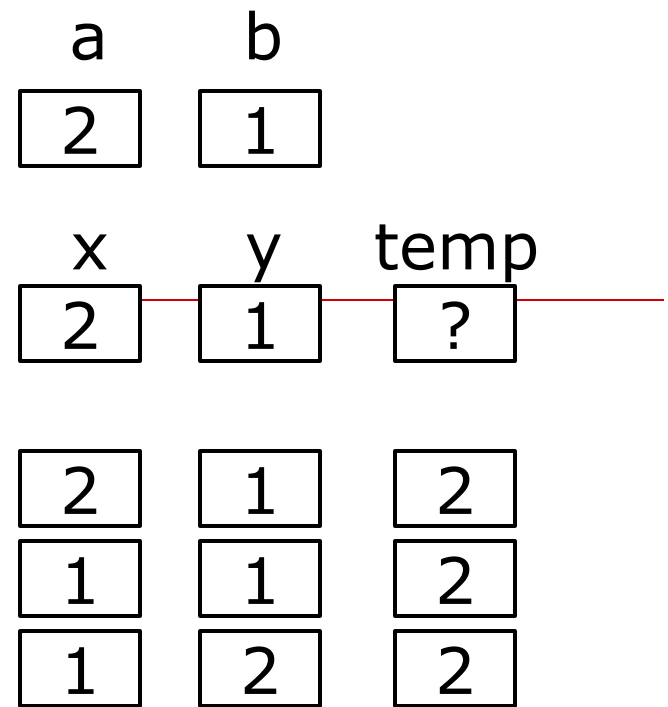
Passing Reference to Function

- C++ supports the following three methods for passing values to functions:
 - by value
 - by reference.
- When passed to functions, references have a clear advantage over pointers.
 - The amount of memory movement can be reduced

```

#include <iostream>
using namespace std;
void swap1(int x, int y)
{
    int temp;
    temp=x;
    x=y;
    y=temp;
}

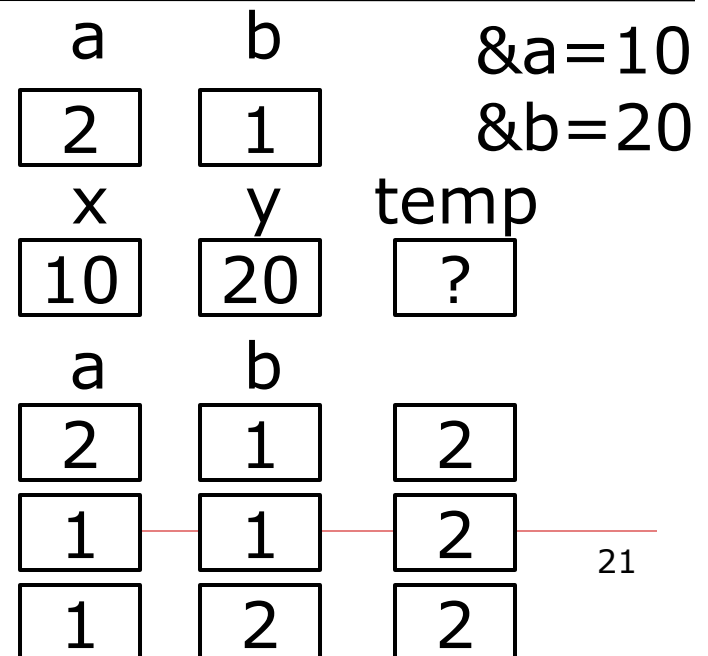
```



```

void swap2(int * x, int * y)
{
    int temp;
    temp=*x;
    *x=*y;
    *y=temp;
}

```

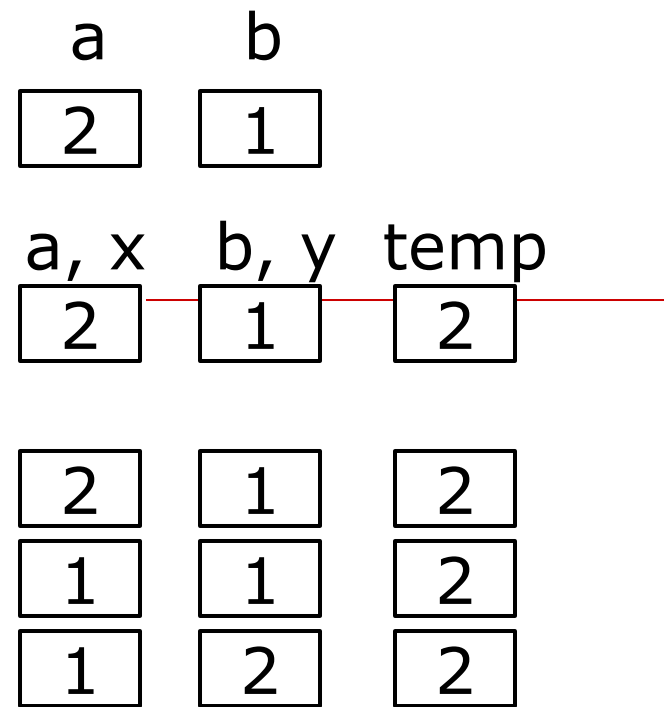


&a=10
&b=20

```

void swap3(int & x, int & y)
{
    int temp;
    temp=x;
    x=y;
    y=temp;
}
int main()
{
    int a=2, b=1;
    cout << a << ' ' << b << endl;
    swap1(a,b);          // output: 2 1
    //swap2(&a, &b);      // output: 1 2
    //swap3(a,b);        // output: 1 2
    cout << a << ' ' << b << endl;
    return 0;
}

```



Returning Reference by Function

- ❑ A function may return a reference.
- ❑ This is particularly important when overloading some types of operators-e.g., inserter and extractor
- ❑ Returning a reference by a function also permits the function to be called from the **left side of the assignment** operator.

```
#include <iostream>
#include <cstdlib>
■ using namespace std;
const int SIZE = 6;
int & put_val(int a[], int n)
//function returns a reference
{
    if(n>=SIZE || n<0)
    {
        cout<<"Outside of boundaries!";
        exit(1);
    }
    return a[n];
}
```

```
int main()
{
    int array[SIZE];
    for(int i=0; i<SIZE; i++)
        put_val(array, i)=i*2;
        //function call is on the left side
    for(int j=0; j<SIZE; j++)
        cout<<"array["<<j<<"] = "<<array[j]<<endl;
    return 0;
}
```

-
- This approach is less error prone than a direct assignment.
 - Array boundary checking
 - This prevents run-time errors such as array overflows or underflows caused by assigning a value to an array element specified by the index that is outside of the boundaries of the array.

Using References and Pointers with Constants

- If the `const` keyword is applied to references and pointers, one of the following four types can be created
 - A reference to a constant
 - A pointer to a constant
 - A constant pointer

A Reference to a Constant

- By preceding a reference type with the `const` keyword, a reference to a constant is created.
- It is a read-only alias, which 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 reference to a constant
x=33;
cout << xref ; //Displays 33
xref=15; //ERROR! Cannot modify a
        // reference to a constant.
x=50; //OK
```

A Constant Pointer

- ❑ A constant pointer can be used to change the value it points to
- ❑ It cannot be changed to point to another variable-i.e., the **address** stored in a constant pointer cannot be changed.

```
int var1=15, var2=8;
//A constant pointer to an integer
int * const cpt=&var1;
*cpt=34;
cout <<var1; //Displays 34
cpt=&var2;    // ERROR
```

A Pointer to a Constant

- The pt pointer to a constant used in this example can store different addresses-it is, it can point to different variables.
- It cannot be used to change a value, which is stored at the address to which it points.

```
const double *pt; //A pointer to a constant
double x=3.3, y=4.4;
pt=&x;
cout << *pt;      //Displays 3.3
pt=&y;
cout << *pt;      //Displays 4.4
*pt=5.05; //ERROR! Cannot modify a
           //pointer to a constant.
```

Static Memory Allocation

- Static memory allocation is a technique that uses the explicit variable and fixed-size array declarations to allocate memory.
 - An amount of memory allocated using this technique is reserved when a program is loaded into the memory.

Static Memory Allocation (contd.)

- ❑ A program could fail when running on some computer systems that lack enough memory
- ❑ A program could reserve an excessive amount of memory and make it difficult to run any other programs at the same time.

```
struct comp_part {  
    char code[7];  
    char description[30];  
    int on_stock;  
    int sold;  
    float price;  
};  
comp_part[100];
```


Dynamic Memory Allocation

- Run-time allocate

 - Heap: a region of memory

- C: malloc(), calloc(), realloc(), free()

- C++: new and delete

 - new will return NULL when error occurs in memory allocation

```
pointer_var = new data_type (initial_value);  
delete pointer_var;
```

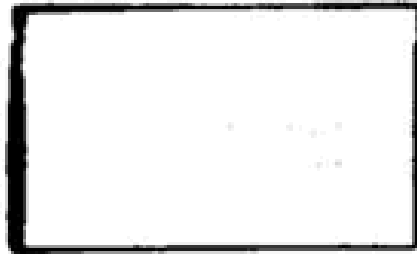
```
float *fpt = new float(0.0);  
//Allocates a float variable and initializes  
//to 0.0  
if( fpt == NULL )  
//Checks for a memory allocation error  
//Can be changed to if(!pt)  
{  
    cout << "Memory allocation error."  
    exit(1);  
}  
*fpt=3.45; //Uses pointer to access memory  
cout << *fpt;  
delete fpt;
```

Memory Leak

- ❑ A common type of error and failing to prevent it could result in a memory resource problem.
- ❑ If the pointer is set to point to another block of memory without the delete statement that frees the previous block, it causes a memory leak.

```
float *ptr = new float; //Allocates first block
*ptr = 7.9;           //Accesses first block
ptr = new float; //Allocates second block
*ptr = 5.1;           //Accesses second block
```

ptr

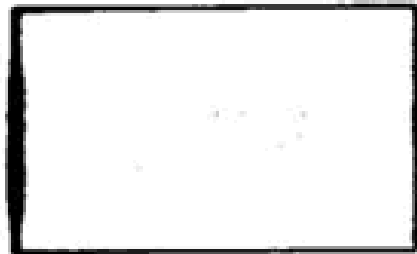


first block

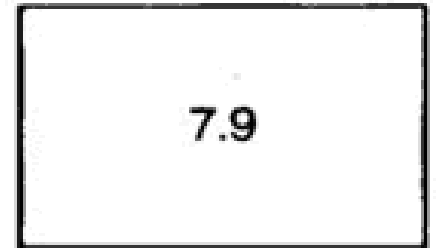


7.9

ptr

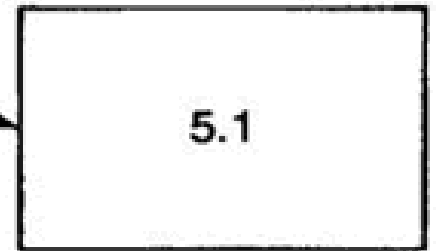


first block is lost



7.9

second block



5.1

Dynamic Arrays

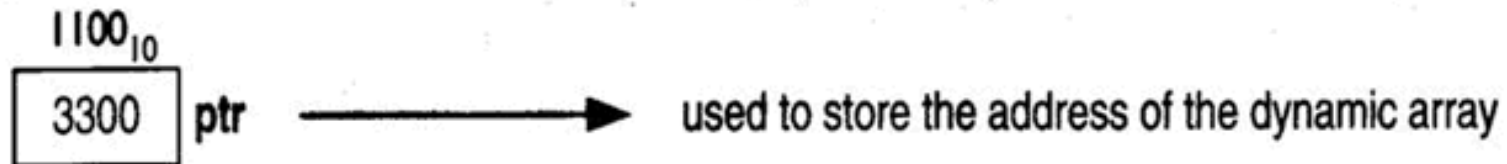
- ❑ May have a variable size, run time allocate
- ❑ Using new and delete

```
pointer_var = new data_type[size];  
delete []pointer_var;
```

- ❑ Size: integer, variable or expression

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int *ptr, n=10;
    ptr = new int[n]; //Allocates an array dynamically
    if(!ptr) //Checks for a memory allocation error
    {
        cout<<"Memory allocation error!";
        exit(1);
    }
    for(int i=0; i<n; i++)
    {
        ptr[i]=(i+1)*2; //Initializes the array
        cout<<setw(4)<<ptr[i]; //Displays its values
    }
    delete []ptr; //Deletes the array from the heap
    return 0;
}
```

MEMORY ALLOCATION



3300_{10}	3304_{10}	3308_{10}	3312_{10}	3316_{10}	3320_{10}	3324_{10}	3328_{10}	3332_{10}	3336_{10}
2	4	6	8	10	12	14	16	18	20
ptr[0]	ptr[1]	ptr[2]	ptr[3]	ptr[4]	ptr[5]	ptr[6]	ptr[7]	ptr[8]	ptr[9]

OUTPUT

2 4 6 8 10 12 14 16 18 20

Multi-Dimension Array

```
#include <iostream>
#include <iomanip>
using namespace std;
void memError() {
    cout<<"Memory allocation error!";
    exit(1);
}
int main() {
    int rows, columns, i, j;
    int **p2d; //Declares a pointer to pointer
    cout<<"Enter a number of rows => ";
    cin>>rows;
    cout<<"\nEnter a number of columns => ";
    cin>>columns;
    p2d=new int*[rows];
```

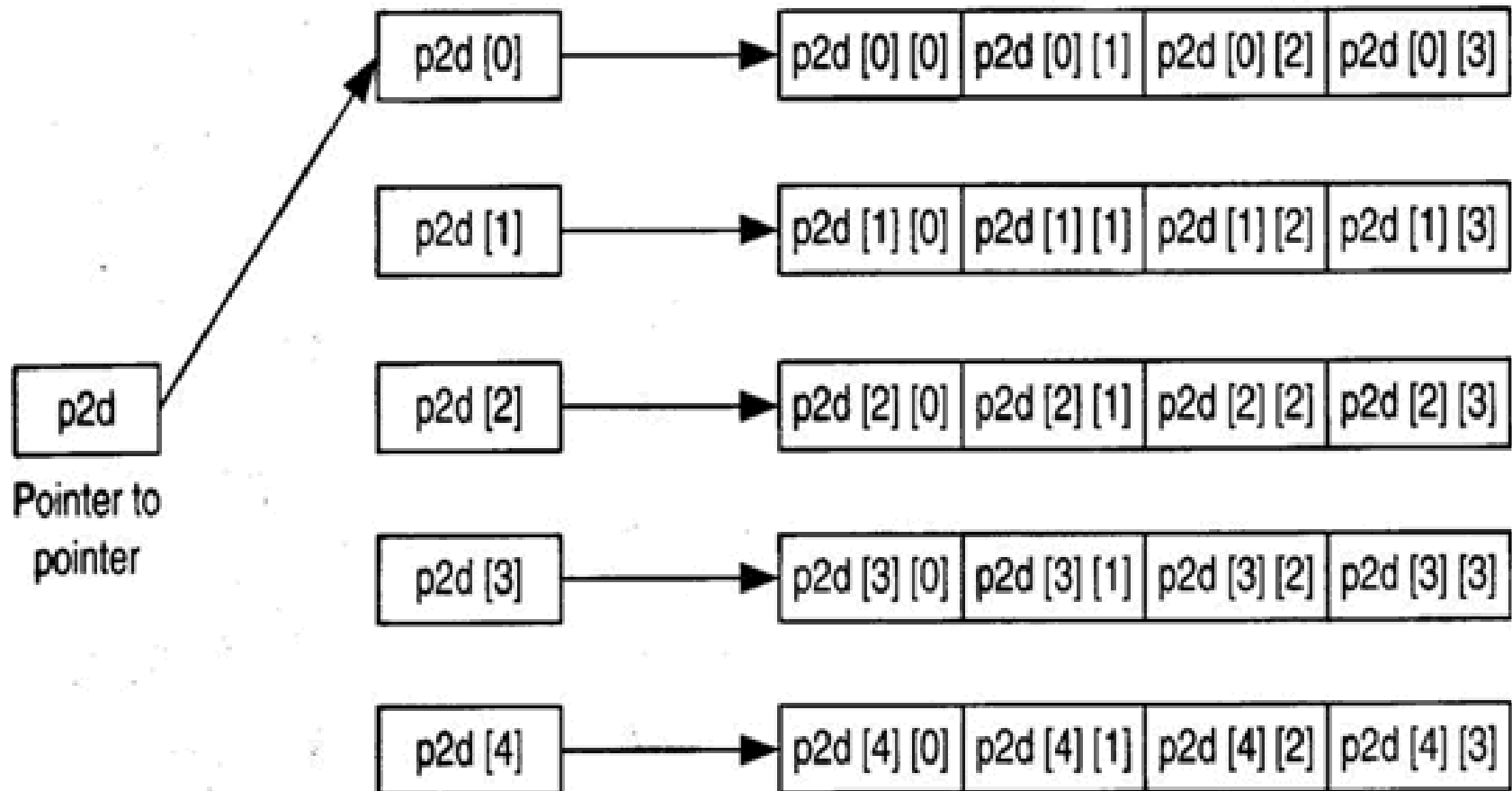


```

if(!p2d) //Checks for an allocation error
    memError( );
for(i=0; i<rows; i++) {
    p2d[i] = new int [columns]; //Sets up the columns
    if(!p2d[i])                //Checks for an allocation error
        memError();
}
cout<<"\n***MULTIPLICATION TABLE***"<<endl;
for(i=0; i<rows; i++) {
    for(j=0; j<columns; j++) {
        p2d[i][j]=(i+1)*(j+1);    //Initializes the array
        cout<<setw(5)<<p2d[i][j]; //Displays its values
    }
    cout<<endl;
}
for(i=0; i<rows; i++)
    delete [] p2d[i];              //Deletes the columns
delete [] p2d;                    //Deletes the rows
return 0;
}

```

Two-dimensional dynamic array: $p2d[i][j]$



$p2d[i]$ Array of pointers, each points to a row of the 2-D array

Enter a number of rows => 3

Enter a number of columns => 3

MULTIPLICATION TABLE

1	2	3
2	4	6
3	6	9