Pointers and Arrays

CS 201

This slide set covers pointers and arrays in C++. You should read Chapter 8 from your Deitel & Deitel book.

Pointers

- Powerful but difficult to master

- Used to simulate pass-by-reference (without using a reference parameter)
  Especially important in C, since it does not support the pass-by-reference mechanism

- Used to create and manipulate dynamic data structures (objects and arrays)
  - `Book* B1 = new Book;` `// dynamically allocating an object`
  - `Book* B2 = new Book [5];` `// dynamically allocating an array`
  - `Book B3[10];` `// declaring an automatically allocated array, whose name is indeed a constant pointer (B3 is not a dynamically allocated array)`

  You will learn these declarations, allocations, necessary deallocations, and more

Pointers

- Pointers are variables that contain memory addresses as their values
- There are two ways of accessing a memory location
  - **Direct reference**: Through the name of a variable associated with this memory location, which requires declaring each variable separately and being in its scope
  - **Indirect reference**: Through dereferencing if you know the address of this memory location
    - You do not need to declare each variable separately
      - For example, you may declare an array corresponding to consecutive memory locations and you can access all these memory locations using the array name and the subscript operator (which indeed uses dereferencing)
    - You can access an out-of-the-scope variable through dereferencing

This makes pointers very powerful (as you can keep address values only in pointer variables)

Pointers

- To declare a pointer, use * before the variable name in a declaration line

```
int* p1;                    // p1 is a pointer (or points) to an integer
Book *B1, *B2;              // B1 and B2 are pointers (or point) to
                            // Book objects
double *d1, d2, *d3;        // d1 and d3 are pointers (or point) to
                            // doubles (but d2 is just a double variable)
```

- Operators closely related with pointers
  - Address (or address-of) operator
  - Dereferencing (or indirection) operator
- Other operators also defined for pointers
  - Assignment operator
  - Equality operators
  - Relational operators
  - Arithmetic (addition and subtraction) operators

Relational and arithmetic operators are meaningless if their operands do not point to the members of the same array

Address operator (&)

- Returns the memory address of its operand
- Can be applied to any memory location (to a variable of any data type)
- Its return value can be used only as an *rvalue*

```
int y;
int* yptr;      // yptr is a pointer to an integer (its value is garbage now)
yptr = &y;      // & takes the address of y and this address is assigned to yptr

// return value of the address operator cannot be used as an lvalue
// &y = yptr;  COMPILE-TIME ERROR
```

Dereferencing operator (*)

- Returns the memory location whose address is kept in its operand
- Can only be applied to an address value (to a pointer variable)
- Its return value can be used as an *rvalue* or an *lvalue*
- The compiler should know the data type of the memory location pointed by its operand
  - Since it considers the operand's value as the starting address but cannot know how many bytes to access without knowing this data type
  - Thus, this operator cannot be applied to `void*` (void pointers cannot be dereferenced)

***VERY IMPORTANT**: Dereferencing a pointer that does not keep the address of a specific location in memory may cause a fatal run-time error. It may accidentally modify important data and cause the program to run to completion, possibly with incorrect results*

```
int y, *yptr;
*yptr = 10;     // THIS MAY CAUSE A PROGRAM CRASH -- RUN-TIME ERROR
yptr = &y;
y = 5;          // direct access to y
*yptr = 10;     // indirect access to y (by dereferencing yptr), *yptr is an lvalue
cout << *yptr; // indirect access to y (by dereferencing yptr), *yptr is an rvalue
```

Pointer operators

- **Assignment operator (=)** assigns the address values to pointers of the same type

- Its right-hand-side can be
  - A pointer of the same type
  - 0 or NULL (indicating that it is an invalid memory address)
    - *NULL pointers should not be dereferenced*
  - A value obtained by the address operator (but cannot be an arbitrary value)

```
int a, *b, **c;
double x = 3.4, *y = &x; // y points to x

b = &a;    // b points to a
c = &b;    // c points to b
a = 5;     // direct access to a
**c = 20; // indirect access to a
           // through double dereferencing

// Compile-time errors: RHS and LHS
// should be of the same type
// b = y;       b is int* and y is double*
// y = b;       y is double* and b is int*
// y = &a;      y is double* and &a is int*
// c = &a;      c is int** and &a is int*
// *c = *b;     *c is int* and *b is int

// Compile-time error: Arbitrary values
// cannot be assigned to pointers
// b = 100000;

// The following is a valid assignment
// since *b is int and *y is double
*b = *y;
```

Do not confuse declarations and operators

```
// all * given in red are pointer declarations (variables, parameters, or return types)
// all * given in blue are dereferencing operators
// all & given in green are reference declarations (variables, parameters, or return types)
// all & given in pink are address operators

int* bar(int* x, int& a, int** v) {
   // ...
}
int& foo( ) {
   // ...
}
int main() {
    int a, *b = &a, **c = &b, *d = *c, *e = b, f = **c, &g = a;
    *b = 3;
    **c = a;
    b = *c;
    c = &d;
    b = bar(&a, f, &d);
    e = bar(*c, *b, c);
    a = foo( );
}
```

What is the output of the following program?

```
int main() {
    int x = 4, y = 5, *a = &x, *b = a, **c = &b;
    cout << "Address of x: " << &x << endl;
    cout << "Address of y: " << &y << endl;
    cout << "Address of a: " << &a << endl;
    cout << "Address of b: " << &b << endl;
    cout << "Address of c: " << &c << endl;

    cout << "**c: " << **c << endl;
    cout << "*c : " << *c << endl;
    cout << "c  : " << c << endl;
    cout << "&c : " << &c << endl << endl;

    b = &y;
    *b = 3;
    cout << "**c: " << **c << endl;
    cout << "*c : " << *c << endl;
    cout << "c  : " << c << endl;
    cout << "&c : " << &c << endl << endl;

    cout << "*&c: " << *&c << endl;
    cout << "&*c: " << &*c << endl;
    cout << "*&x: " << *&x << endl;
    // cout << "&*x: " << &*x;         compile-time error
    return 0;
}
```

Output:

```
Address of x: 0x7fff57a1c9c8
Address of y: 0x7fff57a1c9c4
Address of a: 0x7fff57a1c9b8
Address of b: 0x7fff57a1c9b0
Address of c: 0x7fff57a1c9a8
**c: 4
*c : 0x7fff57a1c9c8
c  : 0x7fff57a1c9b0
&c : 0x7fff57a1c9a8

**c: 3
*c : 0x7fff57a1c9c4
c  : 0x7fff57a1c9b0
&c : 0x7fff57a1c9a8

*&c: 0x7fff57a1c9b0
&*c: 0x7fff57a1c9b0
*&x: 4
```

Getting effects of pass-by-reference using pointers

- Using a pointer parameter and the dereferencing operator, one can get the effects of pass-by-reference without using a reference parameter
  - Caller function passes the address of a variable that it wants to change
  - Called function takes this address in a pointer parameter
  - Called function changes the variable's value by indirectly accessing it through the dereferencing operator

BE CAREFUL: Here the aim is NOT to change the value of the parameter y (if it was, it would not work). But, the aim is to change the value of the location where y points to.

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

void nextIntByPointer( int* );

int main() {
    int x = 30;
    nextIntByPointer( &x );
    cout << x << endl;
    return 0;
}
void nextIntByPointer( int* y ) {
    (*y)++;
}
```

`nextIntByPointer` *uses pass-by-value*
(a copy of the x's address is passed to the function)

*Implement and call them **with** and **without** using reference parameters*

```
// swaps two integers
void swapIntegers( ? i1, ? i2 ) {

}
// swaps two integer pointers
void swapIntegerPointers( ? p1, ? p2 ) {

}
// swaps the id data members of two Student objects
// (assuming that the Student class has been
// defined and it has a public id data member)
void swapStudentIds( ? S1, ? S2 ) {

}
// swaps two Student objects
void swapStudentObjects( ? S1, ? S2 ) {

}
// swaps two Student object pointers
void swapStudentObjectPointers( ? P1, ? P2 ) {

}
```

```
int main() {
  int a, b, *c = &a, *d = &b;
  Student S, R, *X = &S, *Y = &R;

  // swap a and b
  swapIntegers( ?, ? );

  // swap c and d
  swapIntegerPointers( ?, ? );

  // swap the ids of S and R
  swapStudentIds( ?, ? );

  // swap S and R
  swapStudentObjects( ?, ? );

  // swap X and Y
  swapStudentObjectPointers( ?, ? );

 return 0;
}
```

Principle of least privilege

- *Functions should not be given the capability to modify its parameters unless it is absolutely necessary*

- Non-constant pointer to non-constant data (no `const` qualifier is used)
  - Pointer value can be modified
  - Data value can be modified *through dereferencing* this pointer

```
int data;
int* ptr;
ptr = &data;        // pointer ptr can be modified
*ptr = 5;           // dereferencing operator can be applied on ptr
```

Principle of least privilege

- Non-constant pointer to constant data (using `const` qualifier)
  - Pointer value can be modified
  - Data value **cannot be** modified through dereferencing this pointer

```
int data;
const int* ptr;
ptr = &data;       // pointer ptr can be modified
// *ptr = 5;       Compile-time error: dereferencing operator CANNOT be
//                                     applied on ptr
```

Another example:

```
void display( const int* arr, const int size ) {
   for ( int i = 0; i < size; i++ )
      cout << arr[i] << endl;
   // *arr = 5;    Compile-time error: dereferencing operator CANNOT be
   //                                  applied on ptr
   // arr[2] = 2;  Compile-time error: subscript operator uses dereferencing
}
```

Principle of least privilege

- Constant pointer to non-constant data (using `const` qualifier)
  - Pointer value **cannot be** modified
    - Must be initialized at its declaration
    - Always points to the same location
    - Default for an automatically allocated array name (but not for a dynamically allocated array name)
  - Data value can be modified through dereferencing this pointer

```
int data;
int*const ptr = &data;
*ptr = 5;                 // dereferencing operator can be applied on ptr
// ptr = NULL;            Compile-time error: ptr CANNOT be modified
// int*const ptr2;        Compile-time error: ptr2 must be initialized
```

Another example:

```
int arr[5];               // automatically allocated array declaration
// arr = new int [4];     Compile-time error: automatically allocated array
                          name is constant
```

Dynamic memory management

- Enables programmers to allocate and deallocate memory for any built-in or user-defined type using the `new` and `delete` operators
- Operator **`new`**
  - Allocates memory for an object or an array from the heap (free-store) at execution time
  - Returns the starting address of the allocated memory
  - Calls a constructor if it is called for an object (or for all objects in an array of objects)

```
Book* B1 = new Book( 2001, 20.5 );    // dynamically allocates a single object
Book* B2 = new Book [5];              // dynamically allocates an array of
                                      // five objects

double* d = new double [4];           // dynamically allocates an array of
                                      // four doubles
```

Dynamic memory management

- Enables programmers to allocate and deallocate memory for any built-in or user-defined type using the `new` and `delete` operators
- Operator **`delete`**
  - Deallocates (releases) the memory allocated for an object or an array from the heap
  - So that, this deallocated memory can be reused for other allocations
  - Calls the destructor if it is called for an object (or for an array of objects)
  - When `[]` is put after delete, it calls the destructor for every object in the array. Otherwise, it calls the destructor just for the first object.

```
delete B1;          // deallocates a single object
delete [] B2;       // deallocates an array of objects
delete [] d;        // deallocates an array of doubles
```

MEMORY LEAK happens when you do not release the memory which is no longer needed. There is no built-in garbage collection in C or C++.

Dynamic memory management **for a single object**

- An object can be allocated in two different ways: (1) through object variable declaration and (2) by dynamic allocation using the `new` operator

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

class Book {
public:
  Book( int i = 0 );
  void displayId();

private:
  int id;
};
Book::Book( int i ) {
  id = i;
}
void Book::displayId()
{
  cout << id << endl;
}
```

```
void foo() {

  Book X( 10 );          // X is a local object allocated from the stack
                         // through variable declaration.

  Book* Y;               // Y is a local pointer variable stored in the
                         // stack. It is not an object but can keep the
                         // starting address of memory allocated for
                         // an object.

  Y = new Book( 20 );    // The new operator dynamically allocates
                         // memory for an object from the heap and
                         // returns the starting address of this memory.
                         // The assignment operator puts this returned
                         // address inside pointer Y.

  // continues with the next slide
```

Dynamic memory management **for a single object**

- An object can be allocated in two different ways: (1) through object variable declaration and (2) by dynamic allocation using the `new` operator

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

class Book {
public:
  Book( int i = 0 );
  void displayId();

private:
  int id;
};
Book::Book( int i ) {
  id = i;
}
void Book::displayId()
{
  cout << id << endl;
}
```

```
// ... void foo() function continues

X.displayId();       // It invokes displayId() for object X.


// Y.displayId();    Compile-time error since Y is not an object.


(*Y).displayId();    // It first accesses the dynamically allocated
                     // object by dereferencing pointer Y and then
                     // invokes displayId() for the accessed object.


Y->displayId();      // The arrow operator is defined for pointers.
                     // Y->displayId() is equivalent to
                     // (*Y).displayId()
```

Dynamic memory management **for a single object**

- An object can be allocated in two different ways: (1) through object variable declaration and (2) by dynamic allocation using the `new` operator

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

class Book {
public:
  Book( int i = 0 );
  void displayId();

private:
  int id;
};
Book::Book( int i ){
  id = i;
}
void Book::displayId()
{
  cout << id << endl;
}
```

```
// ... void foo() function continues

delete Y;            // The delete operator releases the allocated
                     // memory whose starting address is kept in
                     // pointer Y. But, it does not release the
                     // pointer itself.

// delete X;         Compile-time error since X is not an address.

delete &X;           // Program crash since X is not dynamically
                     // allocated. The delete operator should only
                     // be called for the objects that are
                     // dynamically allocated from the heap.

Y = &X;              // It assigns the address of X to Y.
Y->displayId();
delete Y;            // Program crash since Y does not keep the
                     // address of an object that is dynamically
                     // allocated from the heap.
```

Dynamic memory management **for a single object**

- An object can be allocated in two different ways: (1) through object variable declaration and (2) by dynamic allocation using the `new` operator

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

class Book {
public:
  Book( int i = 0 );
  void displayId( );

private:
  int id;
};
Book::Book( int i ) {
  id = i;
}
void Book::displayId()
{
  cout << id << endl;
}
```

```
  // ... void foo() function continues

  Y = new Book( 30 );
  Y = new Book( 40 ); // Memory leak since the object allocated by
                      // the previous statement is not released and
                      // cannot be accessed anymore.

  delete Y;
  delete Y;           // Program crash since the memory pointed by
                      // Y has already been released in the previous
                      // statement. Note that some runs may cause
                      // this program to crash while some others
                      // may not. However, there is always a logic
                      // error here and you have to fix this problem.

}
```

Dynamic memory management **for a single object**

Write a function that allocates a `Test` *object and returns it to the caller*

```
class Test {
public:
   Test( int i = 0 );
   void setNext( Test* nval );
   Test* getNext( );

private:
   int id;
   Test* next;
};
Test::Test( int i ) {
   id = i;
   next = NULL;
}
void Test::setNext( Test* nval ){
   next = nval;
}
Test* Test::getNext() {
   return next;
}
```

```
Test returnObject( int oid ){
   Test obj( oid );       // Local object declaration
   return obj;            // This statement returns a
                          // copy of the local object to
                          // the caller function
}
int main(){
   Test T = returnObject( 300 );

   // Compile-time error: T is an object not an address
   // delete T;

   return 0;
}
```

// BE CAREFUL: That is different than Java!!!

Dynamic memory management **for a single object**

Write a function that dynamically allocates a `Test` *object and returns its address to the caller*

```
class Test {
public:
   Test( int i = 0 );
   void setNext( Test* nval );
   Test* getNext( );

private:
   int id;
   Test* next;
};
Test::Test( int i ) {
   id = i;
   next = NULL;
}
void Test::setNext( Test* nval ){
   next = nval;
}
Test* Test::getNext() {
   return next;
}
```

```
Test* returnAddress( int oid ){
   Test* A = new Test( oid );
   return A;              // Returns address A, which
                          // keeps the address of the
                          // dynamically allocated object
}
int main(){
   Test* first = new Test( 400 );
   first->setNext( returnAddress( 500 ) );

   // call the delete operator to avoid memory leak
   delete first->getNext();
   delete first;

   return 0;
}
```

Dynamic memory management **for a single object**

Write a function that dynamically allocates a `Test` *object and returns its address to the caller*

```
class Test {
public:
   Test( int i = 0 );
   void setNext( Test* nval );
   Test* getNext( );

private:
   int id;
   Test* next;
};
Test::Test( int i ) {
   id = i;
   next = NULL;
}
void Test::setNext( Test* nval ){
   next = nval;
}
Test* Test::getNext() {
   return next;
}
```

```
// Logic error: Memory for object A is taken from the
// stack and is released when the function returns.
// In general, do not return the address of any stack
// memory associated with a local variable.
Test* incorrectReturnAddress( int oid ){
   Test A( oid );
   return &A;
}
int main(){
   Test* ptr = incorrectReturnAddress( 600 );

   delete ptr; // Program crash: ptr keeps the address of
               // a released stack memory (which was not
               // dynamically allocated from the heap).
               // The delete operator should only be
               // called for the objects that are
               // dynamically allocated from the heap.

   return 0;
}
```

Arrays

Arrays are data structures that contain values of any non-reference data type. Array items are of the same type and kept in consecutive memory locations.

In C++, there are two ways to define arrays

- An automatically allocated array through declaration
  - Always remains the same size once created
  - Its name is a constant pointer that keeps the starting address of the array items
  - If it is a local declaration, memory for array items is taken from the stack
- A dynamically allocated array
  - Memory for array items is taken from the heap using the `new` operator
  - Starting address of the allocated memory is kept in a pointer whose value can be changed

```
void foo() {
    // Automatically allocated array
    // declaration
    // Neither the value of arr1 nor
    // its size can be changed.
    int arr1[5];

    // Dynamic array allocation
    // Array items are dynamically
    // allocated and arr2 keeps the
    // starting address of the first
    // item. Throughout the execution,
    // this array can be released and
    // re-allocated.
    int* arr2 = new int [4];
    delete [] arr2;
    arr2 = new int [5];
    delete [] arr2;
}
```

Declaring automatically allocated arrays

- Array size should be specified at its declaration
  - In standard C++, the size should be either an integer literal or a constant integer variable
  - It should be positive
  - It cannot be changed throughout the execution

```
const int arrSize = 5;  // B1 and B2 are arrays of Book objects. They are
Book B1[ arrSize ];     // declared as automatically allocated arrays.
Book B2[ 12 ];          // If these are local declarations, their items
                        // are kept in the stack.

// int no = 3;          Compile-time error: In standard C++, the size used in
// Book B3[ no ];       an array declaration should be a literal or a constant
//                      variable. no is not a constant variable.

// delete [] B2;        Compile-time error: B2 is not a dynamic array.
```

- Multiple arrays of the same type can be declared in a single declaration

```
int D[ 5 ], E, F[ 6 ]; // D and F are integer arrays whereas E is an integer.
```

Initializing automatically allocated arrays

- If it is a local array declaration, array items have garbage values
- Array items can be initialized at array declaration using an initializer list
  - You may omit the array size if you use an initializer list. In this case, the compiler determines the array size based on the number of initializers.

```
int n1[ ] = { 10, 20, 30, 40, 50 };
```

  - If the list contains less initializers, the remaining array items are initialized with 0

```
int n2[ 10 ] = { 0 };
```

  - If the list contains more initializers, the compiler gives a compile-time error

```
int n3[ 3 ] = { 0, 10, 30, 50 };
```

Subscript operator

- It provides a direct access to an individual array item whose position is specified as an integer expression in square brackets
  - *Very efficient* as the access time is independent of the position specified in square brackets

- Regardless of whether it is an automatically allocated array or a dynamically created array, the subscript operator dereferences the address calculated using the pointer value (or the array name) and the value specified in square brackets

```
int A[ 4 ];
int* B = new int [ 5 ];
A[ 2 ] = 10;          // it is  equivalent to *(A + 2) = 10
B[ 3 ] = 20;          // it is  equivalent to *(B + 3) = 20
```

In order to understand how the subscript operator works, you need to understand pointer (address) arithmetic

Pointer (address) arithmetic

- Adding/subtracting an integer to/from a pointer
  - Using `++` , `+` , `+=` , `-` , `--`, `-=` operators
- Subtracting one pointer from another (two pointers should be of the same type)

Example: Consider the following declarations on a machine that uses 4 bytes to represent `int`

```
int  arr1[ 4 ];
int* arr2 = new int [ 5 ];
int* ptr  = arr2 + 3;
```

Pointer arithmetic is meaningless if not performed on a pointer to an array. It is a logic error.

`arr1` : address of the first item in the declared array (whose index is 0)

`arr2` : address of the first item in the dynamically allocated array (whose index is 0)

`ptr` : address obtained by adding the number of bytes to represent 3 integers (in our example, 12 bytes) to `arr2`. Thus, it gives the address of the fourth array item (whose index is 3)

`ptr - arr2` : subtracts these two pointers and then gives how many integers (as they are integer pointers) one can keep in the difference (in our example, 3)

Subscript operator

- C++ has **no array bounds checking**
  - C++ does not prevent the computer from referring to memory that has not been allocated
  - That is, C++ does not prevent the computer from dereferencing an unallocated memory
    - No compile-time error
    - May cause execution-time error (program crash)
    - But always logic error

```
int A[ 4 ];
int* B = new int [ 5 ];

// Each statement below always has a logic error and may cause a program crash.
// However, a program crash may occur in one run and may not in another. Thus,
// it is hard to detect statements with this logic error. Such statements often
// result in changes to the value of an unrelated variable or a fatal error
// that terminates the program execution.
A[ 5 ]  = 30;        // it is  equivalent to *(A + 5) = 30
B[ 50 ] = 40;        // it is  equivalent to *(B + 50) = 40
```

Passing arrays to functions

- Functions can take arrays as arguments
- Function parameter list must specify an array as a parameter

```
void init( int* A, int size, int C ) {
void init( int A[], int size, int C ) {
void init( int A[40], int size, int C ){
```

All three are equivalent. In the last one, the size specified in square brackets will be ignored by the compiler.

```
// Write a function to initialize the
// array items with the value of C
void init( int* A, int size, int C ) {
   for ( int i = 0; i < size; i++ )
      A[i] = C;
}
int main() {
   int A1[ 40 ];
   int *A2 = new int [ 60 ];

   init( A1, 40, 10 );
   init( A2, 60, 100 );

   // Initialize the items in the
   // first half of the array with 1
   // and those in the second half
   // of the array with -1
   init( A2, 30, 1 );
   init( A2 + 30, 30, -1 );

   delete [] A2;
   return 0;
}
```

Passing arrays to functions

- When this function is called, only an address will be passed
  - This will be considered as the starting address of an array in the called function
  - So the function knows where the array starts but does not know where it ends
  - Thus, the array size should be defined as another function parameter
- `init` function used pass-by-value to define its pointer (array) parameter `A`
  - Thus, changes in the value of this parameter cannot be seen after returning from the function
  - However, changes in the values of array items will be seen as it accesses the array items through dereferencing (subscript operator)

```
// Write a function to initialize the
// array items with the value of C
void init( int* A, int size, int C ) {
   for ( int i = 0; i < size; i++ )
      A[i] = C;
}
int main() {
   int A1[ 40 ];
   int *A2 = new int [ 60 ];

   init( A1, 40, 10 );
   init( A2, 60, 100 );

   // Initialize the items in the
   // first half of the array with 1
   // and those in the second half
   // of the array with -1
   init( A2, 30, 1 );
   init( A2 + 30, 30, -1 );

   delete [] A2;
   return 0;
}
```

Example: Write a global function that returns a double array with the size of 10

```
// createArray_1 returns the array as a return value
double* createArray_1( ) {
   return new double [ 10 ];
}
// createArray_2 returns the array from the parameter list
// (using a reference parameter)
void createArray_2( double*& arr ) {
    arr = new double [ 10 ];
}
// createArray_3 returns the array from the parameter list
// (without using a reference parameter but simulating
// pass-by-reference using a pointer)
void createArray_3( double** arr ) {
    *arr = new double [ 10 ];
}
// What is wrong with the following two functions?
// void incorrectCreateArray_1( double* arr ) {
//    arr = new double [ 10 ];
//}
// double* incorrectCreateArray_2( ) {
//    double arr[ 10 ];
//    return arr;
// }
```

```
int main() {
   double* D;

   D = createArray_1();
   delete [] D;

   createArray_2( D );
   delete [] D;

   createArray_3( &D );
   delete [] D;

   return 0;
}
```

Example: Write a global function that takes an array and its size as input and creates an exact copy of this input array and returns it (*use the parameter list to return the output*)

```
// shallowCopy_1 and shallowCopy_2 create a shallow copy of the input array
// What is wrong with the shallow copy operation?

void shallowCopy_1( double* arr, const int arrSize, double*& output ) {
   output = arr;
}

void shallowCopy_2( double* arr, const int arrSize, double** output ) {
   *output = arr;
}
```

Example: Write a global function that takes an array and its size as input and creates an exact copy of this input array and returns it (*use the parameter list to return the output*)

```
void deepCopy_1( const double* arr, const int arrSize, double*& output ) {
   output = new double [ arrSize ];
   for (int i = 0; i < arrSize; i++)
      output[ i ] = arr[i];

   // arr[ 0 ] = 10;          Compile-time error since arr is a pointer to constant data
   //                         dereferencing (subscript) operator cannot be used as
   //                         an l-value
}

void deepCopy_2( const double* arr, const int arrSize, double** output ) {
   *output = new double [ arrSize ];
   for (int i = 0; i < arrSize; i++)
      (*output)[ i ] = arr[i];
}
```

If you need a separate copy of an array (array items), use DEEP COPY, do not shallow copy the array. Otherwise, you will have two pointer variables (array names) that point to the same array. That is, if you change an item of one array, you will also see this change in the other array.

Example: Write a global function that partitions an array such that its first part contains items smaller than a pivot value and its second part contains greater items (*use pointer iterators*)

```
void partition( int arr[], int arrSize, int pivot ) {
   int* first = arr;
   int* second = arr + arrSize - 1;      // Equivalent to second = &arr[ arrSize - 1 ];

   while ( first <= second ) {           // Relational operator defined on pointers
                                         // Meaningless if its operands do not point to
                                         // the items of the same array
      if ( *first >= pivot && *second < pivot ) {
         int temp = *first;
         *first = *second;
         *second = temp;
         first++;
         second--;
      }
      else {
         if ( *first < pivot )
            first++;
         if ( *second >= pivot )
            second--;
      }
   }
}
```

Example: Write a global function that searches a `Student` object whose id is equal to the given key in the input array and returns the address of this object

```
Student* findStudent( const Student* stuArr, const int stuSize, const int key ) {
   for ( int i = 0; i < stuSize; i++ )
      if ( stuArr[i].getId() == key )
         return stuArr + i;
   return NULL;
}

int main() {
   Student* arr = new Student [ 10 ];
   // ...
   if ( findStudent( arr, 10, 2000 ) == NULL )     // Equality operator
      cout << "Unsuccessful search" << endl;
   else
      cout << "Successful search" << endl;

   delete [] arr;              // Deallocates the student objects in the array
                               // When [] is put after delete, it calls the destructor
                               // for each object in the array. Otherwise, it calls the
                               // destructor only for the first object
                               // (we will see it again after discussing destructors)
   return 0;
}
```

Example: Extend the GradeBook class such that it keeps the grades of multiple students (use an automatically allocated array)

```
class GradeBook {
public:
   const static int studentNo = 10;
   GradeBook( int, const int [] );
   // ...
private:
   int courseNo;
   int grades[ studentNo ];
};
GradeBook::GradeBook( int no, const int S[] ) {
   courseNo = no;
   for ( int i = 0; i < studentNo; i++ )
      grades[i] = S[i];
}
int main() {
    int A[ GradeBook::studentNo ];
    GradeBook G( 201, A );
    return 0;
}
```

`static` data members

(also called class variables)

- One copy is shared among all objects of the class
- Each object does not have a separate copy
- Can be accessed even when no object of the class exists using the class name followed by the binary scope resolution operator

One final remark

- `sizeof` operator returns the size of its operand in bytes
  - Can be used with a variable name, a type name, or a constant value
  - Number of bytes to store a particular type may vary from one computer system to another

- It is used to make allocations in C
  - Some students may try to use it to get the size of an array. However, this mostly causes an incorrect result.

```
// Use of sizeof in C style allocations
int *A1, *A2, *A3;
double *D;

A1 = (int* ) malloc( 20 * sizeof(int) );
A2 = (int* ) calloc( 7, sizeof(int) );
A3 = (int* ) realloc( A3, 10 * sizeof(int) );
D  = (double* ) calloc( 5, sizeof(double) );
```

```
int incorrectUse( int A[] ) {
   int size = sizeof(A) / sizeof(int);
   return size;
}
int main() {
   int A[] = { 1, 2, 3, 4, 5} ;
   int* B = new int [ 5 ];

   cout << sizeof(A) / sizeof(int);
   cout << sizeof(B) / sizeof(int);
   cout << incorrectUse(A);
   cout << incorrectUse(B);
   return 0;
}
```

C-style strings

- In C++, there is a `string` class
- In C, character pointers (arrays) are used to store and manipulate strings
  - `<cstring>` header file contains many C-style string manipulation functions
  - `cin >>` and `cout <<` work differently for character pointers (different than how they work on pointers of the other types)
  - They all assume that a C-style string ends with a null character (otherwise they will not work correctly)

When used with `cin >>`, the array should be large enough to store the string typed at the keyboard. Otherwise, it may result in data loss (logic error) and may cause program crashes (run-time errors).

```
int main() {
   // char array declaration using an
   // initializer list, equivalent to
   // char a[] = {'H', 'i', '\0'};
   char a[] = "Hi";

   cout << a << endl;
   a[ 1 ] = '\0';
   cout << a << endl;
   cin >> a;
   cout << a << endl;

   char *b = new char [ 10 ];
   cin >> b;
   cout << b << endl;
   delete []b;

   return 0;
}
```

<cstring> functions

Function prototype	Function description
`char *strcpy( char *s1, const char *s2 );`	Copies the string `s2` into the character array `s1`. The value of `s1` is returned.
`char *strncpy( char *s1, const char *s2, size_t n );`	Copies at most `n` characters of the string `s2` into the character array `s1`. The value of `s1` is returned.
`char *strcat( char *s1, const char *s2 );`	Appends the string `s2` to `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`char *strncat( char *s1, const char *s2, size_t n );`	Appends at most `n` characters of string `s2` to string `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`int strcmp( const char *s1, const char *s2 );`	Compares the string `s1` with the string `s2`. The function returns a value of zero, less than zero or greater than zero if `s1` is equal to, less than or greater than `s2`, respectively.
`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to `n` characters of the string `s1` with the string `s2`. The function returns zero, less than zero or greater than zero if the `n`-character portion of `s1` is equal to, less than or greater than the corresponding `n`-character portion of `s2`, respectively.

Function prototype	Function description
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to `strtok` breaks string `s1` into *tokens*—logical pieces such as words in a line of text. The string is broken up based on the characters contained in string `s2`. For instance, if we were to break the string `"this:is:a:string"` into tokens based on the character `':'`, the resulting tokens would be `"this"`, `"is"`, `"a"` and `"string"`. Function `strtok` returns only one token at a time—the first call contains `s1` as the first argument, and subsequent calls to continue tokenizing the same string contain `NULL` as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, `NULL` is returned.
`size_t strlen( const char *s );`	Determines the length of string `s`. The number of characters preceding the terminating null character is returned.

C++ `string` class

- `string` objects are mutable
- `string` class provides many operators and member functions
  - subscript operator `[]` to access the character at a specified index
  - concatenation operators `+` and `+=`
  - comparison operators `==`, `!=`, `<`, `<=`, `>=`, `>` to compare two strings
  - member functions such as `append()`, `insert()`, `erase()`
  - `cin >>` and `cout <<` input/output operations

    `cin >> str` *reads just one word into* `str`. *To read the entire line, use the global function* `getline( cin, str )`.

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

int main() {

   string s1 = "happy", s2 = "birthday", s3;

   // subscript operator
   s1[0] = 'H';
   cout << "s1: " << s1 << endl;
   cout << "first char: " << s1[0] << endl;

   // string concatenation
   s1 += s2;
   cout << "s1: " << s1 << endl;
   cout << "s2: " << s2 << endl;


   // continues with the next slide
```

C++ `string` class

- `string` objects are mutable
- `string` class provides many operators and member functions
  - subscript operator `[]` to access the character at a specified index
  - concatenation operators `+` and `+=`
  - comparison operators `==`, `!=`, `<`, `<=`, `>=`, `>` to compare two strings
  - member functions such as `append()`, `insert()`, `erase()`
  - `cin >>` and `cout <<` input/output operations

    `cin >> str` *reads just one word into* `str`. *To read the entire line, use the global function* `getline( cin, str )`.

```
// ... int main() function continues


// string comparison
// Outputs of the following statements
// are given as comments. The program
// uses 0 to display false and 1 to
// display true.

cout << ( s1 == s2 ) << endl;     // 0
cout << ( s1 != s2 ) << endl;     // 1
cout << ( s1 >  s2 ) << endl;     // 0
cout << ( s1 >= s2 ) << endl;     // 0
cout << ( s1 <  s2 ) << endl;     // 1
cout << ( s1 <= s2 ) << endl;     // 1



// continues with the next slide
```

C++ `string` class

- `string` objects are mutable
- `string` class provides many operators and member functions
  - subscript operator `[]` to access the character at a specified index
  - concatenation operators `+` and `+=`
  - comparison operators `==`, `!=`, `<`, `<=`, `>=`, `>` to compare two strings
  - member functions such as `append()`, `insert()`, `erase()`
  - `cin >>` and `cout <<` input/output operations

    `cin >> str` *reads just one word into* `str`. *To read the entire line, use the global function* `getline( cin, str )`.

```
    // ... int main() function continues

    cout << "Enter your name: ";
    cin >> s3;
    // cin reads just one word (reads the
    // console input until it encounters a
    // white space). To read the entire line
    // use getline( cin, s3 ) global function

    // some member functions
    cout << "length of s2: " << s2.length();
    cout << "length of s2: " << s2.size();

    s1.append( " " + s3 );
    cout << "s1: " << s1 << endl;
    s1.insert( 5, " and happy " );
    cout << "s1: " << s1 << endl;
    s1.erase( 5, 4 );
    cout << "s1: " << s1 << endl;

    return 0;
}
```

Example: What is the output of the following program?

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

void fun( string& x, string y ) {
   x.erase( 0, 2 );
   x.insert( 2, y.substr( 1, 2) );
   x.append( "***" );
   cout << "x: " << x << endl;

   y += x[0];
   y[2] = 'e';
   cout << "y: " << y << endl;
}
int main() {
   string s1 = "python", s2 = "java";

   cout << "s1: " << s1 << endl;
   cout << "s2: " << s2 << endl;
   fun( s1, s2 );
   cout << "s1: " << s1 << endl;
   cout << "s2: " << s2 << endl;
   return 0;
}
```