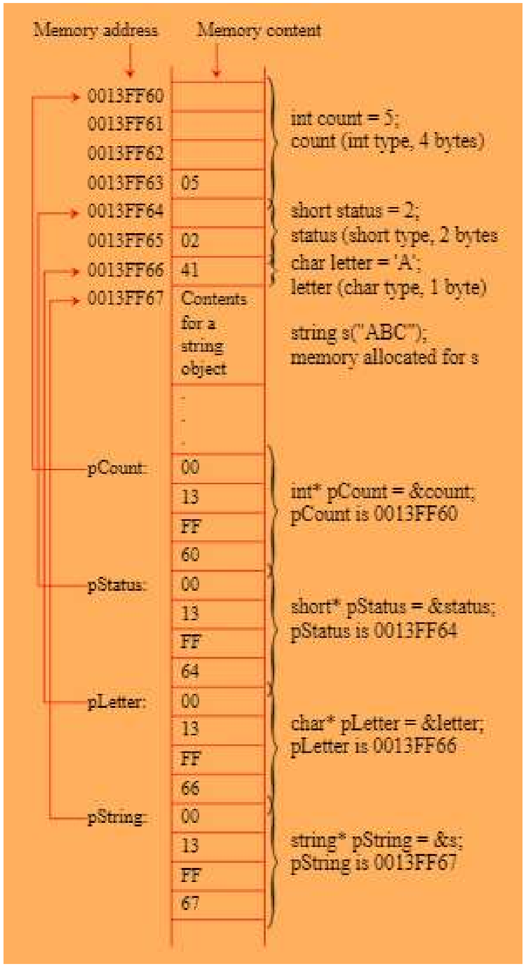
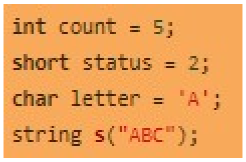
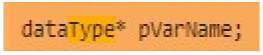
11.2 Pointer Basics

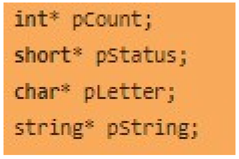
Monday, March 20, 2023 1:03 PM

- Pointer Var holds the mem address, use pointer for dereference operator (*) to access the actual value at a specific mem location
- *Pointer Variables* aka pointers are declared to hold mem addresses as their vals
- Usually a var has a data val (like integer, float-pt, a char)
- Pointer has mem address of var tho, and that has a data val
- 

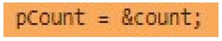
- Each byte of mem has unique address, vars address is address of the first byte allocated to that var
- Ex:
  - So if 4 vars are declared : count, status, letter, and s
  - 

    ```
    int count = 5;
    short status = 2;
    char letter = 'A';
    string s("ABC");
    ```
  - Count is declared as an int type, so has 4 bytes
  - Status is declared as short type, so has 2 bytes
  - Letter is declared as char type, so has 1 byte
  - S can be diff size bc it's a string, but its fixed after declared
- To declare a pointer, use this syntax
- 

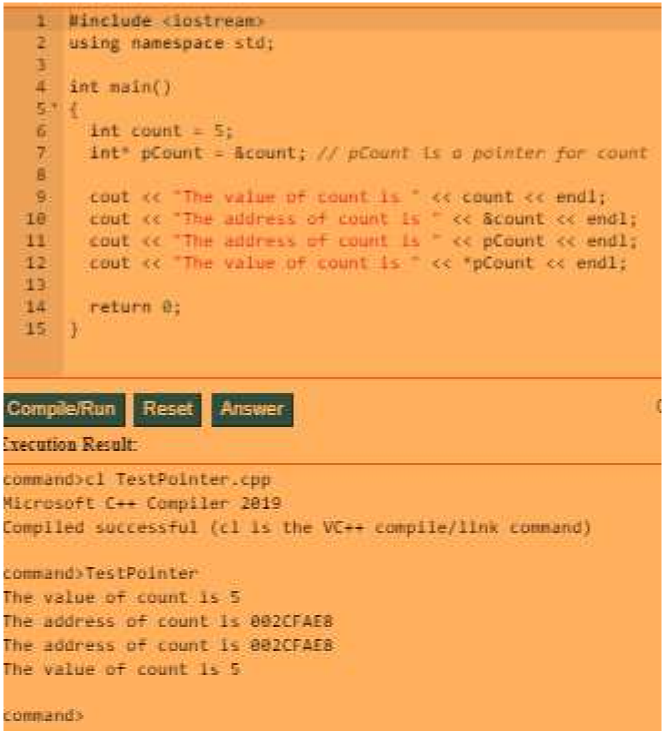
  ```
  dataType* pVarName;
  ```
- Each var being declared as pointer have to have * after

- Back to ex:
- To declare pointers for the ex vars, use the same syntax
- 

```
int* pCount;
short* pStatus;
char* pLetter;
string* pString;
```

- All the pointers point to their specific types
- Can now assign the address of a var to a pointer, like for pCount, assign the address of var count to it by:
- 

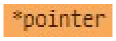
```
pCount = &count;
```

- & symbol (ampersand) is called the address operator **when put in front of a var**
- It's a Urinary op that returns the var's address
- 

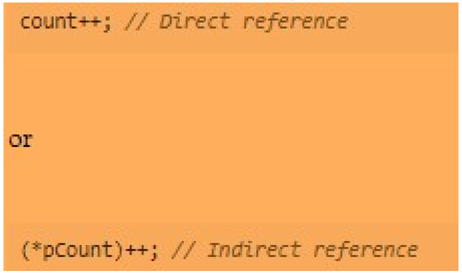
```
#include <iostream>
using namespace std;

int main()
{
  int count = 5;
  int* pCount = &count; // pCount is a pointer for count

  cout << "The value of count is " << count << endl;
  cout << "The address of count is " << &count << endl;
  cout << "The address of count is " << pCount << endl;
  cout << "The value of count is " << *pCount << endl;

  return 0;
}
```

Execution Result:

```
command>cl TestPointer.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestPointer
The value of count is 5
The address of count is 002CFAE8
The address of count is 002CFAE8
The value of count is 5

command>
```

  - pCount == &count; // correct
  - *pCount == &count // wrong
- Referencing a var thru a pointer usually called indirection, syntax is:
- 

```
*pointer
```

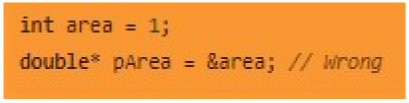
  - For ex:
  - Can increase count using
  - 

```
count++; // Direct reference
```

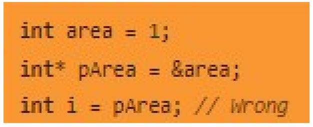
or

```
(*pCount)++; // Indirect reference
```

- The asterisk (*) used in that ^ is known as indirection operator aka deference operator (deference = indirect reference)
- When pointer is dereferenced, the val at the address stored in the pointer is retrieved
  - Can say that *pCount as value indirectly pointed by pCount, or just pointed by pCount
- Noteworthy stuff:
  - Asterisk (*) can be used in 3 diff ways in C++:

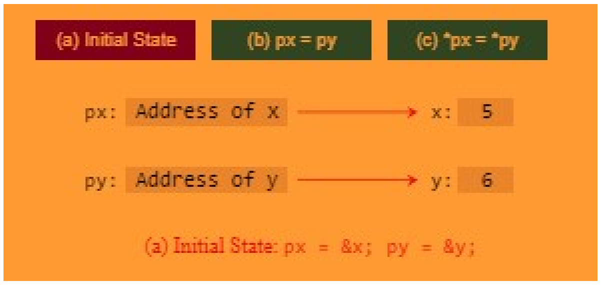
  - Multiplication operator
    - double area = radius * radius * 3.1415926;
  - Declare a pointer var
    - int* pCount = &count;
  - Deference operator
    - (*pCount)++;
- Compiler usually says what its used for in pgrm
- pointer var declared w/ a type (int/double/etc.), have to assign address of the var of same type, syntax error if types don't match
  - This is wrong
  - 

```
int area = 1;
double* pArea = &area; // Wrong
```

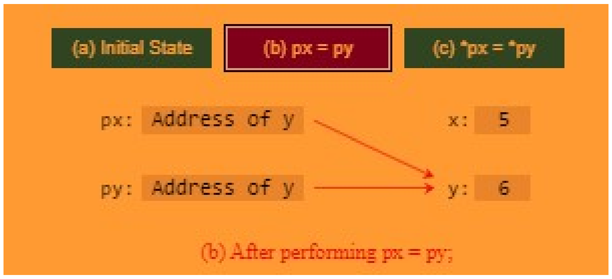
- Can assign pointer to another pointer of same type, but not to non-pointer var, this is wrong:
  - 

```
int area = 1;
int* pArea = &area;
int i = pArea; // Wrong
```

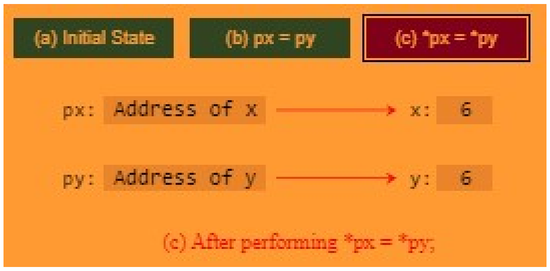
- Pointers are vars, so naming conventions still apply, usually use p prefix (like pCount/pArea), also array name is actually pointer
- Like local var, local pointer is assigned an arbitrary val if don't initialize it, can be initialized to 0 (special value, it points to nothing), to stop errors, always initialize pointers, dereferencing a pointer that isn't initialized can cause big runtime error/modify important data
  - Lots of C++ libraries (like <iostream>) define NULL as constant w/ val 0, more descriptive to use NULL instead of 0, better to use nullptr (C++11 intro, its keyword for null pointer) than NULL bc NULL can accidentally be redefined in the pgrm
-



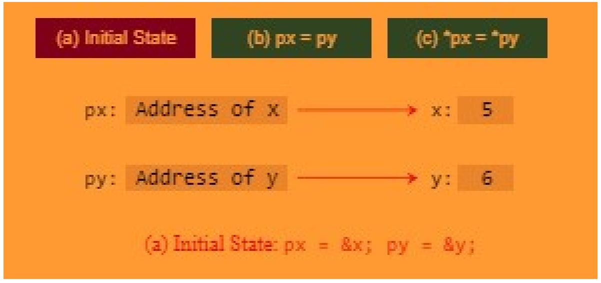
(a) Initial State: px = &x; py = &y;



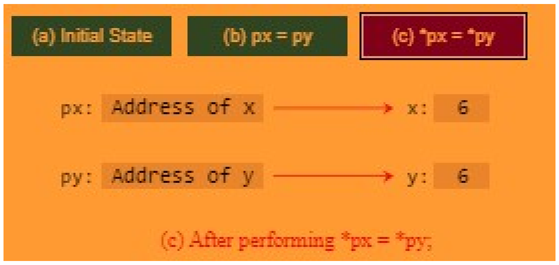
(b) After performing px = py;

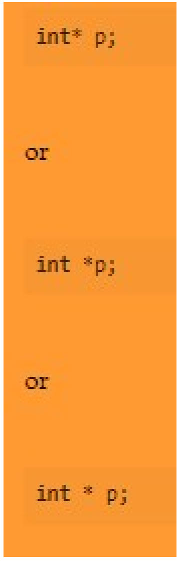
- 

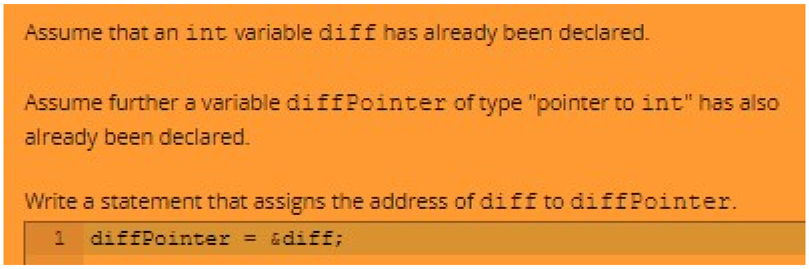
  px: Address of x → x: 6
  py: Address of y → y: 6
  (c) After performing *px = *py;
- But this is if go in order, so:



px: Address of x → x: 5
py: Address of y → y: 6
(a) Initial State: px = &x; py = &y;

-
- 

  px: Address of x → x: 6
  py: Address of y → y: 6
  (c) After performing *px = *py;
- This^ is also possibility
-
- pX stands for address, *pX stands for contents of the address
- Many diff syntax, all work (for pointer)
  - 
    ```
    int* p;

    or

    int *p;

    or

    int * p;
    ```
- Don't do this:
  - int* p1, p2;
- Do this:
  - int* p1;
  - int* p2;
-

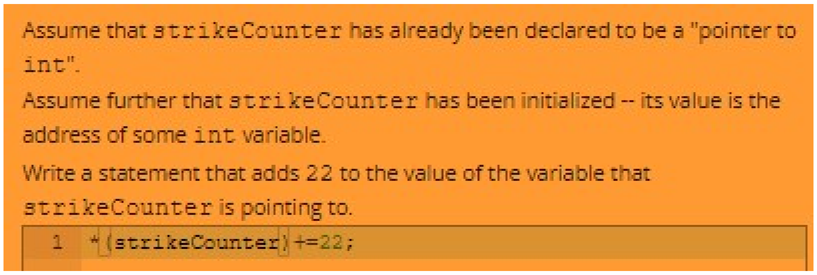


Assume that an int variable diff has already been declared.

Assume further a variable diffPointer of type "pointer to int" has also already been declared.

Write a statement that assigns the address of diff to diffPointer.

```
diffPointer = &diff;
```

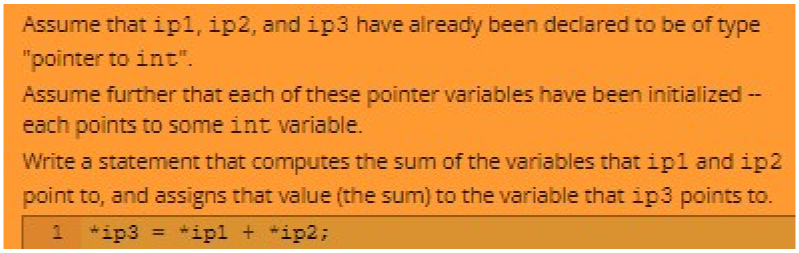


Assume that strikeCounter has already been declared to be a "pointer to int".

Assume further that strikeCounter has been initialized -- its value is the address of some int variable.

Write a statement that adds 22 to the value of the variable that strikeCounter is pointing to.

```
*(strikeCounter)+=22;
```

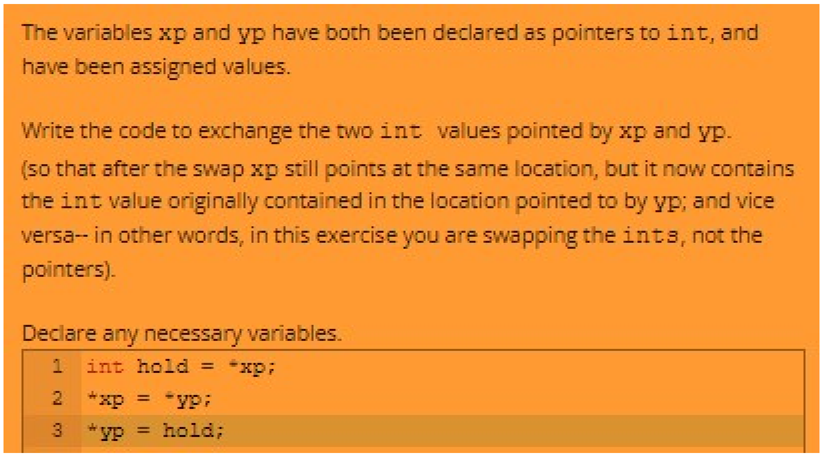


Assume that ip1, ip2, and ip3 have already been declared to be of type "pointer to int".

Assume further that each of these pointer variables have been initialized -- each points to some int variable.

Write a statement that computes the sum of the variables that ip1 and ip2 point to, and assigns that value (the sum) to the variable that ip3 points to.

```
*ip3 = *ip1 + *ip2;
```



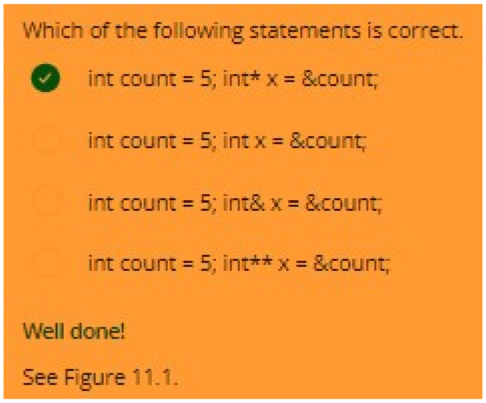
The variables xp and yp have both been declared as pointers to int, and have been assigned values.

Write the code to exchange the two int values pointed by xp and yp.

(so that after the swap xp still points at the same location, but it now contains the int value originally contained in the location pointed to by yp; and vice versa-- in other words, in this exercise you are swapping the ints, not the pointers).

Declare any necessary variables.

```
int hold = *xp;
*xp = *yp;
*yp = hold;
```



Which of the following statements is correct.

✓ int count = 5; int* x = &count;

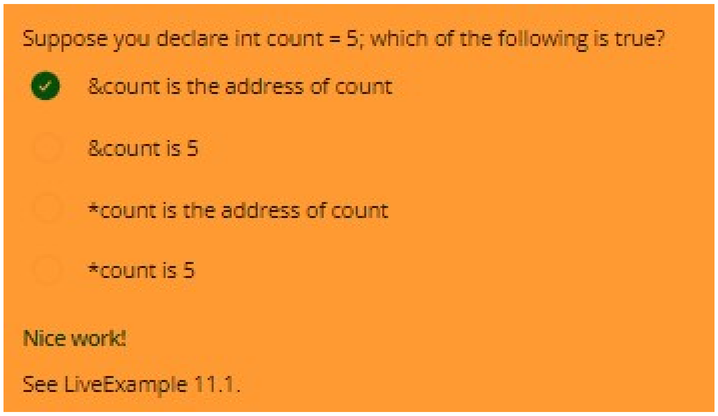
○ int count = 5; int x = &count;

○ int count = 5; int& x = &count;

○ int count = 5; int** x = &count;

Well done!

See Figure 11.1.

- 

  Suppose you declare int count = 5; which of the following is true?

  ✓ &count is the address of count

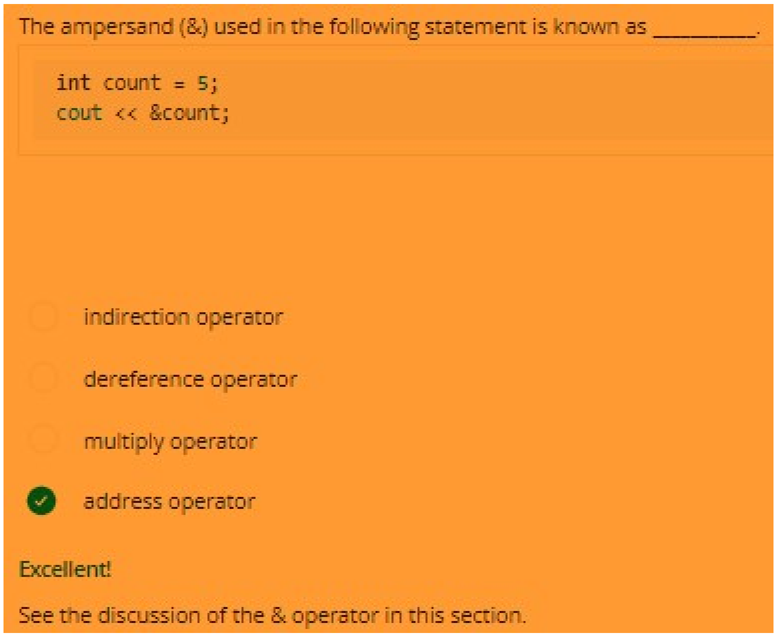
  ○ &count is 5

  ○ *count is the address of count

  ○ *count is 5

  Nice work!

  See LiveExample 11.1.

- 

  The ampersand (&) used in the following statement is known as ________.

  ```
  int count = 5;
  cout << &count;
  ```

  ○ indirection operator

  ○ dereference operator

  ○ multiply operator

  ✓ address operator

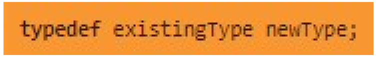
  Excellent!

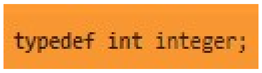
  See the discussion of the & operator in this section.

-
-

11.3 Defining Synonymous Types Using the typedef Keyword

Monday, March 20, 2023 8:55 PM

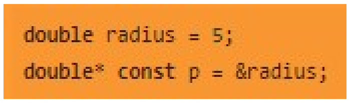
- Synonymous type can be defined the typedef keyword
- Unsigned type is synonymous to unsigned int
- C++ lets you define custom synonymous types using the typedef keyword
- Synonymous types can be used to simplify coding and avoid potential errors
- Syntax is:
- 

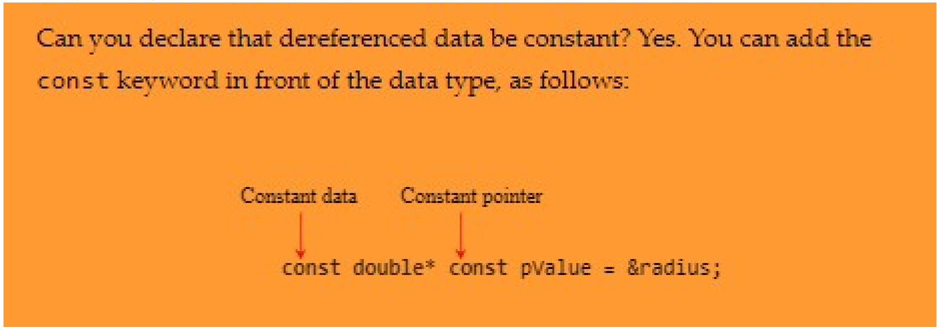
  ```
  typedef existingType newType;
  ```
- Ex:
- 

  ```
  typedef int integer;
  ```
- So can now declare an int var using
  - integer value = 40;
- The typedef declaration doesn't make new data types, just makes synonyms for a data type, useful for defining pointer type name to make the pgrm easy to read
  - Ex:
  - Can define a type for pointer:
  - typedef int* intPointer;
  - Now we can say easier things like:
  - intPointer p;
  - Same as : int*p;
- Easy and good to use bc avoid errors w/ missing *
  - If want to declare 2 pointer vars, this is wrong:
  - int* p1,p2;
  - Bbbuutt, this is correct:
  - intPointer p1, p2;
-

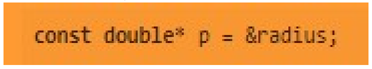
11.4 Using const w/ Pointers

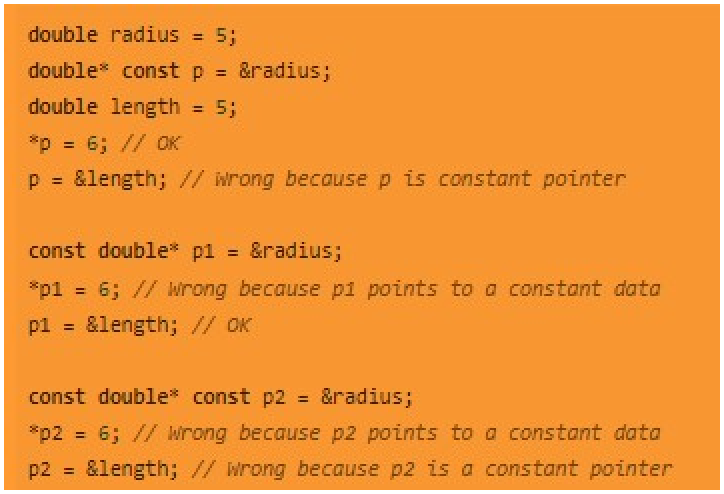
Monday, March 20, 2023 9:06 PM

- Constant pointer points to a constant mem location, but actual val in the mem location can be changed
- Already know how to declare constant using const keyword
- Can declare constant pointer (that's literally the whole sentence)
- Ex:
    - 

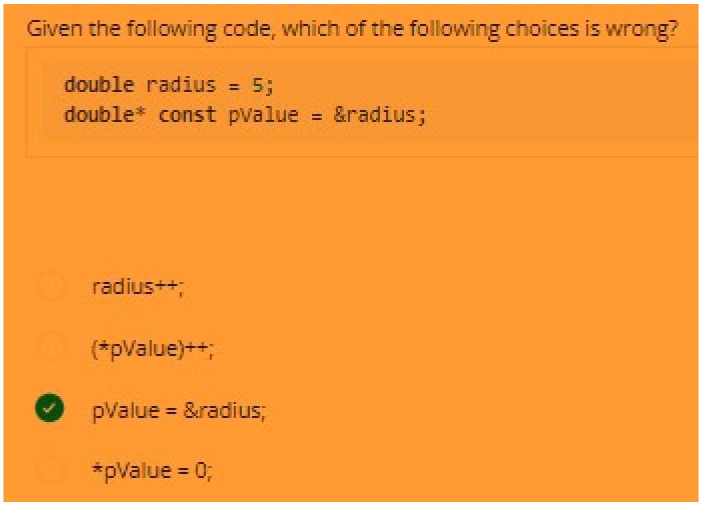
      ```
      double radius = 5;
      double* const p = &radius;
      ```
- Here is p, a constant pointer, must be declared an initialized in the same statement, cant assign a new address to p later, p is a constant, but data pointed to by p isnt constant
- So p* = 10; is still valid, it just changed the val to 10
- Also syntax/position is wiered, :
- 

  Can you declare that dereferenced data be constant? Yes. You can add the `const` keyword in front of the data type, as follows:

  ```
  const double* const pValue = &radius;
  ```
- For this^ tho, the pointer is a constant and the data pointed to by the pointer is also a constant
- If declare pointer as:
- 

  ```
  const double* p = &radius;
  ```
- Then pointer isn't constant, but data pointed to by the pointer is a constant
- Ex:
    - 

      ```
      double radius = 5;
      double* const p = &radius;
      double length = 5;
      *p = 6; // OK
      p = &length; // Wrong because p is constant pointer

      const double* p1 = &radius;
      *p1 = 6; // Wrong because p1 points to a constant data
      p1 = &length; // OK

      const double* const p2 = &radius;
      *p2 = 6; // Wrong because p2 points to a constant data
      p2 = &length; // Wrong because p2 is a constant pointer
      ```
-

- 

Given the following code, which of the following choices is wrong?

```
double radius = 5;
double* const pValue = &radius;
```

○ radius++;

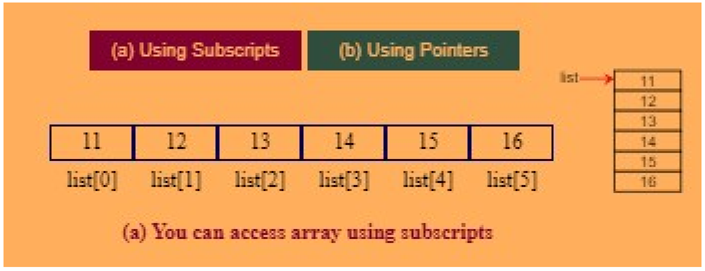
○ (*pValue)++;

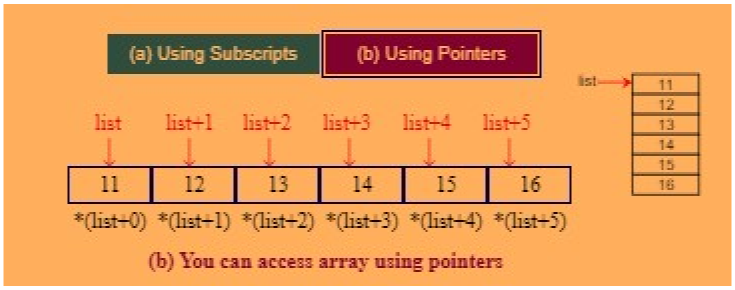
✓ pValue = &radius;

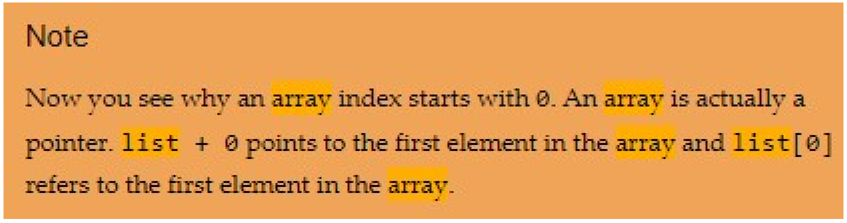
○ *pValue = 0;

11.5 Arrays and Pointers

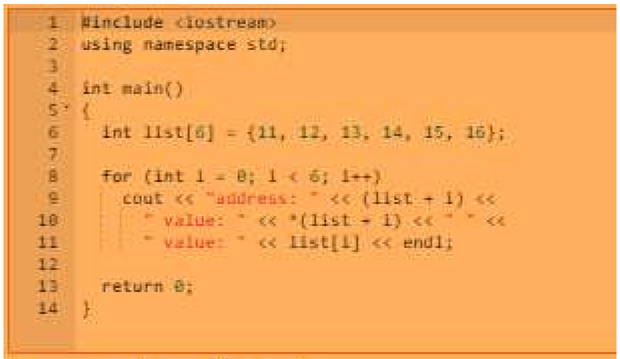
Monday, March 20, 2023 9:18 PM

- A C++ array name is actually a constant pointer to the first element in the array
- Array name actually reps the starting address of the array
- So array is basically a pointer
- Like this array:
- `int list[6] = {11, 12, 13, 14, 15, 16};`
- Has an address that can literally be shown by this
- `cout << "The starting address of the array is " << list << endl;`
- And this is possible to:
  - 

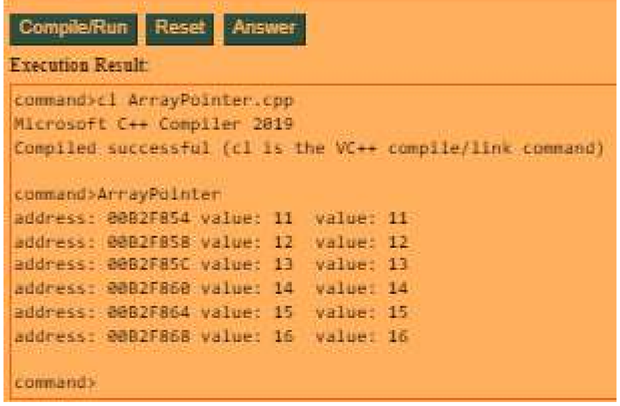
    (a) You can access array using subscripts
  -
  - 

    (b) You can access array using pointers
- An int can be added/subtracted from a pointer, it is incremented/decremented by that int times the size of the element
  - If list (an array) points to starting address of 1000, list + 1 won't be 1001, but it will be 1000 + sizeof(int)
  - This bc list is declares as array on int elements, C++ auto calc address for the next element by adding sizeof(int)
  - The sizeof(type) returns the size of a data type, size of each type is machine dependent
  - So no matter how big each element of the list, list + 1 points to the second element of the list, and list + 2 points to the second, ...
  - 
    Note

    Now you see why an array index starts with `0`. An array is actually a pointer. `list + 0` points to the first element in the array and `list[0]` refers to the first element in the array.
  -

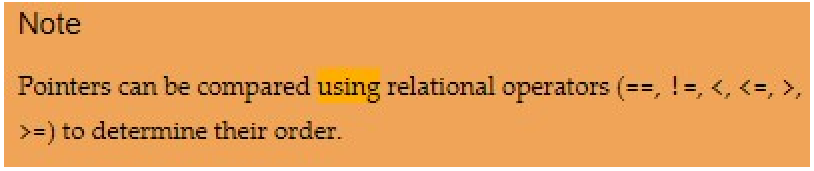


```
#include <iostream>
using namespace std;

int main()
{
  int list[6] = {11, 12, 13, 14, 15, 16};

  for (int i = 0; i < 6; i++)
    cout << "address: " << (list + i) <<
      " value: " << *(list + i) << " " <<
      " value: " << list[i] << endl;

  return 0;
}
```

- ○ 

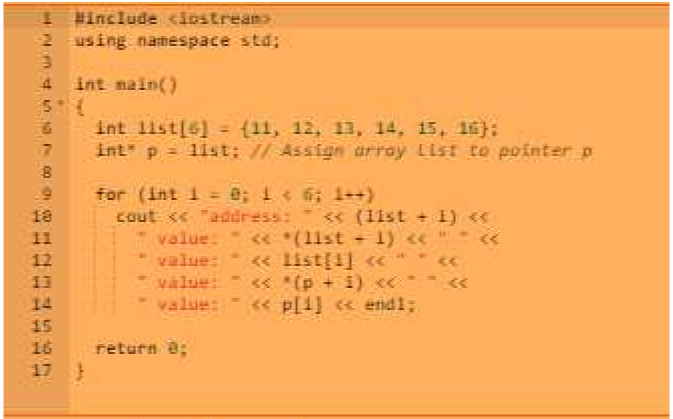
Execution Result:

```
command>cl ArrayPointer.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>ArrayPointer
address: 00B2F854 value: 11  value: 11
address: 00B2F858 value: 12  value: 12
address: 00B2F85C value: 13  value: 13
address: 00B2F860 value: 14  value: 14
address: 00B2F864 value: 15  value: 15
address: 00B2F868 value: 16  value: 16

command>
```

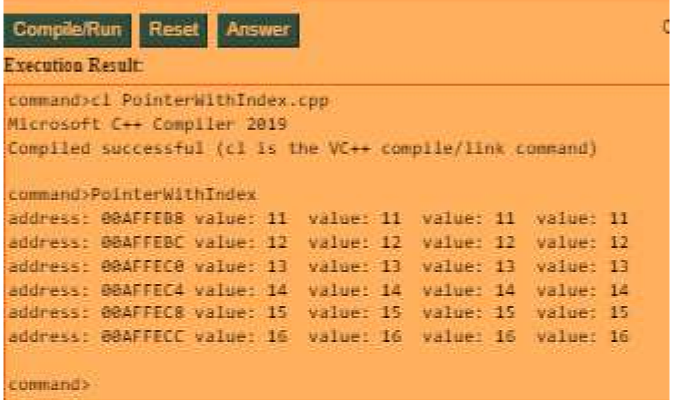
- So list[i] is same as *(list + i)
  - ○ *list + 1 literally adds 1 to value of first element of array
  - ○ *(list + 1) deferences the element at address (list + 1) in the array
  - ○ 

    Note

    Pointers can be compared using relational operators (==, ! =, <, <=, >, >=) to determine their order.
- Arrays and pointers form close relationship, pointer for an array can be used just like an array, can even use pointer w/ index

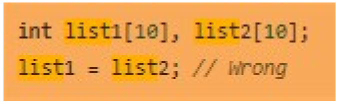


```
#include <iostream>
using namespace std;

int main()
{
  int list[6] = {11, 12, 13, 14, 15, 16};
  int* p = list; // Assign array list to pointer p

  for (int i = 0; i < 6; i++)
    cout << "address: " << (list + i) <<
      " value: " << *(list + i) << " " <<
      " value: " << list[i] << " " <<
      " value: " << *(p + i) << " " <<
      " value: " << p[i] << endl;

  return 0;
}
```

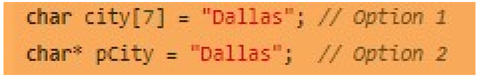
- 

Execution Result:

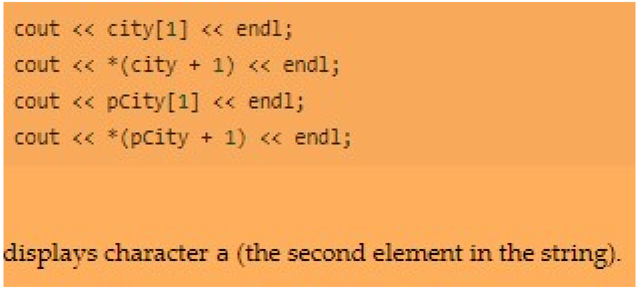
```
command>cl PointerWithIndex.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>PointerWithIndex
address: 00AFFEB8 value: 11  value: 11  value: 11  value: 11
address: 00AFFEBC value: 12  value: 12  value: 12  value: 12
address: 00AFFEC0 value: 13  value: 13  value: 13  value: 13
address: 00AFFEC4 value: 14  value: 14  value: 14  value: 14
address: 00AFFEC8 value: 15  value: 15  value: 15  value: 15
address: 00AFFECC value: 16  value: 16  value: 16  value: 16

command>
```

- Don’t need to use address operator & to assign the address of the array to the pointer, bc the name of the array is already the starting address of the array
  - Line is equivalent to int* p = &list[0];
  - Where &list reps the address of list[0]
- 1 big diff, once array is declared, cant change the address:
- 

```
int list1[10], list2[10];
list1 = list2; // Wrong
```

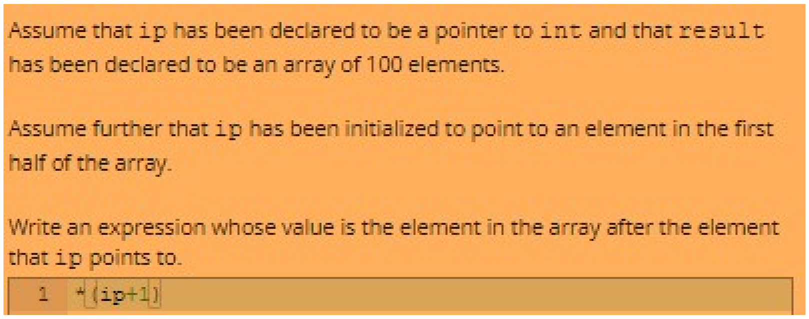
- Array name is kinda treated as a constant pointer in C++
- C-strings sometimes referred to as pointer-based strings, bc can be conveniently access using pointers
- 

```
char city[7] = "Dallas"; // Option 1
char* pCity = "Dallas";  // Option 2
```

- Each declaration makes a sequence that has chars 'D', 'a', 'l', 'l', 'a', 's', and '\0'
- Can access city or pCity using the array syntax/pointer syntax
- 

```
cout << city[1] << endl;
cout << *(city + 1) << endl;
cout << pCity[1] << endl;
cout << *(pCity + 1) << endl;
```

displays character a (the second element in the string).

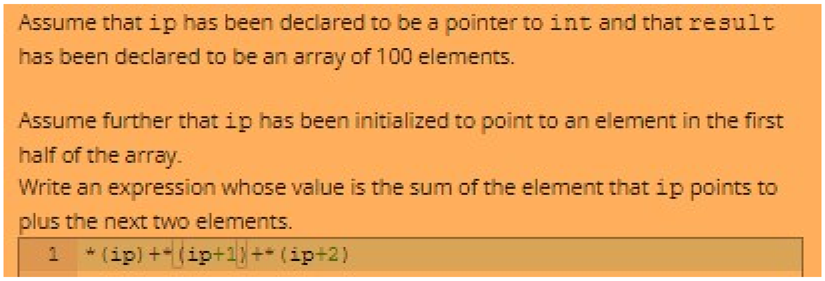
- 

Assume that ip has been declared to be a pointer to int and that result has been declared to be an array of 100 elements.

Assume further that ip has been initialized to point to an element in the first half of the array.

Write an expression whose value is the element in the array after the element that ip points to.

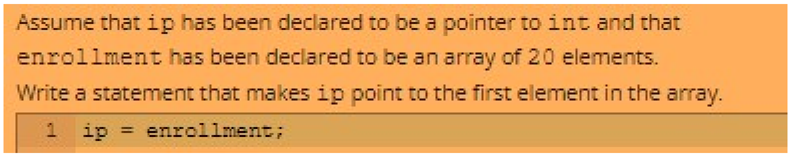
```
*(ip+1)
```

- 

Assume that ip has been declared to be a pointer to int and that result has been declared to be an array of 100 elements.

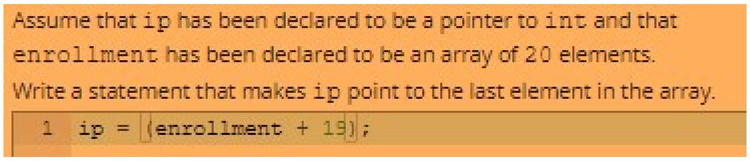
Assume further that ip has been initialized to point to an element in the first half of the array.
Write an expression whose value is the sum of the element that ip points to plus the next two elements.

```
*(ip)+*(ip+1)+*(ip+2)
```

- 

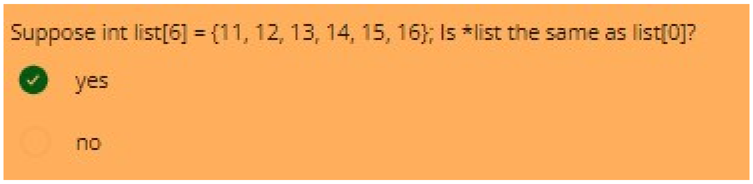
Assume that ip has been declared to be a pointer to int and that enrollment has been declared to be an array of 20 elements.
Write a statement that makes ip point to the first element in the array.

```
ip = enrollment;
```

- 

Assume that ip has been declared to be a pointer to int and that enrollment has been declared to be an array of 20 elements.
Write a statement that makes ip point to the last element in the array.

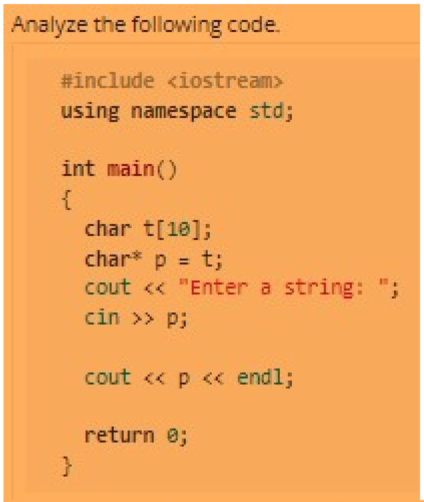
```
ip = (enrollment + 19);
```

- 

  Suppose int list[6] = {11, 12, 13, 14, 15, 16}; Is *list the same as list[0]?

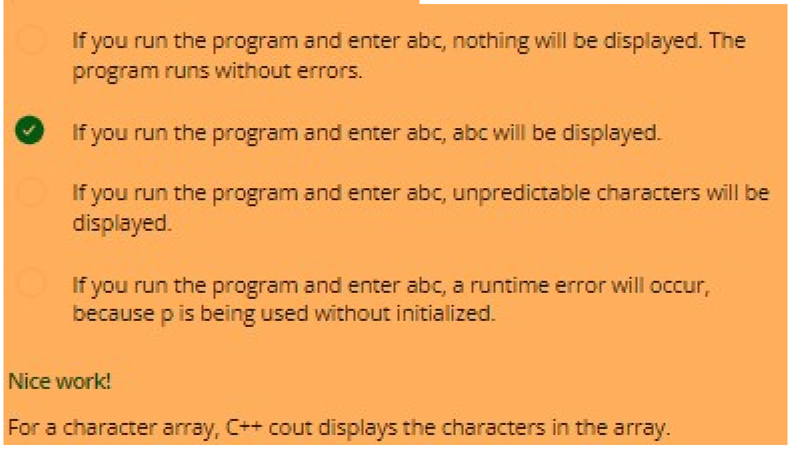
  ✓ yes

  ○ no

-
- 

  Analyze the following code.

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

  int main()
  {
    char t[10];
    char* p = t;
    cout << "Enter a string: ";
    cin >> p;

    cout << p << endl;

    return 0;
  }
  ```

- 

  ○ If you run the program and enter abc, nothing will be displayed. The program runs without errors.

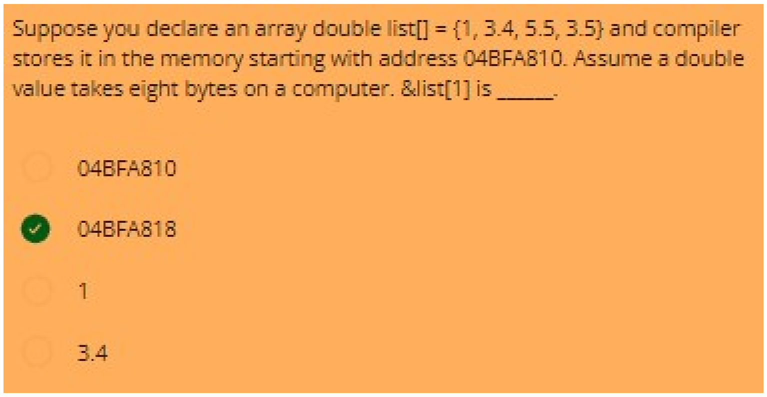
  ✓ If you run the program and enter abc, abc will be displayed.

  ○ If you run the program and enter abc, unpredictable characters will be displayed.

  ○ If you run the program and enter abc, a runtime error will occur, because p is being used without initialized.

  Nice work!

  For a character array, C++ cout displays the characters in the array.

-
- 

  Suppose you declare an array double list[] = {1, 3.4, 5.5, 3.5} and compiler stores it in the memory starting with address 04BFA810. Assume a double value takes eight bytes on a computer. &list[1] is ______.

  ○ 04BFA810

  ✓ 04BFA818

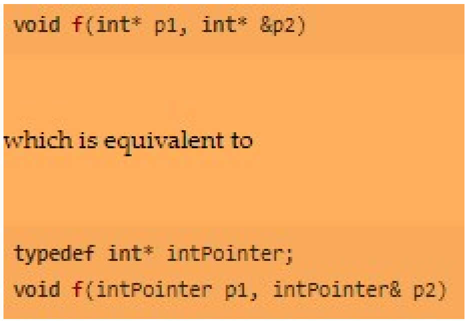
  ○ 1

  ○ 3.4

-
-

11.6 Passing Pointer Arguments in a Function Call

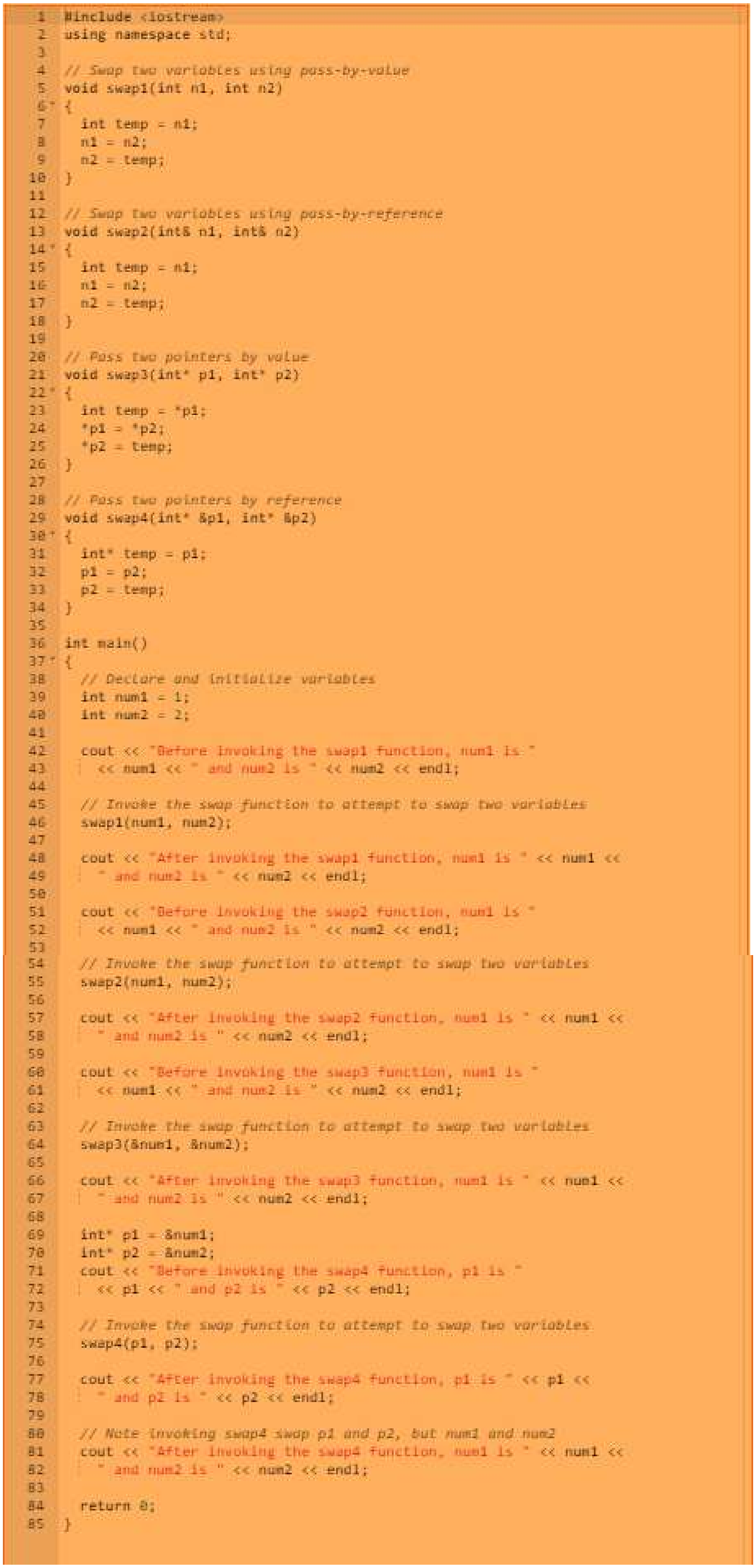
Tuesday, March 21, 2023 4:30 PM

- A C++ function may have pointer parameters
- already know 2 ways to pass args to a function in C++:
  - Pass by val
  - Pass by reference
- can also pass pointer args in a fn call, a pointer arg can be passed by val/ref
- Ex:
  - 

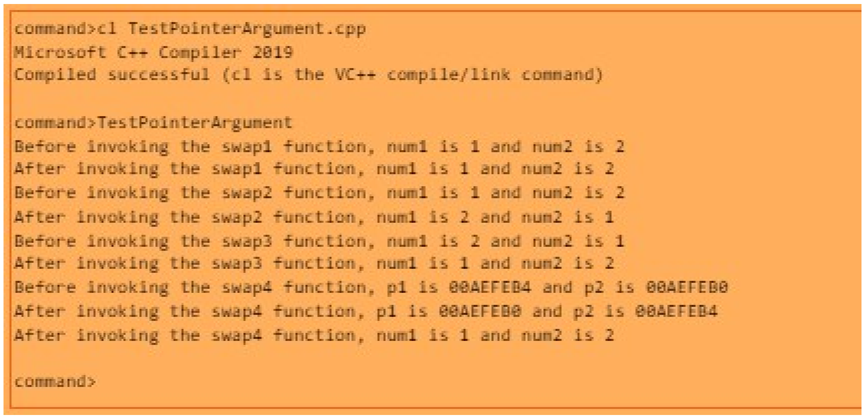
    ```
    void f(int* p1, int* &p2)
    ```

    which is equivalent to

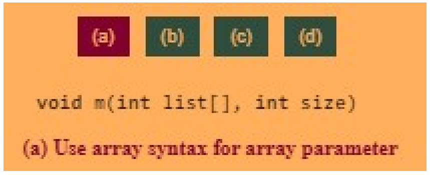
    ```
    typedef int* intPointer;
    void f(intPointer p1, intPointer& p2)
    ```
- What if invoke function f(q1, q2) w/ 2 pointers q1 and q2:
  - The pointer q1 is passed to p1 by val, so *p1 and *q1 point to the same content
  - If function f changes *p1 (like *p1 = 20), *q1 is changed too, but if function f changes p1 (like p1 = somePointerVar), q1 is not changed
  - Pointer q2 is passed to p2 by reference, so q2 and p2 are now aliases, basically the same.
  - If function f changes *p2(like *p2 = 20), *q2 is changed too
  - If function f changes p2 (like p2 = somePointerVar), q2 is changed to



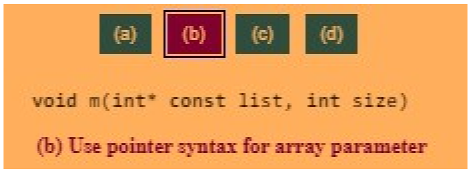
```cpp
#include <iostream>
using namespace std;

// Swap two variables using pass-by-value
void swap1(int n1, int n2)
{
  int temp = n1;
  n1 = n2;
  n2 = temp;
}

// Swap two variables using pass-by-reference
void swap2(int& n1, int& n2)
{
  int temp = n1;
  n1 = n2;
  n2 = temp;
}

// Pass two pointers by value
void swap3(int* p1, int* p2)
{
  int temp = *p1;
  *p1 = *p2;
  *p2 = temp;
}

// Pass two pointers by reference
void swap4(int* &p1, int* &p2)
{
  int* temp = p1;
  p1 = p2;
  p2 = temp;
}

int main()
{
  // Declare and initialize variables
  int num1 = 1;
  int num2 = 2;

  cout << "Before invoking the swap1 function, num1 is "
    << num1 << " and num2 is " << num2 << endl;

  // Invoke the swap function to attempt to swap two variables
  swap1(num1, num2);

  cout << "After invoking the swap1 function, num1 is " << num1 <<
    " and num2 is " << num2 << endl;

  cout << "Before invoking the swap2 function, num1 is "
    << num1 << " and num2 is " << num2 << endl;

  // Invoke the swap function to attempt to swap two variables
  swap2(num1, num2);

  cout << "After invoking the swap2 function, num1 is " << num1 <<
    " and num2 is " << num2 << endl;

  cout << "Before invoking the swap3 function, num1 is "
    << num1 << " and num2 is " << num2 << endl;

  // Invoke the swap function to attempt to swap two variables
  swap3(&num1, &num2);

  cout << "After invoking the swap3 function, num1 is " << num1 <<
    " and num2 is " << num2 << endl;

  int* p1 = &num1;
  int* p2 = &num2;
  cout << "Before invoking the swap4 function, p1 is "
    << p1 << " and p2 is " << p2 << endl;

  // Invoke the swap function to attempt to swap two variables
  swap4(p1, p2);

  cout << "After invoking the swap4 function, p1 is " << p1 <<
    " and p2 is " << p2 << endl;

  // Note invoking swap4 swap p1 and p2, but num1 and num2
  cout << "After invoking the swap4 function, num1 is " << num1 <<
    " and num2 is " << num2 << endl;

  return 0;
}
```

-
- 

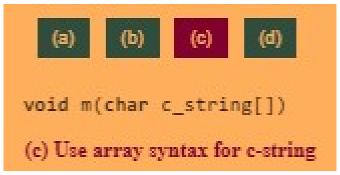
```
command>cl TestPointerArgument.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestPointerArgument
Before invoking the swap1 function, num1 is 1 and num2 is 2
After invoking the swap1 function, num1 is 1 and num2 is 2
Before invoking the swap2 function, num1 is 1 and num2 is 2
After invoking the swap2 function, num1 is 2 and num2 is 1
Before invoking the swap3 function, num1 is 2 and num2 is 1
After invoking the swap3 function, num1 is 1 and num2 is 2
Before invoking the swap4 function, p1 is 00AEFEB4 and p2 is 00AEFEB0
After invoking the swap4 function, p1 is 00AEFEB0 and p2 is 00AEFEB4
After invoking the swap4 function, num1 is 1 and num2 is 2

command>
```

- An array parameter in a function can always be replaced using a pointer parameter
  - A & b are equivalent, c & d are equivalent
  - 

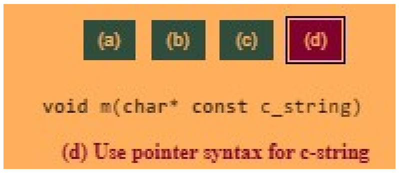
    ```
    void m(int list[], int size)
    ```

    (a) Use array syntax for array parameter
  -
  - 

    ```
    void m(int* const list, int size)
    ```

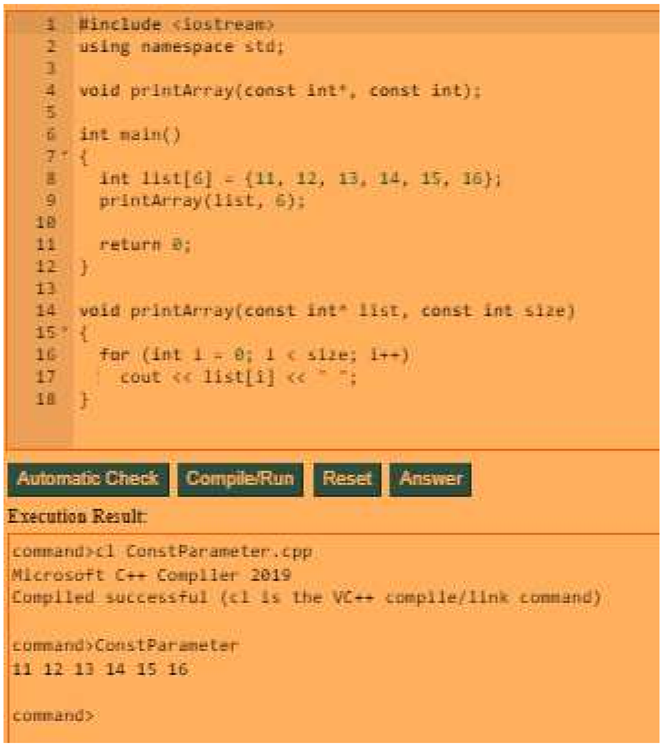
    (b) Use pointer syntax for array parameter
  -
  - 

    ```
    void m(char c_string[])
    ```

    (c) Use array syntax for c-string
  -
  - 

    ```
    void m(char* const c_string)
    ```

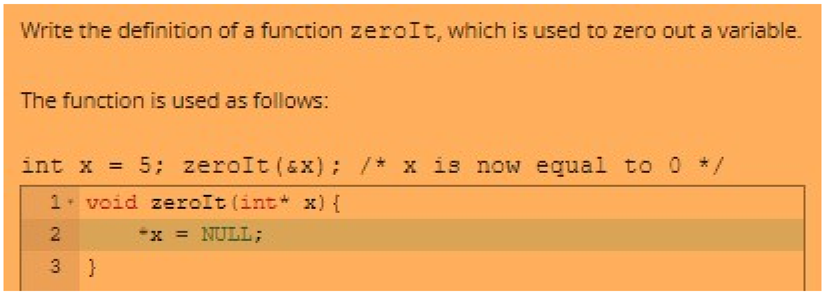
    (d) Use pointer syntax for c-string
  - Remember that C-string is an array of chars that ends w/ a null terminator, the size of a C-string can be detected from the C-string itself
- If val doesn't change, declare it const to prevent it from being accidentally changed



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

void printArray(const int*, const int);

int main()
{
  int list[6] = {11, 12, 13, 14, 15, 16};
  printArray(list, 6);

  return 0;
}

void printArray(const int* list, const int size)
{
  for (int i = 0; i < size; i++)
    cout << list[i] << " ";
}
```

Execution Result:

```
command>cl ConstParameter.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>ConstParameter
11 12 13 14 15 16

command>
```

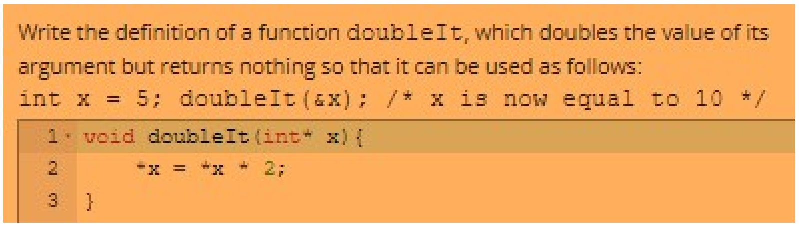


Write the definition of a function `zeroIt`, which is used to zero out a variable.

The function is used as follows:

```
int x = 5; zeroIt(&x); /* x is now equal to 0 */
```

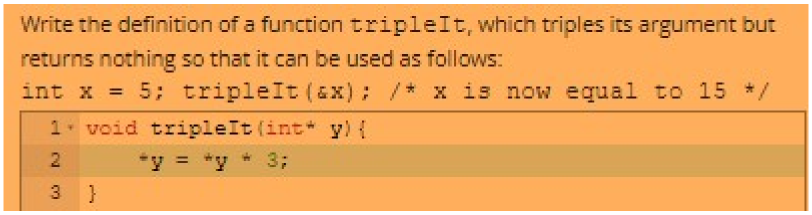
```
void zeroIt(int* x){
    *x = NULL;
}
```



Write the definition of a function `doubleIt`, which doubles the value of its argument but returns nothing so that it can be used as follows:

```
int x = 5; doubleIt(&x); /* x is now equal to 10 */
```

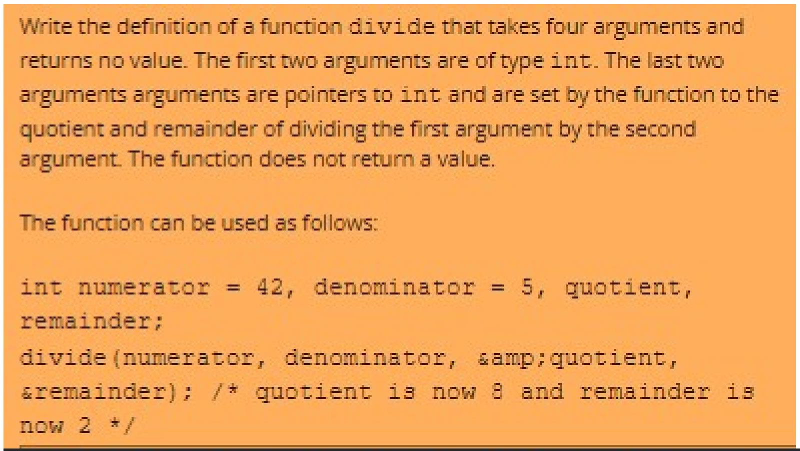
```
void doubleIt(int* x){
    *x = *x * 2;
}
```



Write the definition of a function `tripleIt`, which triples its argument but returns nothing so that it can be used as follows:

```
int x = 5; tripleIt(&x); /* x is now equal to 15 */
```

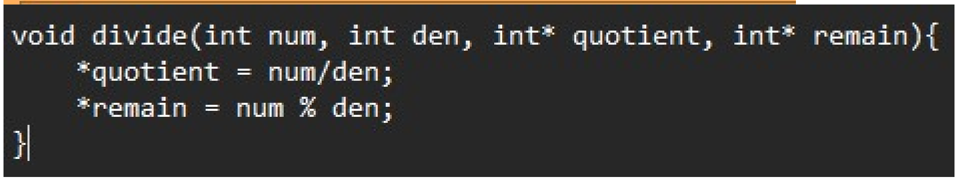
```
void tripleIt(int* y){
    *y = *y * 3;
}
```

- 

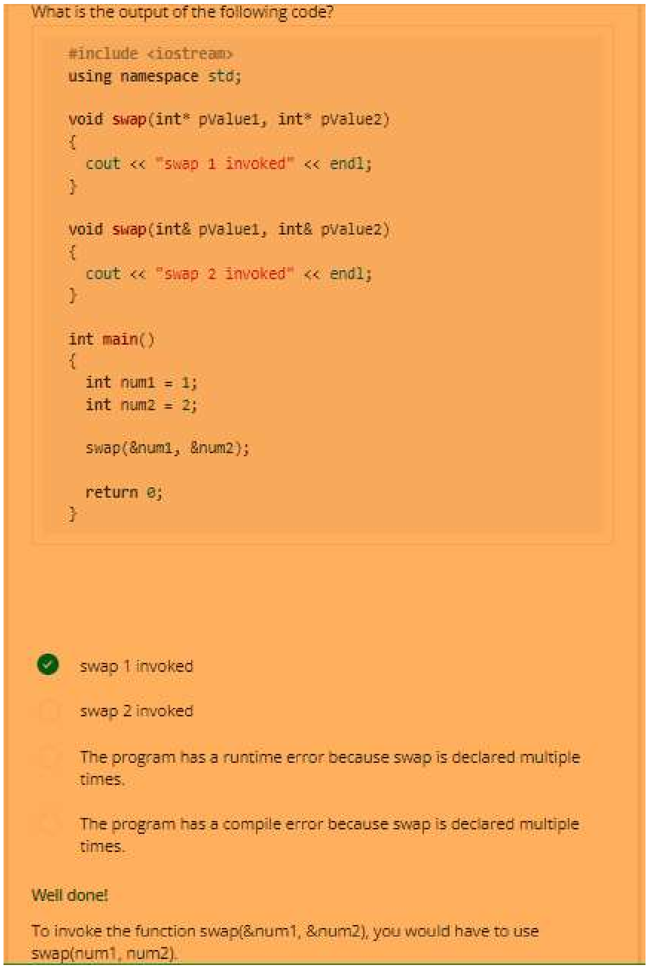
Write the definition of a function `divide` that takes four arguments and returns no value. The first two arguments are of type `int`. The last two arguments arguments are pointers to `int` and are set by the function to the quotient and remainder of dividing the first argument by the second argument. The function does not return a value.

The function can be used as follows:

```
int numerator = 42, denominator = 5, quotient,
remainder;
divide(numerator, denominator, &amp;quotient,
&remainder); /* quotient is now 8 and remainder is
now 2 */
```

- 

```
void divide(int num, int den, int* quotient, int* remain){
    *quotient = num/den;
    *remain = num % den;
}
```

-
- 

What is the output of the following code?

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

void swap(int* pValue1, int* pValue2)
{
  cout << "swap 1 invoked" << endl;
}

void swap(int& pValue1, int& pValue2)
{
  cout << "swap 2 invoked" << endl;
}

int main()
{
  int num1 = 1;
  int num2 = 2;

  swap(&num1, &num2);

  return 0;
}
```

✓ swap 1 invoked

○ swap 2 invoked

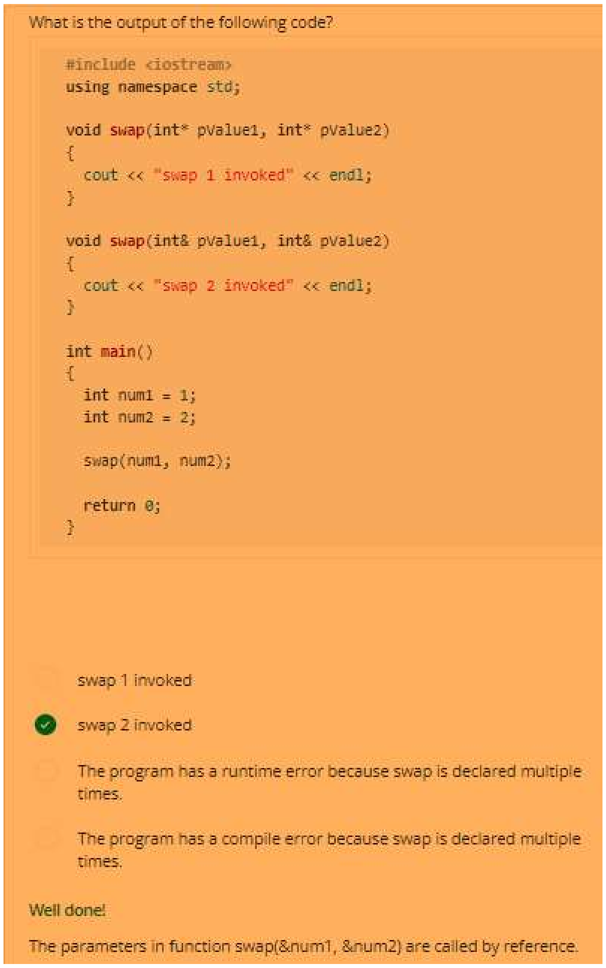
○ The program has a runtime error because swap is declared multiple times.

○ The program has a compile error because swap is declared multiple times.

Well done!

To invoke the function swap(&num1, &num2), you would have to use swap(num1, num2).

-

- 

  What is the output of the following code?

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

  void swap(int* pValue1, int* pValue2)
  {
    cout << "swap 1 invoked" << endl;
  }

  void swap(int& pValue1, int& pValue2)
  {
    cout << "swap 2 invoked" << endl;
  }

  int main()
  {
    int num1 = 1;
    int num2 = 2;

    swap(num1, num2);

    return 0;
  }
  ```

  ○ swap 1 invoked

  ✓ swap 2 invoked

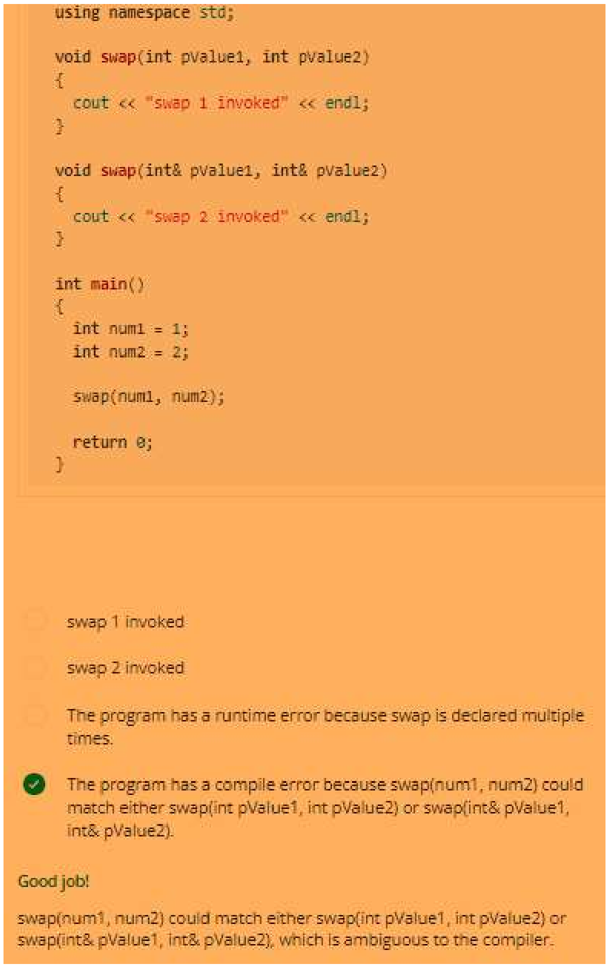
  ○ The program has a runtime error because swap is declared multiple times.

  ○ The program has a compile error because swap is declared multiple times.

  Well done!

  The parameters in function swap(&num1, &num2) are called by reference.

-



```
using namespace std;

void swap(int pValue1, int pValue2)
{
  cout << "swap 1 invoked" << endl;
}

void swap(int& pValue1, int& pValue2)
{
  cout << "swap 2 invoked" << endl;
}

int main()
{
  int num1 = 1;
  int num2 = 2;

  swap(num1, num2);

  return 0;
}
```

○ swap 1 invoked

○ swap 2 invoked

○ The program has a runtime error because swap is declared multiple times.

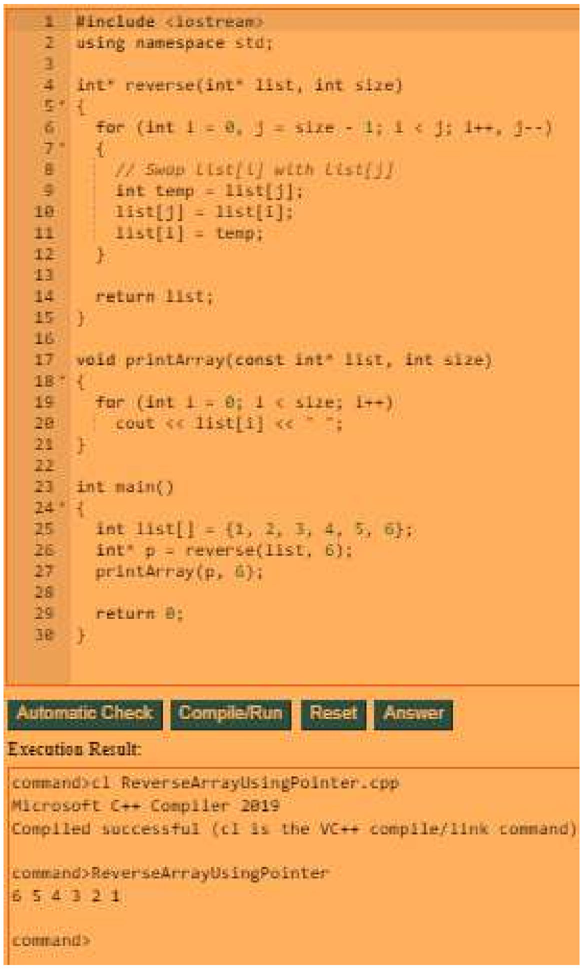
✓ The program has a compile error because swap(num1, num2) could match either swap(int pValue1, int pValue2) or swap(int& pValue1, int& pValue2).

Good job!

swap(num1, num2) could match either swap(int pValue1, int pValue2) or swap(int& pValue1, int& pValue2), which is ambiguous to the compiler.

11.7 Returning a Pointer from Functions

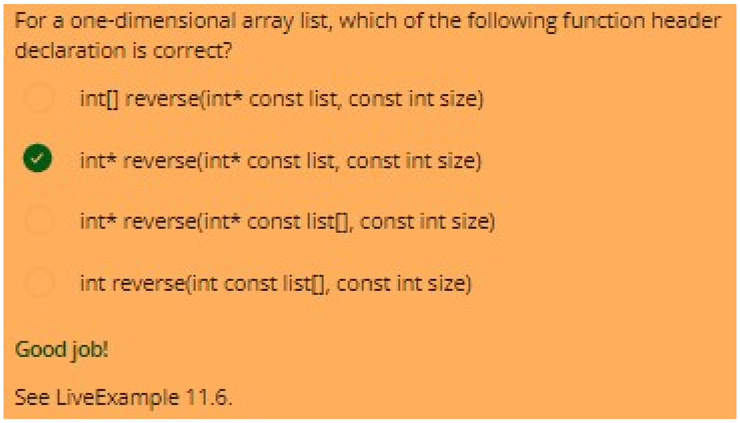
Tuesday, March 21, 2023 5:21 PM

- A C++ fn may return a pointer
-
- 

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

int* reverse(int* list, int size)
{
  for (int i = 0, j = size - 1; i < j; i++, j--)
  {
    // Swap list[i] with list[j]
    int temp = list[j];
    list[j] = list[i];
    list[i] = temp;
  }

  return list;
}

void printArray(const int* list, int size)
{
  for (int i = 0; i < size; i++)
    cout << list[i] << " ";
}

int main()
{
  int list[] = {1, 2, 3, 4, 5, 6};
  int* p = reverse(list, 6);
  printArray(p, 6);

  return 0;
}
```

Execution Result:

```
command>cl ReverseArrayUsingPointer.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>ReverseArrayUsingPointer
6 5 4 3 2 1

command>
```

- Reverse fn prototype is like this:
- `int* reverse(int* list, int size)`
- The return value type is an int pointer, it swaps the first element w/ the last, second w/ 2nd to last, ...
-
- 

For a one-dimensional array list, which of the following function header declaration is correct?

○ int[] reverse(int* const list, const int size)

✓ int* reverse(int* const list, const int size)

○ int* reverse(int* const list[], const int size)

○ int reverse(int const list[], const int size)

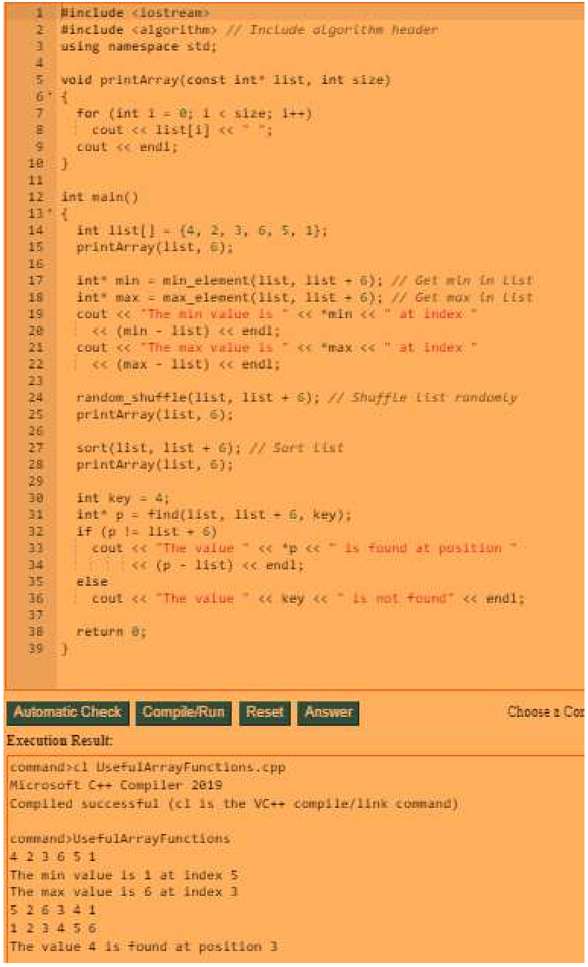
Good job!

See LiveExample 11.6.

-

11.8 Useful Array Functions

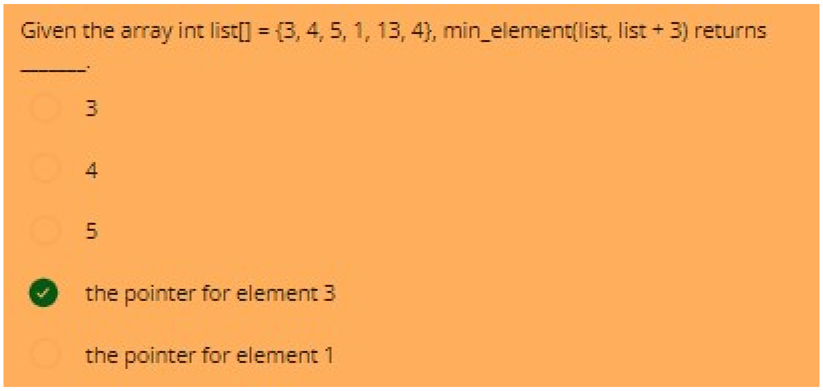
Tuesday, March 21, 2023 5:31 PM

- The min_element, max_element, sort, random_shuffle, and find fn can be used for arrays
- C++ gives many fn's to manipulate arrays
- Can use min_element and max_element fns to return the pointer to the min/max element of an array
- Can use sort to sort an array
- Can use random_shuffle fn to rando shuffle an array
- Can use find fn to find an element in an array
- All these fn use pointers in the arg and in return val
- 

```cpp
#include <iostream>
#include <algorithm> // Include algorithm header
using namespace std;

void printArray(const int* list, int size)
{
  for (int i = 0; i < size; i++)
    cout << list[i] << " ";
  cout << endl;
}

int main()
{
  int list[] = {4, 2, 3, 6, 5, 1};
  printArray(list, 6);

  int* min = min_element(list, list + 6); // Get min in list
  int* max = max_element(list, list + 6); // Get max in list
  cout << "The min value is " << *min << " at index "
    << (min - list) << endl;
  cout << "The max value is " << *max << " at index "
    << (max - list) << endl;

  random_shuffle(list, list + 6); // Shuffle list randomly
  printArray(list, 6);

  sort(list, list + 6); // Sort list
  printArray(list, 6);

  int key = 4;
  int* p = find(list, list + 6, key);
  if (p != list + 6)
    cout << "The value " << *p << " is found at position "
         << (p - list) << endl;
  else
    cout << "The value " << key << " is not found" << endl;

  return 0;
}
```

Execution Result:

```
command>cl UsefulArrayFunctions.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>UsefulArrayFunctions
4 2 3 6 5 1
The min value is 1 at index 5
The max value is 6 at index 3
5 2 6 3 4 1
1 2 3 4 5 6
The value 4 is found at position 3
```

-
- 

  Given the array int list[] = {3, 4, 5, 1, 13, 4}, min_element(list, list + 3) returns ______.

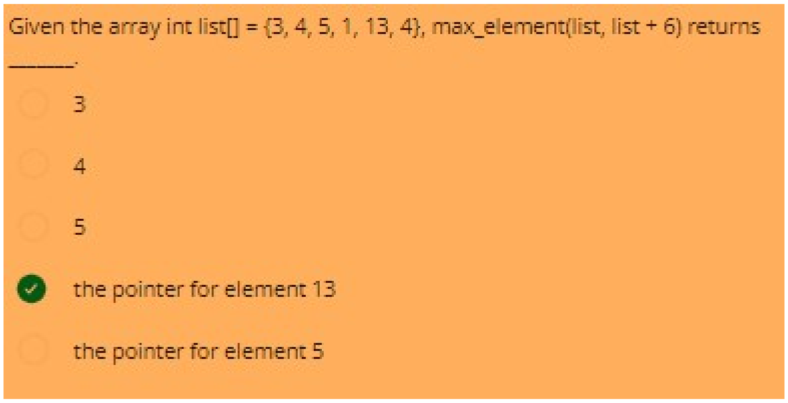
  ○ 3

  ○ 4

  ○ 5

  ✓ the pointer for element 3

  ○ the pointer for element 1
-

- 

Given the array int list[] = {3, 4, 5, 1, 13, 4}, max_element(list, list + 6) returns _______.

○ 3

○ 4

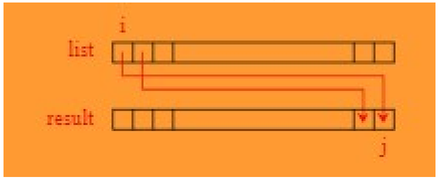
○ 5

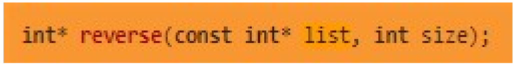
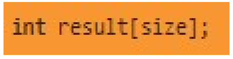
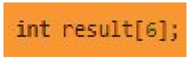
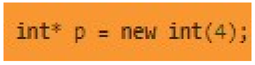
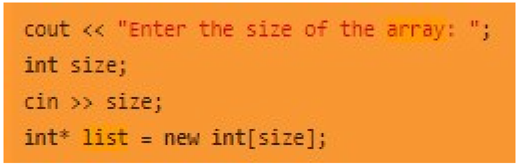
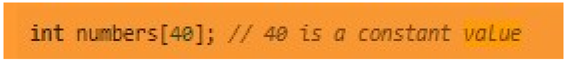
✓ the pointer for element 13

○ the pointer for element 5

11.9 Dynamic Persistent Memory Allocation

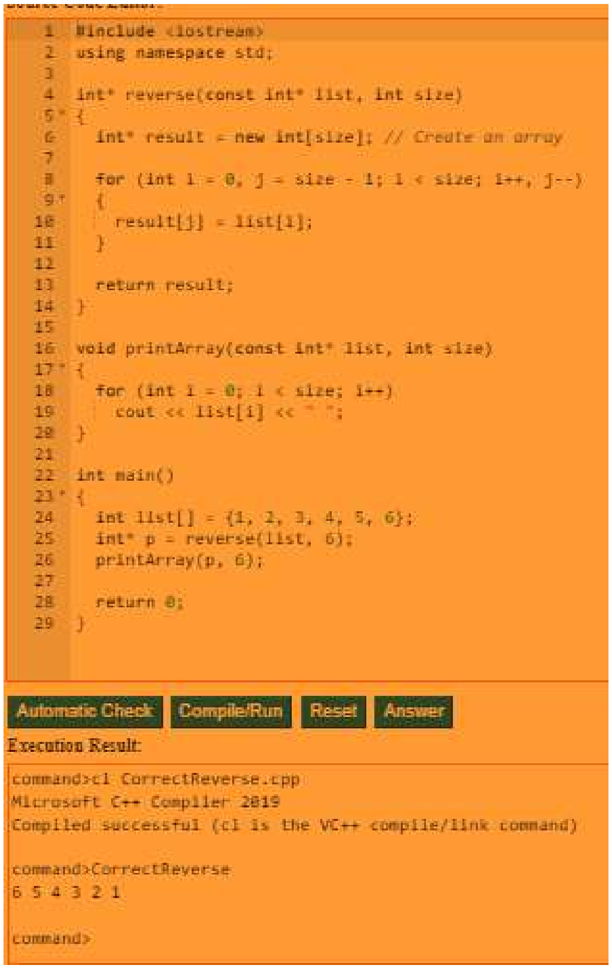
Tuesday, March 21, 2023 5:38 PM

- The **new** operator can be used to make persistent memory at runtime for primitive type values, arrays, and objects
- B4 there was example of fn that passes an array arg, reverses it, returns the array
- But if don't want to change the original array, can rewrite fn that passes an array arg and returns a new array that is reversal of the array arg
- An algorithm for function like this
  - Og array is list
  - Declare new array named result, same size as og array
  - Write loop to copy the first element, second, …, in og array into last element, 2nd last, …, in the new array, like diagram
  - 

  - Return result as pointer
- Fn prototype look like this:
  - 
```
int* reverse(const int* list, int size);
```
- Return val is an int pointer, but to declare new array, can do this
  - 
```
int result[size];
```
- But C++ don't let size be a var, but can overcome this by just assuming array is 6 (that solved everything, what a solution)
  - 
```
int result[6];
```
- The array result is stored in activation record in the call stack, mem in the call stack doesn't persist, when fn returns, the activation record used by the fn in the call stack are thrown away from the call stack
- Trying to access the array by the pointer will result in error and rando vals
- To fix, allocate persistent storage for result array so can be accessed after the fn returns
- C++ supports dynamic mem allocation, which lets allocate persistent storage dynamically
- Mem made using new operator, like
  - 
```
int* p = new int(4);
```
- In this, new int tells comp to allocate mem space for an int var initialized to 4 at runtime, and the address of the var is assigned to the pointer p, so can access the mem thru the pointer
- Can make an array dynamically, like
  - 
```
cout << "Enter the size of the array: ";
int size;
cin >> size;
int* list = new int[size];
```
- In this, int[size] tells pc to allocate mem space for an int array w/ the specified numb of elements, and the address of the array is assigned to list, the array made using the new operator is also known as a dynamic array
- When make a regular array, its size gotta be known at compile time, cant be var, must be constant
  - 
```
int numbers[40]; // 40 is a constant value
```
- When make dynamic array, size is determined at runtime, can be int var, like

- ○

  ```
  int* list = new int[size]; // size is a variable
  ```

- Mem allocated using the new operator is persistent and exists until its explicitly deleted/ the pgrm exits
- Pgrm that runs:
  - ○ 

    ```
     1  #include <iostream>
     2  using namespace std;
     3
     4  int* reverse(const int* list, int size)
     5  {
     6    int* result = new int[size]; // Create an array
     7
     8    for (int i = 0, j = size - 1; i < size; i++, j--)
     9    {
    10      result[j] = list[i];
    11    }
    12
    13    return result;
    14  }
    15
    16  void printArray(const int* list, int size)
    17  {
    18    for (int i = 0; i < size; i++)
    19      cout << list[i] << " ";
    20  }
    21
    22  int main()
    23  {
    24    int list[] = {1, 2, 3, 4, 5, 6};
    25    int* p = reverse(list, 6);
    26    printArray(p, 6);
    27
    28    return 0;
    29  }
    ```

    Execution Result:

    ```
    command>cl CorrectReverse.cpp
    Microsoft C++ Compiler 2019
    Compiled successful (cl is the VC++ compile/link command)

    command>CorrectReverse
    6 5 4 3 2 1

    command>
    ```

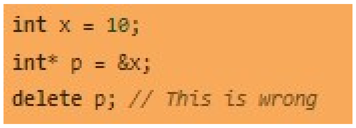
- The size can be a var when making an array using the new operator
- C++ allocates local vars in the stack, but mem allocated by the new operator is in an area of mem called the freestore/heap
  - ○ This mem remains available until explicitly free it/pgrm terminates
  - ○ In image^, fn ln 6, result array is made, then in ln 25, result array is intact, can access it in ln 26 to print all elements in the result array
  - ○ To explicitly free the mem made by the new operator, use delete operator for the pointer, like
  - ○

    ```
    delete p;
    ```

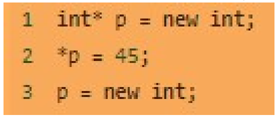
  - ○ The word delete is a keyword in C++, if mem allocated for an array, the [] symbol gotta be put btwn the delete keyword and the pointer to the array to release mem properly, lik
  - ○

    ```
    delete [] list;
    ```

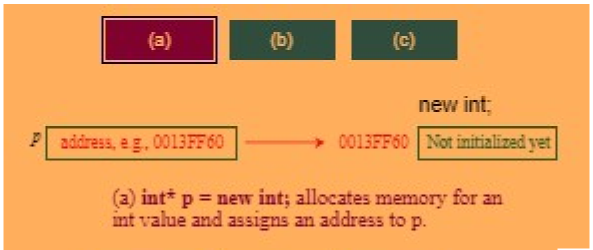
  - ○ C++ knows the array size bc size is stored in hep
- After mem pointed by a pointer is freed, the val of the pointer becomes undefined, and if some other pointer pointed to the same mem that was freed, then that pointer is also undefined. These undefined pointers are called dangling pointers
- Don't apply dereference operator * on dangling pointer, cause error
- Use the delete keyword only w/ the pointer that points to the mem created by the new operator, otherwise would cause unexpected probs.

- 

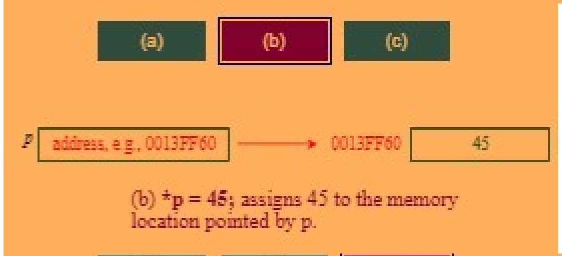
```
int x = 10;
int* p = &x;
delete p; // This is wrong
```

- Can also accidentally reassign a pointer b4 deleting the mem it pts to, like
- 

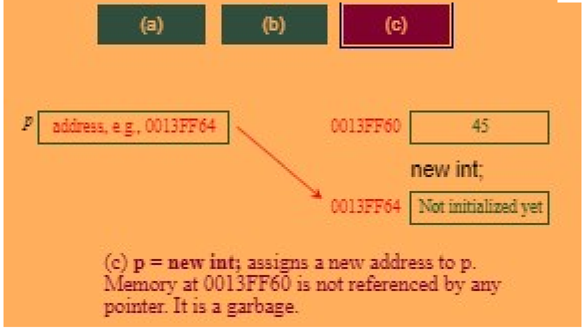
```
int* p = new int;
*p = 45;
p = new int;
```

    - ○
    - ○ 

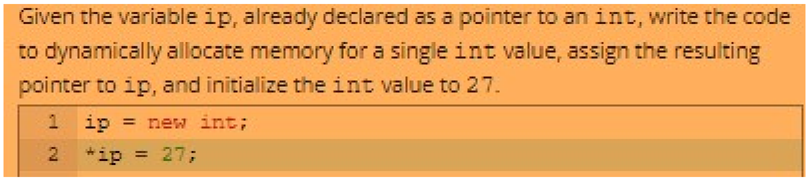
      (a) **int* p = new int;** allocates memory for an int value and assigns an address to p.

    - ○ 

      (b) ***p = 45;** assigns 45 to the memory location pointed by p.

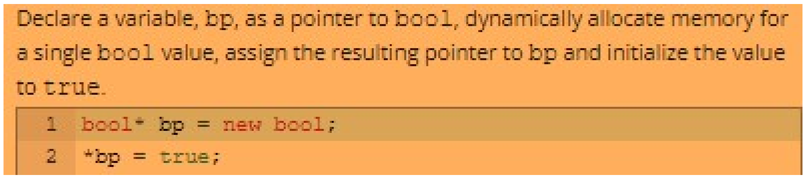
    - ○ 

      (c) **p = new int;** assigns a new address to p. Memory at 0013FF60 is not referenced by any pointer. It is a garbage.

- Dynamic mem allocation is pwrful, use carefully to avoid mem leak & errors
    - ○ Good practice is that every call to new should be matched by a call to delete
-
- 

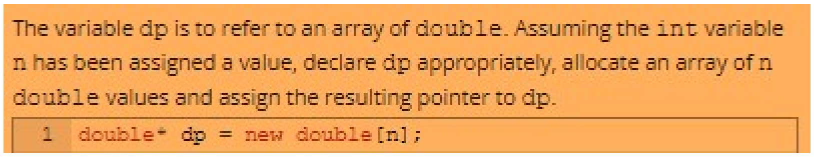
Given the variable ip, already declared as a pointer to an int, write the code to dynamically allocate memory for a single int value, assign the resulting pointer to ip, and initialize the int value to 27.

```
ip = new int;
*ip = 27;
```

-
- 

Declare a variable, bp, as a pointer to bool, dynamically allocate memory for a single bool value, assign the resulting pointer to bp and initialize the value to true.

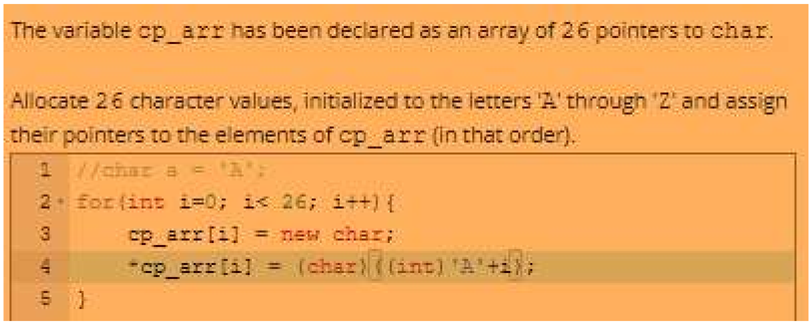
```
bool* bp = new bool;
*bp = true;
```

-
- 

The variable dp is to refer to an array of double. Assuming the int variable n has been assigned a value, declare dp appropriately, allocate an array of n double values and assign the resulting pointer to dp.

```
double* dp = new double[n];
```

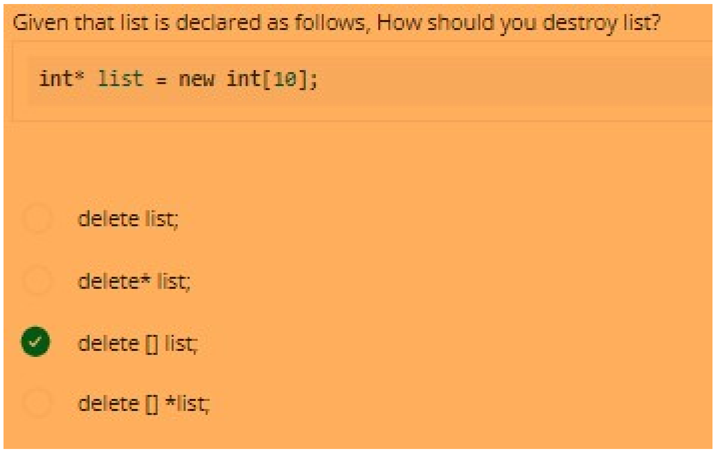
-



The variable `cp_arr` has been declared as an array of 26 pointers to `char`.

Allocate 26 character values, initialized to the letters 'A' through 'Z' and assign their pointers to the elements of `cp_arr` (in that order).

```
//char a = 'A';
for(int i=0; i< 26; i++){
    cp_arr[i] = new char;
    *cp_arr[i] = (char)((int)'A'+i);
}
```



Given that list is declared as follows, How should you destroy list?

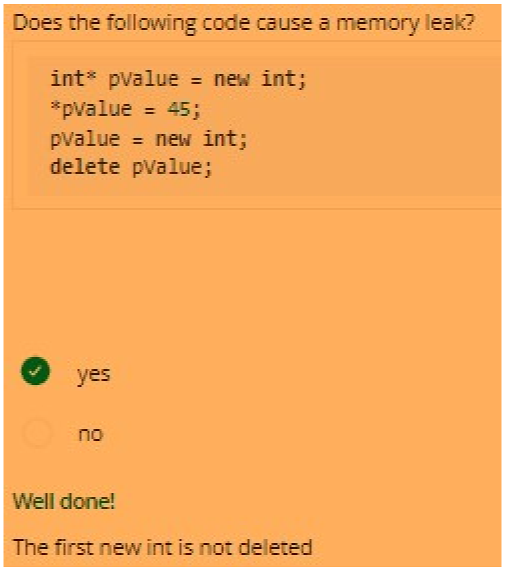
```
int* list = new int[10];
```

○ delete list;

○ delete* list;

✓ delete [] list;

○ delete [] *list;



Does the following code cause a memory leak?

```
int* pValue = new int;
*pValue = 45;
pValue = new int;
delete pValue;
```

✓ yes

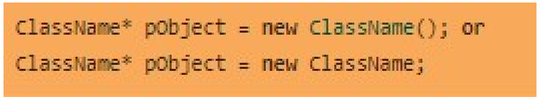
○ no

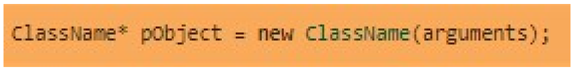
Well done!

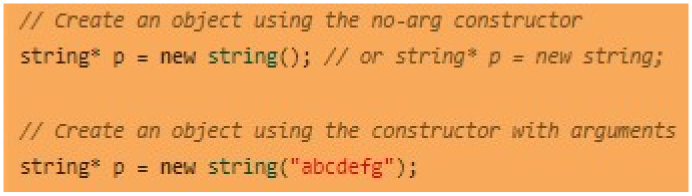
The first new int is not deleted

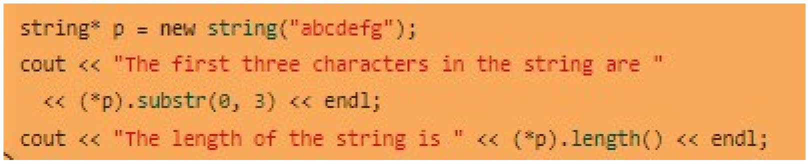
11.10 Creating and Accessing Dynamic Objects

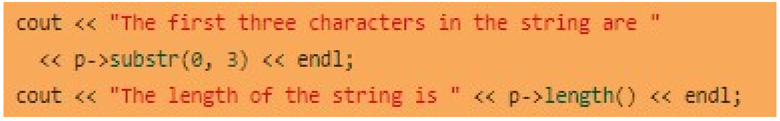
Thursday, March 23, 2023 10:36 AM

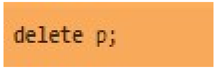
- To make an object dynamically, invoke the constructor for the object using the syntax new ClassName(args)
- Can also make objects dynamically on the hep using this syntax:
  - 

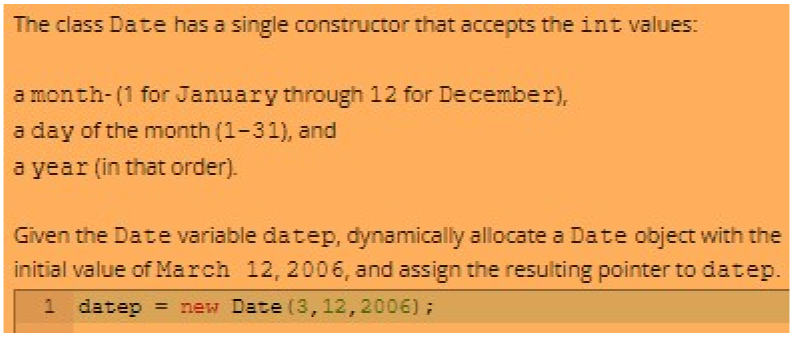
    ```
    ClassName* pObject = new ClassName(); or
    ClassName* pObject = new ClassName;
    ```
- They ^make an object using the no-arg constructor and assigns the object address to the pointer
-
- 

  ```
  ClassName* pObject = new ClassName(arguments);
  ```
- This^ makes an object using the constructor with arguments and assigns the object address to the pointer, like
  - 

    ```
    // Create an object using the no-arg constructor
    string* p = new string(); // or string* p = new string;

    // Create an object using the constructor with arguments
    string* p = new string("abcdefg");
    ```
- To access object membs by a pointer, dereference the pointer and use the dot (.) operator to object's members, like
  - 

    ```
    string* p = new string("abcdefg");
    cout << "The first three characters in the string are "
      << (*p).substr(0, 3) << endl;
    cout << "The length of the string is " << (*p).length() << endl;
    ```
- C++ also gives shorthand memb selection operator for accessing the object membs from a pointer: arrow operator(->), like
  - 

    ```
    cout << "The first three characters in the string are "
      << p->substr(0, 3) << endl;
    cout << "The length of the string is " << p->length() << endl;
    ```
- The objects are destroyed when pgrm terminated, but if want to destroy object b4 (good practice), do this
  - 

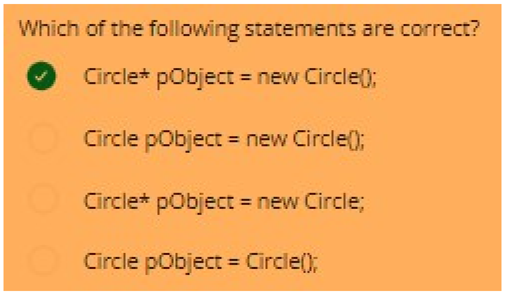
    ```
    delete p;
    ```
-
- 

  The class Date has a single constructor that accepts the int values:

  a month- (1 for January through 12 for December),
  a day of the month (1-31), and
  a year (in that order).

  Given the Date variable datep, dynamically allocate a Date object with the initial value of March 12, 2006, and assign the resulting pointer to datep.

  ```
  datep = new Date(3,12,2006);
  ```
-



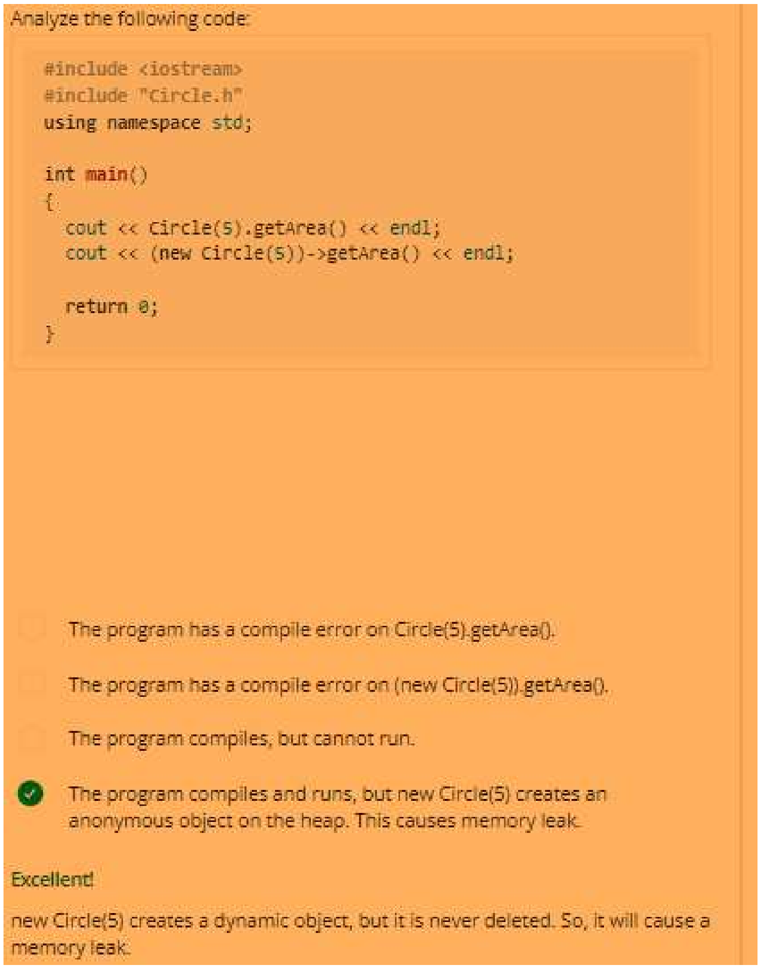
Which of the following statements are correct?

✓ Circle* pObject = new Circle();

○ Circle pObject = new Circle();

○ Circle* pObject = new Circle;

○ Circle pObject = Circle();



Analyze the following code:

```
#include <iostream>
#include "Circle.h"
using namespace std;

int main()
{
  cout << Circle(5).getArea() << endl;
  cout << (new Circle(5))->getArea() << endl;

  return 0;
}
```

□ The program has a compile error on Circle(5).getArea().

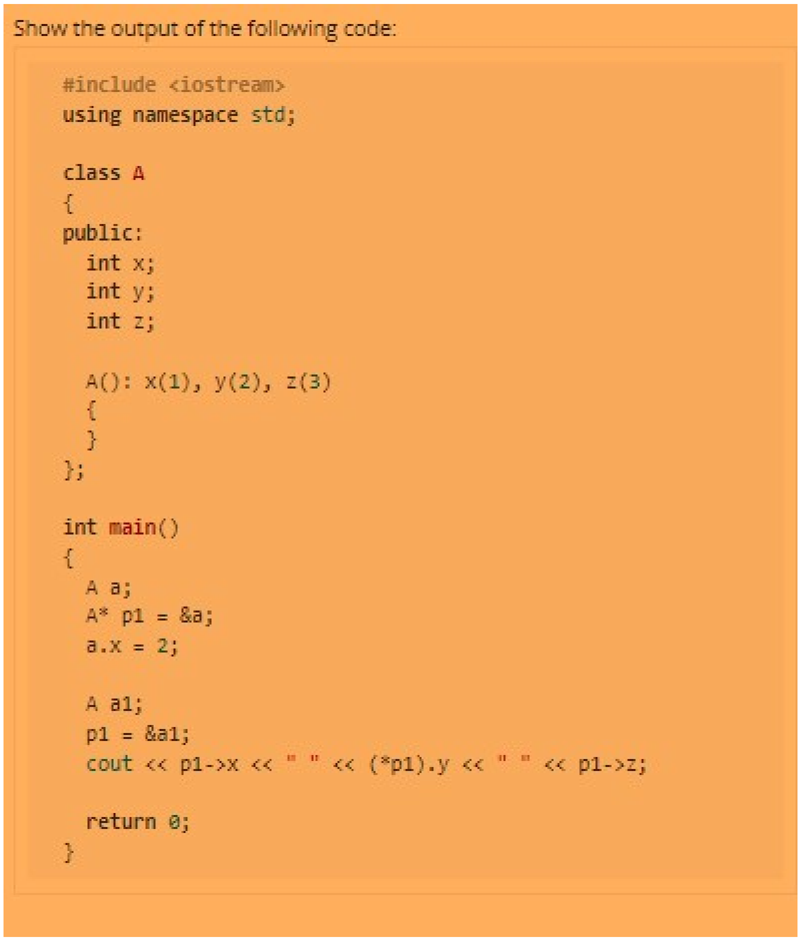
□ The program has a compile error on (new Circle(5)).getArea().

□ The program compiles, but cannot run.

✓ The program compiles and runs, but new Circle(5) creates an anonymous object on the heap. This causes memory leak.

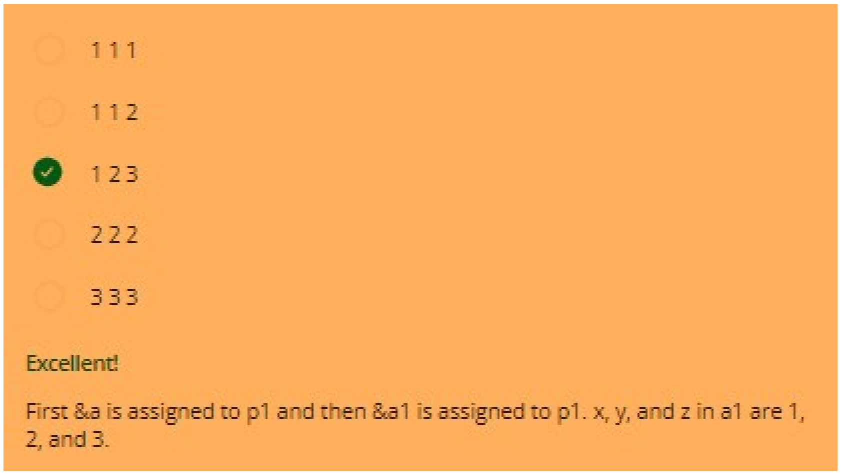
Excellent!

new Circle(5) creates a dynamic object, but it is never deleted. So, it will cause a memory leak.

- 

Show the output of the following code:

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

class A
{
public:
  int x;
  int y;
  int z;

  A(): x(1), y(2), z(3)
  {
  }
};

int main()
{
  A a;
  A* p1 = &a;
  a.x = 2;

  A a1;
  p1 = &a1;
  cout << p1->x << " " << (*p1).y << " " << p1->z;

  return 0;
}
```

-
- 

○ 1 1 1

○ 1 1 2

✓ 1 2 3

○ 2 2 2

○ 3 3 3

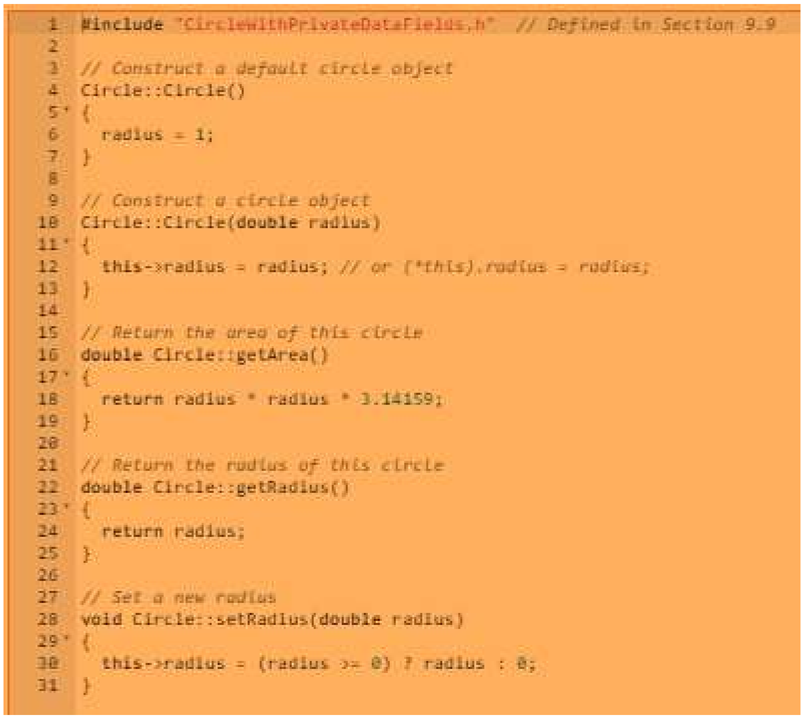
Excellent!

First &a is assigned to p1 and then &a1 is assigned to p1. x, y, and z in a1 are 1, 2, and 3.

-
-

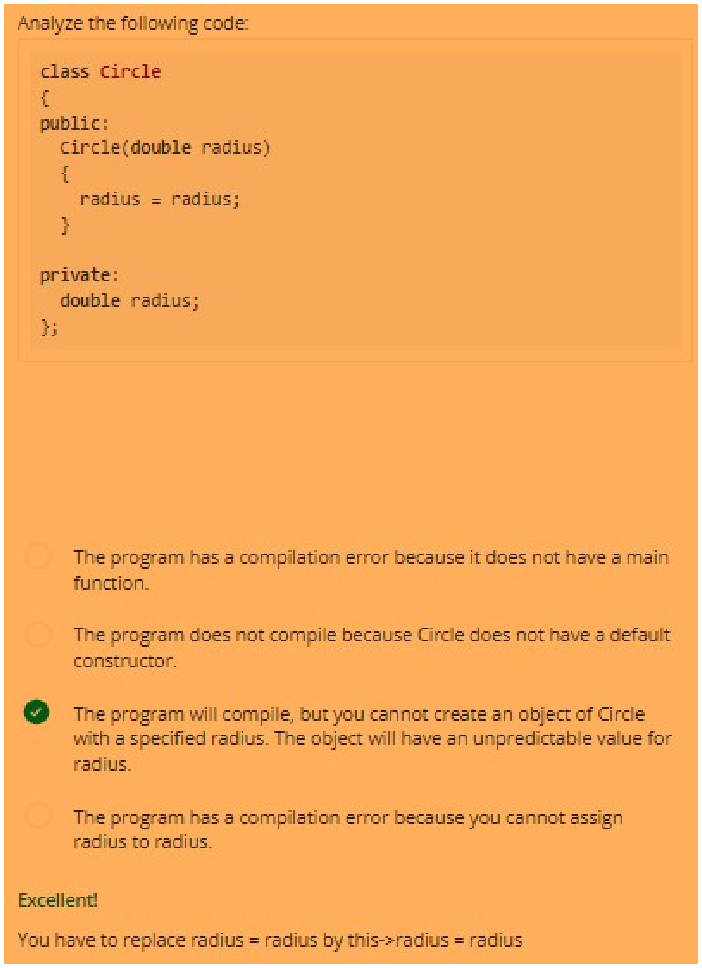
11.11 The this Pointer

Thursday, March 23, 2023 3:54 PM

- The this pointer points to the calling object itself
- Usually in the setter fn, the parameter name is the same as the data field nam, so data field becomes hidden in the fn
- Can use this keywork to reference a hidden data field in the fn, this is a special built in pointer that references the calling object
- See how work:
- 

```
 1 #include "CircleWithPrivateDataFields.h"  // Defined in Section 9.9
 2
 3 // Construct a default circle object
 4 Circle::Circle()
 5 {
 6   radius = 1;
 7 }
 8
 9 // Construct a circle object
10 Circle::Circle(double radius)
11 {
12   this->radius = radius; // or (*this).radius = radius;
13 }
14
15 // Return the area of this circle
16 double Circle::getArea()
17 {
18   return radius * radius * 3.14159;
19 }
20
21 // Return the radius of this circle
22 double Circle::getRadius()
23 {
24   return radius;
25 }
26
27 // Set a new radius
28 void Circle::setRadius(double radius)
29 {
30   this->radius = (radius >= 0) ? radius : 0;
31 }
```

- The parameter radius in the constructor (ln 30) is a local var
- To reference the data field radius in the object, have to use this->radius (ln 12)
- The parameter name radius in the setRadius fn (ln 28) is a local var, to reference the data field radius in the object, have to use this->radius (ln 30)



Analyze the following code:

```
class Circle
{
public:
  Circle(double radius)
  {
    radius = radius;
  }

private:
  double radius;
};
```

○ The program has a compilation error because it does not have a main function.

○ The program does not compile because Circle does not have a default constructor.

✓ The program will compile, but you cannot create an object of Circle with a specified radius. The object will have an unpredictable value for radius.

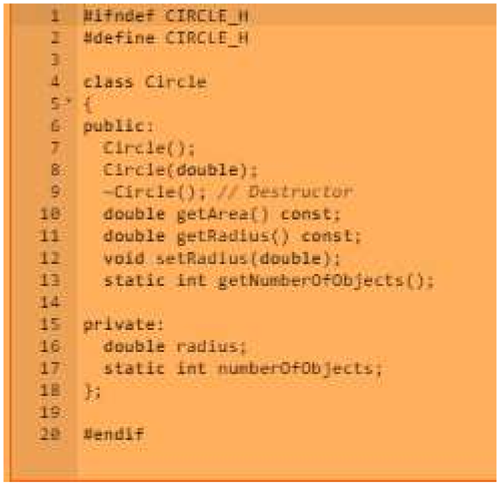
○ The program has a compilation error because you cannot assign radius to radius.

Excellent!

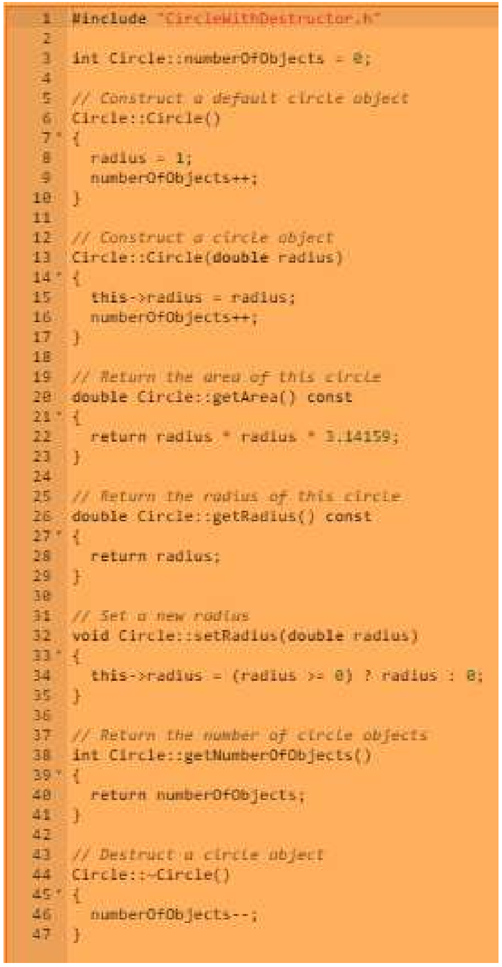
You have to replace radius = radius by this->radius = radius

11.12 Destructors

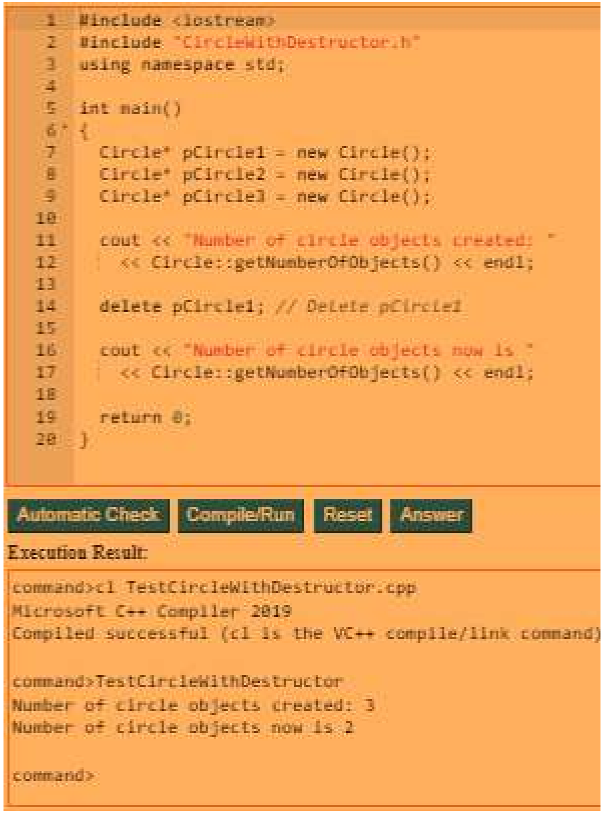
Thursday, March 23, 2023 4:06 PM

- Every class has a destructor, which is called auto when object is deleted
- Destructors are opp of constructors
- Constructor is invoked when an object is made and a destructor auto invoked when object is destroyed
- Every class has a default destructor if the destructor isn't explicitly defined
- Sometimes better to implement destructors to do customized ops
- Destructors named same as constructors, but must put tilde char (~) in front
- 

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle
{
public:
  Circle();
  Circle(double);
  ~Circle(); // Destructor
  double getArea() const;
  double getRadius() const;
  void setRadius(double);
  static int getNumberOfObjects();

private:
  double radius;
  static int numberOfObjects;
};

#endif
```

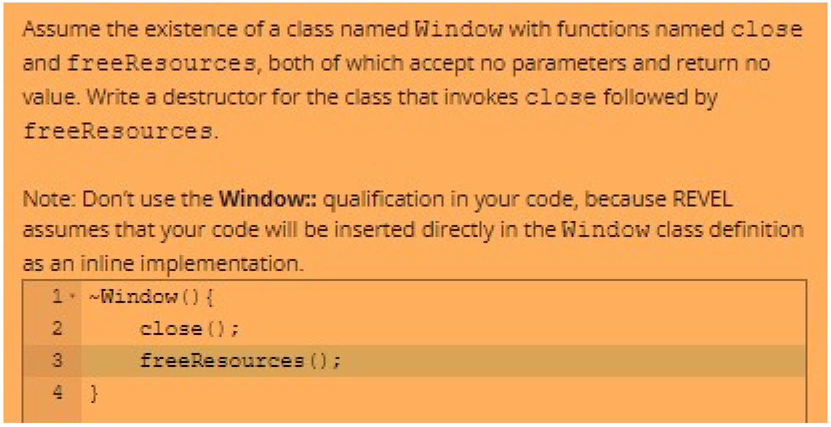
- Destructors have no return type and no args
- 

```
#include "CircleWithDestructor.h"

int Circle::numberOfObjects = 0;

// Construct a default circle object
Circle::Circle()
{
  radius = 1;
  numberOfObjects++;
}

// Construct a circle object
Circle::Circle(double radius)
{
  this->radius = radius;
  numberOfObjects++;
}

// Return the area of this circle
double Circle::getArea() const
{
  return radius * radius * 3.14159;
}

// Return the radius of this circle
double Circle::getRadius() const
{
  return radius;
}

// Set a new radius
void Circle::setRadius(double radius)
{
  this->radius = (radius >= 0) ? radius : 0;
}

// Return the number of circle objects
int Circle::getNumberOfObjects()
{
  return numberOfObjects;
}

// Destruct a circle object
Circle::~Circle()
{
  numberOfObjects--;
}
```

- Implementation similar to smth b4, but destructor implemented to decrement numberOfObjects
- 

```
 1  #include <iostream>
 2  #include "CircleWithDestructor.h"
 3  using namespace std;
 4
 5  int main()
 6  {
 7    Circle* pCircle1 = new Circle();
 8    Circle* pCircle2 = new Circle();
 9    Circle* pCircle3 = new Circle();
10
11    cout << "Number of circle objects created: "
12      << Circle::getNumberOfObjects() << endl;
13
14    delete pCircle1; // Delete pCircle1
15
16    cout << "Number of circle objects now is "
17      << Circle::getNumberOfObjects() << endl;
18
19    return 0;
20  }
```

```
command>cl TestCircleWithDestructor.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestCircleWithDestructor
Number of circle objects created: 3
Number of circle objects now is 2

command>
```

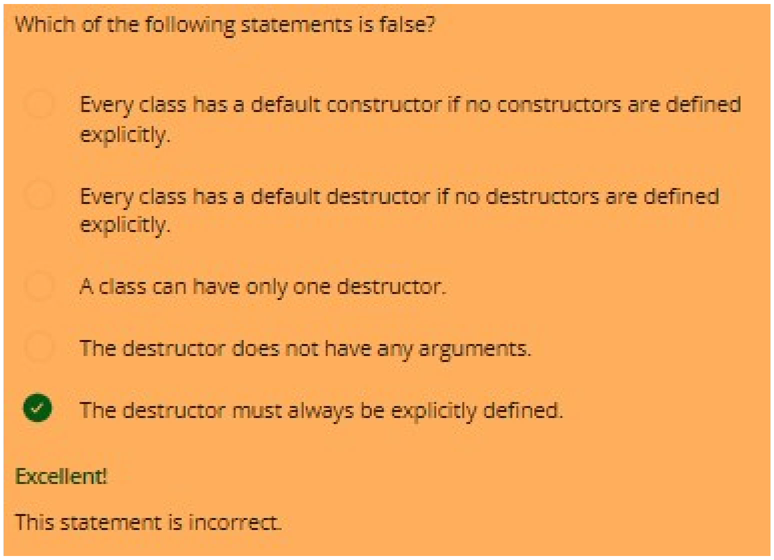
- Pgrm makes 3 Circle objects using new operator (ln 7-9), then numberOfObjects becomes 3
- Pgrm deletes Circle object (ln 14), so numberOfObjects becomes 2
- Destructors useful for deleting mem and other resources dynamically allocated by the object
- 

Assume the existence of a class named `Window` with functions named `close` and `freeResources`, both of which accept no parameters and return no value. Write a destructor for the class that invokes `close` followed by `freeResources`.

Note: Don't use the **Window::** qualification in your code, because REVEL assumes that your code will be inserted directly in the `Window` class definition as an inline implementation.

```
~Window(){
    close();
    freeResources();
}
```

-



Which of the following statements is false?

○ Every class has a default constructor if no constructors are defined explicitly.

○ Every class has a default destructor if no destructors are defined explicitly.

○ A class can have only one destructor.

○ The destructor does not have any arguments.

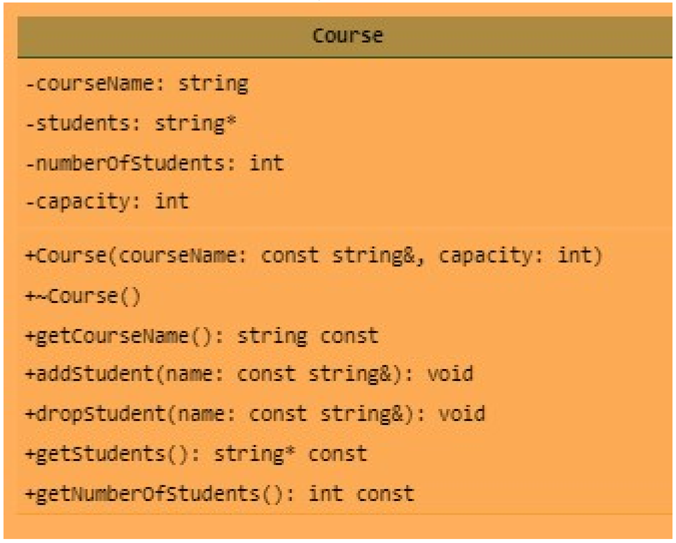
✓ The destructor must always be explicitly defined.

Excellent!

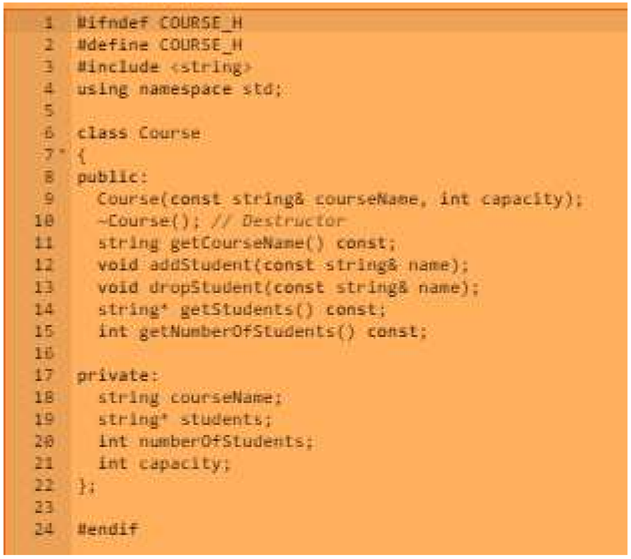
This statement is incorrect.

11.13 Case Study: The Couse Class

Thursday, March 23, 2023 4:31 PM

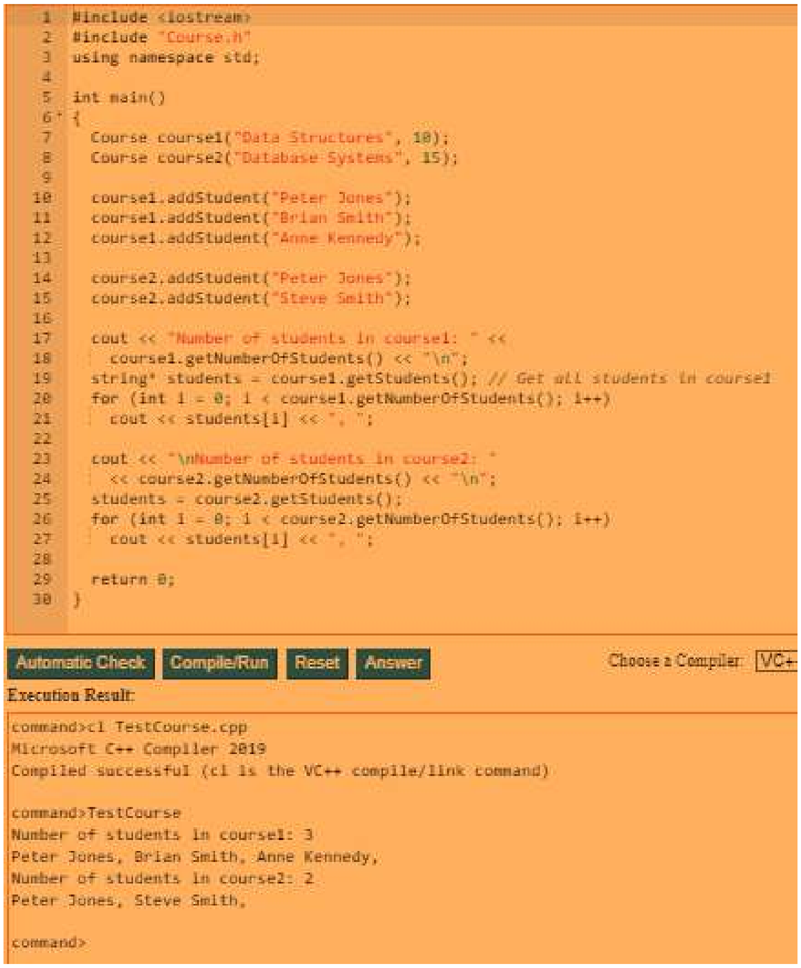
- Class for modeling courses
- Scenario: Need to process course info
- Each course has name, numb of students who take course
- Should be able to add/drop student to/from the course
- 

```
Course
-courseName: string
-students: string*
-numberOfStudents: int
-capacity: int
+Course(courseName: const string&, capacity: int)
+~Course()
+getCourseName(): string const
+addStudent(name: const string&): void
+dropStudent(name: const string&): void
+getStudents(): string* const
+getNumberOfStudents(): int const
```

- A Course object can be made using the constructor Course (string courseName, int capacity) by passing a course name and the max number of student allowed
- Can add a student to the course using the addStudent(string name) fn,
- Can drop a student... using the dropStudent(string name) fn,
- Can return all the students for the course using the getStudent() fn
- 

```
#ifndef COURSE_H
#define COURSE_H
#include <string>
using namespace std;

class Course
{
public:
  Course(const string& courseName, int capacity);
  ~Course(); // Destructor
  string getCourseName() const;
  void addStudent(const string& name);
  void dropStudent(const string& name);
  string* getStudents() const;
  int getNumberOfStudents() const;

private:
  string courseName;
  string* students;
  int numberOfStudents;
  int capacity;
};

#endif
```

-

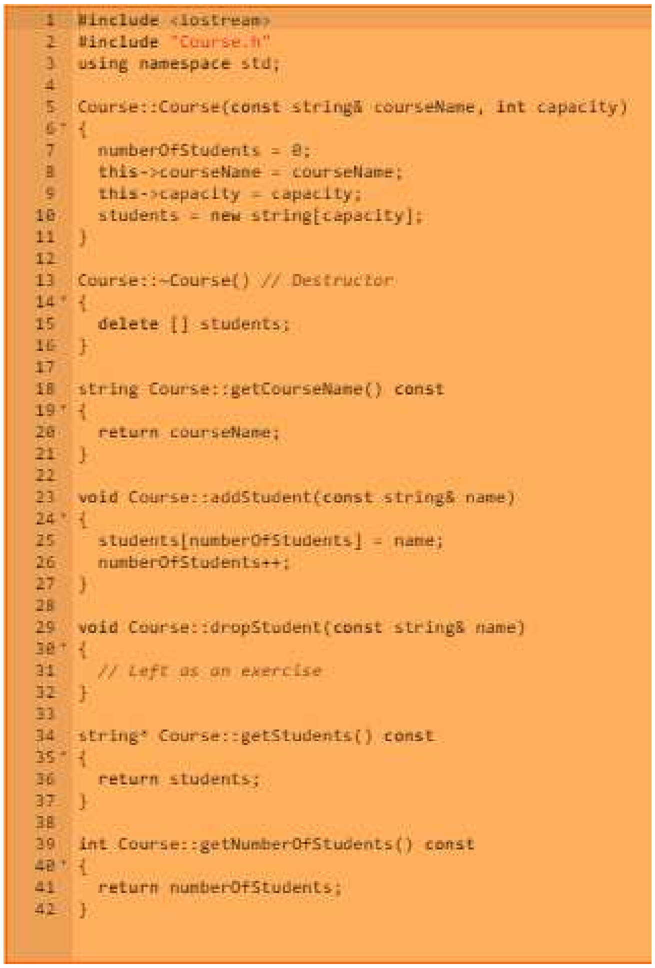
- 

```cpp
#include <iostream>
#include "Course.h"
using namespace std;

int main()
{
  Course course1("Data Structures", 10);
  Course course2("Database Systems", 15);

  course1.addStudent("Peter Jones");
  course1.addStudent("Brian Smith");
  course1.addStudent("Anne Kennedy");

  course2.addStudent("Peter Jones");
  course2.addStudent("Steve Smith");

  cout << "Number of students in course1: " <<
    course1.getNumberOfStudents() << "\n";
  string* students = course1.getStudents(); // Get all students in course1
  for (int i = 0; i < course1.getNumberOfStudents(); i++)
    cout << students[i] << ", ";

  cout << "\nNumber of students in course2: "
    << course2.getNumberOfStudents() << "\n";
  students = course2.getStudents();
  for (int i = 0; i < course2.getNumberOfStudents(); i++)
    cout << students[i] << ", ";

  return 0;
}
```

Execution Result:

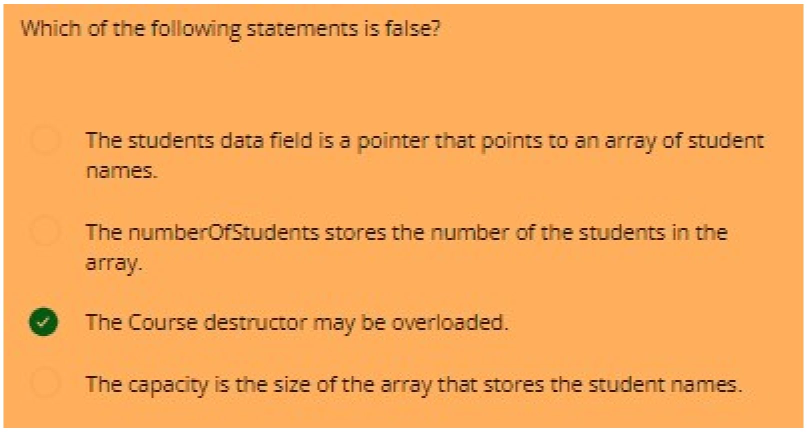
```
command>cl TestCourse.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>TestCourse
Number of students in course1: 3
Peter Jones, Brian Smith, Anne Kennedy,
Number of students in course2: 2
Peter Jones, Steve Smith,

command>
```

-

- 

```
1  #include <iostream>
2  #include "Course.h"
3  using namespace std;
4
5  Course::Course(const string& courseName, int capacity)
6  {
7    numberOfStudents = 0;
8    this->courseName = courseName;
9    this->capacity = capacity;
10   students = new string[capacity];
11 }
12
13 Course::~Course() // Destructor
14 {
15   delete [] students;
16 }
17
18 string Course::getCourseName() const
19 {
20   return courseName;
21 }
22
23 void Course::addStudent(const string& name)
24 {
25   students[numberOfStudents] = name;
26   numberOfStudents++;
27 }
28
29 void Course::dropStudent(const string& name)
30 {
31   // Left as an exercise
32 }
33
34 string* Course::getStudents() const
35 {
36   return students;
37 }
38
39 int Course::getNumberOfStudents() const
40 {
41   return numberOfStudents;
42 }
```

- Course constructor initializes numberOfStudents to 0 (ln 7), sets a new course name (ln 8), sets a capacity (ln 9), and makes a dynamic array (ln 10)
- The Course class uses an array to store the students for the course, array is made when a Course object is constructed, the array size is the max numb of students allowed for the course, so the array is made using new string[capacity]
- When a Course object is destroyed, destructor is invoked to destroy array properly
- The addStudent fn adds a student to the array, fn doesn't need to check numb of students in the class exceeds max capacity
- The getStudents fn returns address of the array for storing the students
- The dropStudent fn removes a student from the array, implemetation of this fn up to us
- User can make a Course and manipulate it thru public fn addStudent, dropStudent, getNumberOfStudents, and getStudents
- But use doesn't know how these fn are implemented, Course encapsulates internal implementation
- When make a Course object, an array of string is made, each element has a defualt string val made by the string class's no-arg constructor
- Caution- u should customize a destructor if the class has a pointer data field that points to dynamically made mem, otherwise, the pgrm can have a mem leak
-

- 

  Which of the following statements is false?

  ○ The students data field is a pointer that points to an array of student names.

  ○ The numberOfStudents stores the number of the students in the array.

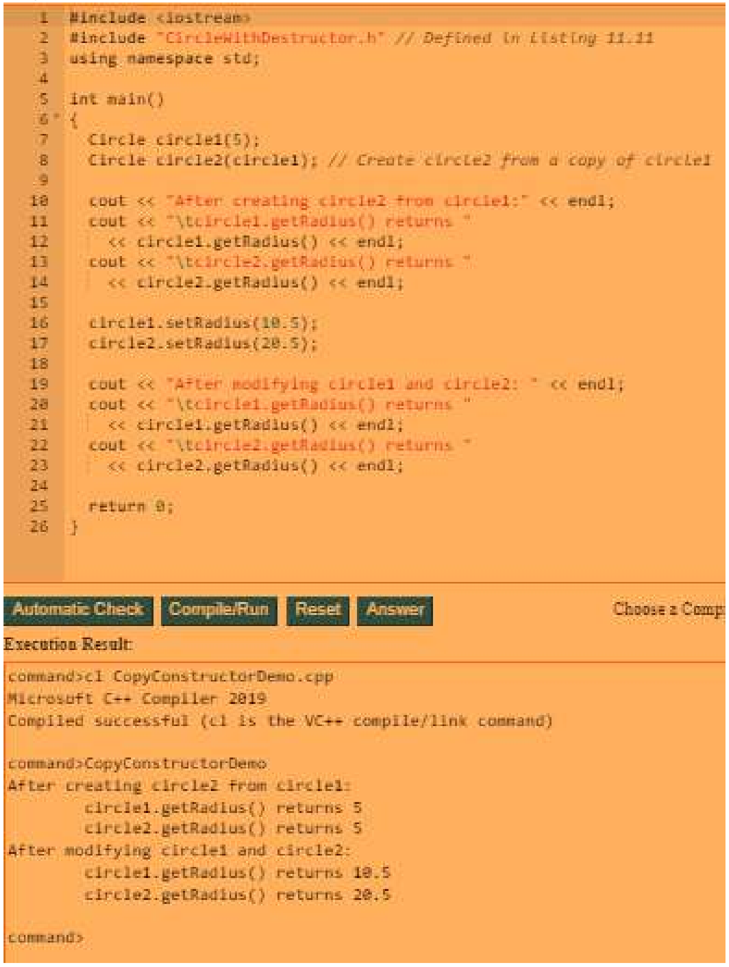
  ✓ The Course destructor may be overloaded.

  ○ The capacity is the size of the array that stores the student names.

-

11.14 Copy Constructors

Thursday, March 23, 2023 4:52 PM

- Every class has a copy constructor, used to copy objects
- Each class can define many overloaded constructors and one destructor
- Along w/ this, every class has a copy constructor, which can be used to make an object initialized with the data of another object of the same class.
- Signatur of the copy constructor is
- `ClassName(const ClassName&)`
- Ex:
  - Copy constructor for the Circle class is
  - `Circle(const Circle&)`
- A default copy constructor is given for each class implicitly, if not defined explicitly
- Default copy constructor just copies each data field in one object to its counterpart in other object, like
- 

```cpp
#include <iostream>
#include "CircleWithDestructor.h" // Defined in Listing 11.11
using namespace std;

int main()
{
  Circle circle1(5);
  Circle circle2(circle1); // Create circle2 from a copy of circle1

  cout << "After creating circle2 from circle1:" << endl;
  cout << "\tcircle1.getRadius() returns "
    << circle1.getRadius() << endl;
  cout << "\tcircle2.getRadius() returns "
    << circle2.getRadius() << endl;

  circle1.setRadius(10.5);
  circle2.setRadius(20.5);

  cout << "After modifying circle1 and circle2: " << endl;
  cout << "\tcircle1.getRadius() returns "
    << circle1.getRadius() << endl;
  cout << "\tcircle2.getRadius() returns "
    << circle2.getRadius() << endl;

  return 0;
}
```

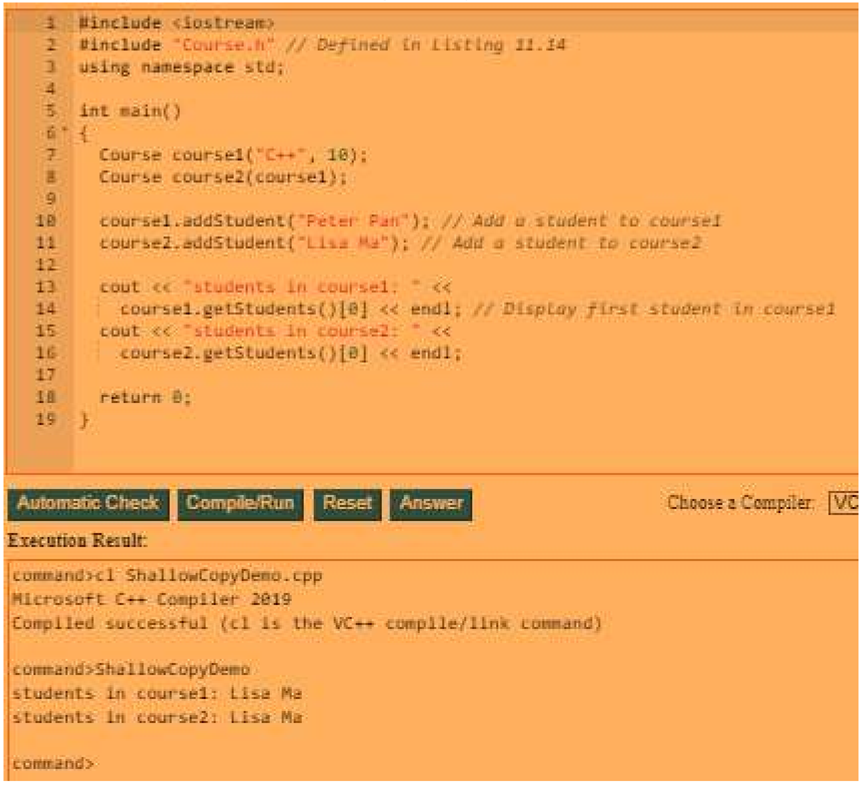
Execution Result:

```
command>cl CopyConstructorDemo.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>CopyConstructorDemo
After creating circle2 from circle1:
        circle1.getRadius() returns 5
        circle2.getRadius() returns 5
After modifying circle1 and circle2:
        circle1.getRadius() returns 10.5
        circle2.getRadius() returns 20.5

command>
```

- The pgrm makes 2 Circle objects: circle1 and circle2
- circle2 made using the copy constructor by copying circle1's data
- Pgrm then mods the radius in circle1 and circle2 and displays their new radius
- Note- memberwise assignment operator and copy constructor are similar in the sense that both assign vals from one object to the other, diff is that a new object is made using a copy constructor, using the assignment operator doesn't make new objects
- Default copy constructor or assignment operator for copying objects performs a shallow copy instead of a deep copy, meaning that if the field is a pointer to some object, the address of the

pointer is copied instead of the contents

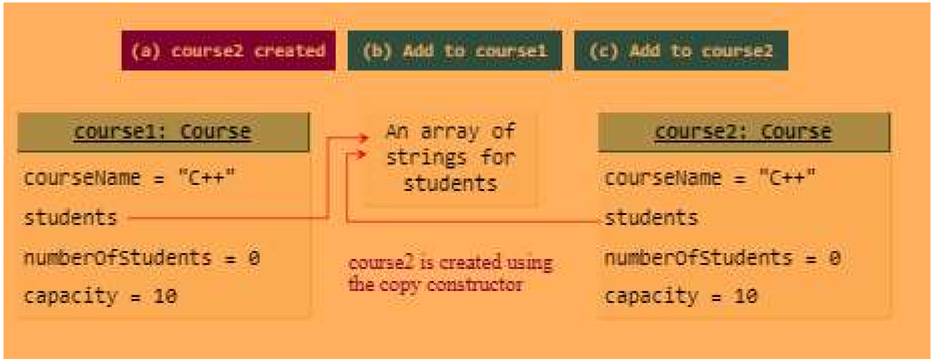
  - ○ Shallow copy – when cloning an object and fields that are a reference type do not have data members copied
  - ○ Deep Copy – when cloning an object and all its fields are cloned recursively

- 

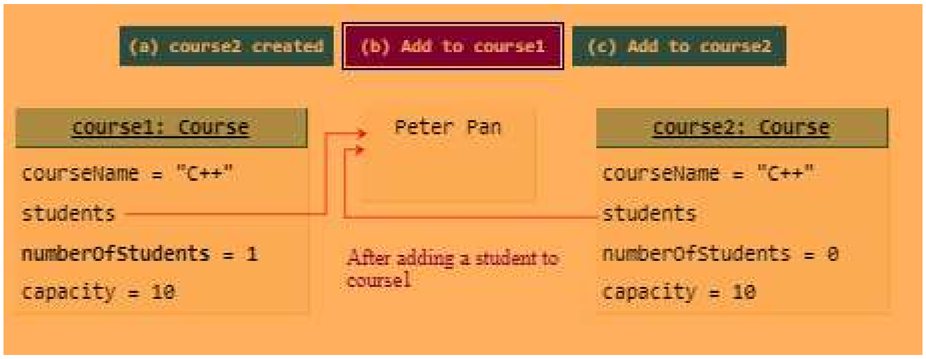
```
#include <iostream>
#include "Course.h" // Defined in Listing 11.14
using namespace std;

int main()
{
  Course course1("C++", 10);
  Course course2(course1);

  course1.addStudent("Peter Pan"); // Add a student to course1
  course2.addStudent("Lisa Ma"); // Add a student to course2

  cout << "students in course1: " <<
    course1.getStudents()[0] << endl; // Display first student in course1
  cout << "students in course2: " <<
    course2.getStudents()[0] << endl;

  return 0;
}
```

Execution Result:

```
command>cl ShallowCopyDemo.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>ShallowCopyDemo
students in course1: Lisa Ma
students in course2: Lisa Ma

command>
```

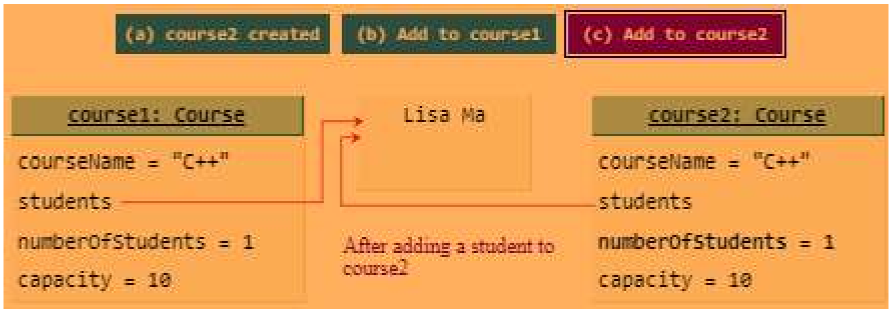
- Course class from last time, pgrm makes a Course object course1, and makes another Course object course2 using the copy constructor, couse2 is a copy of course1
- Course class has 4 data fields: courseName, numberOfStudents, capacity, and students
- The students field is a pointer type, when course1 is copied to course2, all the data fields are copied to course2
- Since students is a pointer, its vals in course1 is copied to course2
- Now both students in course1 and couse2 pt to the same array object
- 

course2 is created using the copy constructor

-
- 

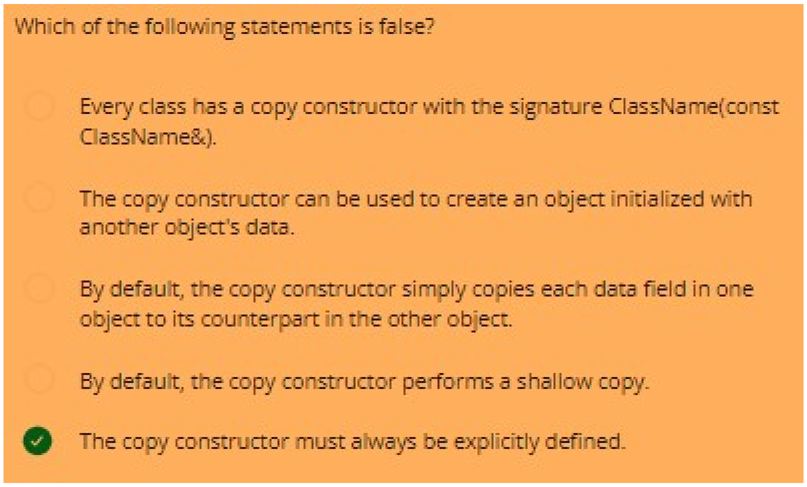
After adding a student to course1

-

- 

- Since both course1 and course2's students pointer point to the same array, the array will be deleted twice, causing a runtime error, to avoid, should just do a deep copy so that course1 and course2 have independent arrays to store student names

-

- 

  Which of the following statements is false?

  ○ Every class has a copy constructor with the signature ClassName(const ClassName&).

  ○ The copy constructor can be used to create an object initialized with another object's data.

  ○ By default, the copy constructor simply copies each data field in one object to its counterpart in the other object.

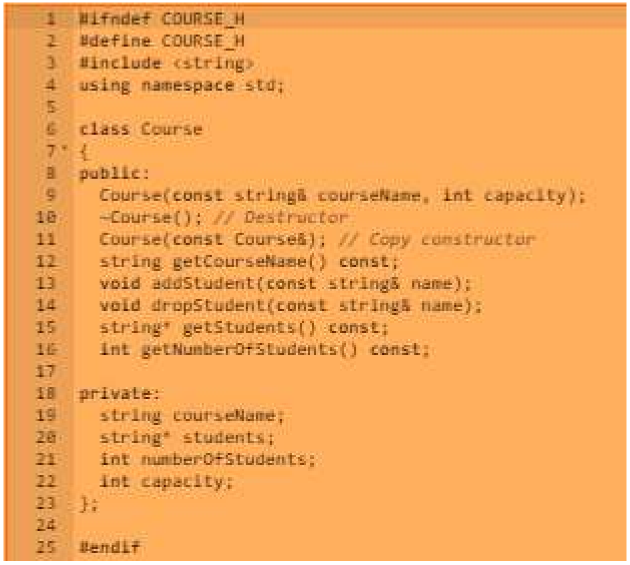
  ○ By default, the copy constructor performs a shallow copy.

  ✓ The copy constructor must always be explicitly defined.

-

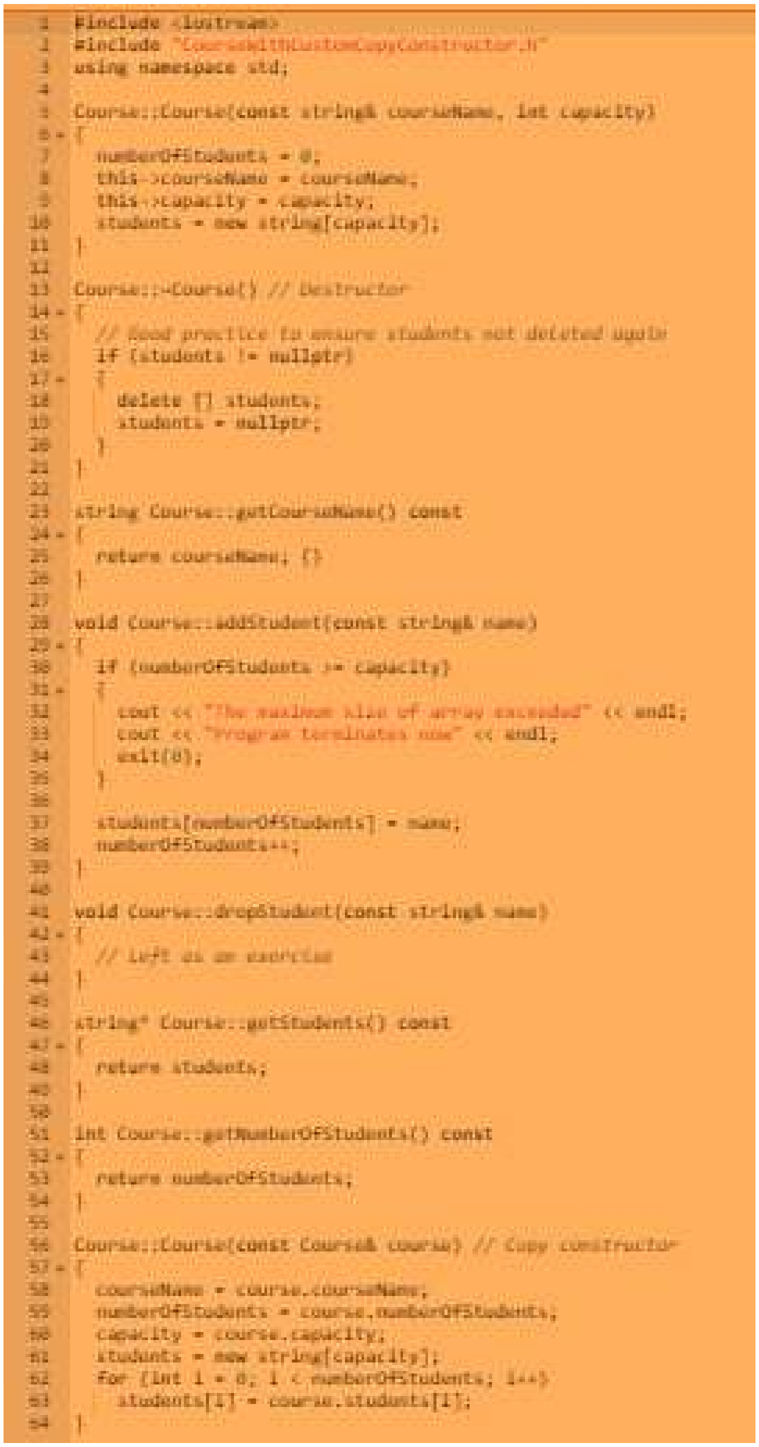
11.15 Customizing Copy Constructors

Thursday, March 23, 2023 5:11 PM

- Can customize the copy constructor to do deep copy
- Default copy constructor/assignment operator = do shallow copy
- To do deep copy, can implement the copy constructor
- 

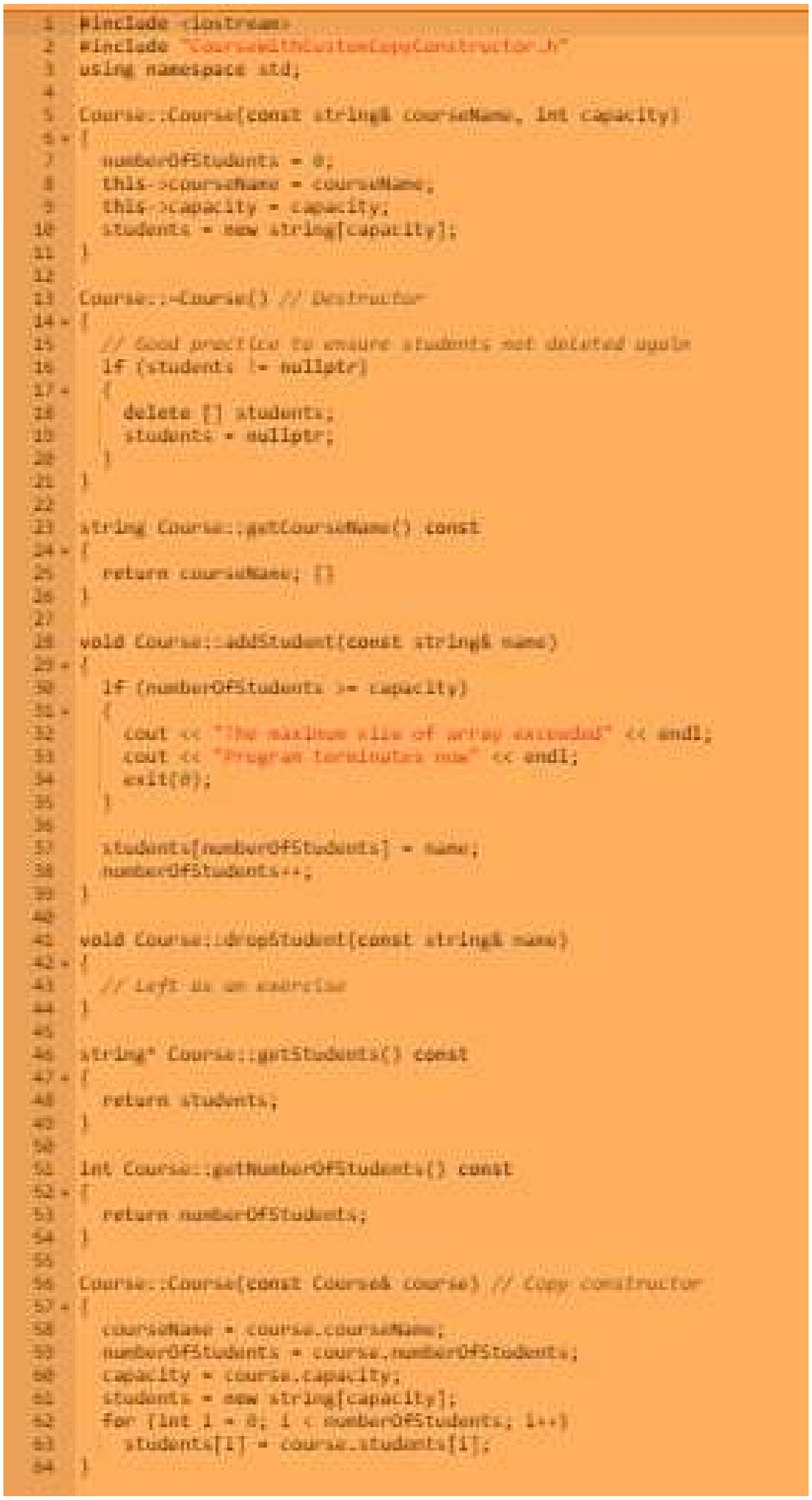
```
#ifndef COURSE_H
#define COURSE_H
#include <string>
using namespace std;

class Course
{
public:
  Course(const string& courseName, int capacity);
  ~Course(); // Destructor
  Course(const Course&); // Copy constructor
  string getCourseName() const;
  void addStudent(const string& name);
  void dropStudent(const string& name);
  string* getStudents() const;
  int getNumberOfStudents() const;

private:
  string courseName;
  string* students;
  int numberOfStudents;
  int capacity;
};

#endif
```

- This revises the Course class to define a copy constructor

- 

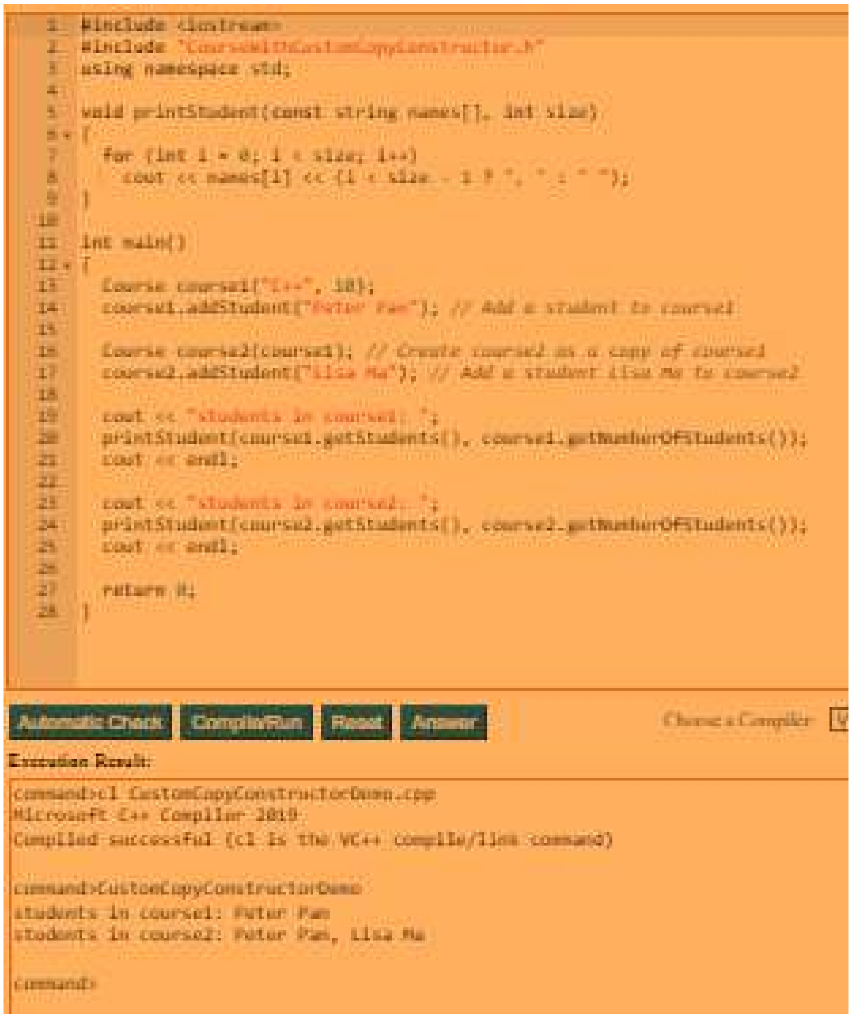
```
#include <iostream>
#include "CourseWithCustomCopyConstructor.h"
using namespace std;

Course::Course(const string& courseName, int capacity)
{
  numberOfStudents = 0;
  this->courseName = courseName;
  this->capacity = capacity;
  students = new string[capacity];
}

Course::~Course() // Destructor
{
  // Good practice to ensure students not deleted again
  if (students != nullptr)
  {
    delete [] students;
    students = nullptr;
  }
}

string Course::getCourseName() const
{
  return courseName; {}
}

void Course::addStudent(const string& name)
{
  if (numberOfStudents >= capacity)
  {
    cout << "The maximum size of array exceeded" << endl;
    cout << "Program terminates now" << endl;
    exit(0);
  }

  students[numberOfStudents] = name;
  numberOfStudents++;
}

void Course::dropStudent(const string& name)
{
  // Left as an exercise
}

string* Course::getStudents() const
{
  return students;
}

int Course::getNumberOfStudents() const
{
  return numberOfStudents;
}

Course::Course(const Course& course) // Copy constructor
{
  courseName = course.courseName;
  numberOfStudents = course.numberOfStudents;
  capacity = course.capacity;
  students = new string[capacity];
  for (int i = 0; i < numberOfStudents; i++)
    students[i] = course.students[i];
}
```

- This implements the new copy constructor, it copies courseName, numberOfStudents, and capacity from one course object to this course object, a new array is made to hold student names in this object

- 

```cpp
#include <iostream>
#include "CourseWithCustomCopyConstructor.h"
using namespace std;

Course::Course(const string& courseName, int capacity)
{
  numberOfStudents = 0;
  this->courseName = courseName;
  this->capacity = capacity;
  students = new string[capacity];
}

Course::~Course() // Destructor
{
  // Good practice to ensure students not deleted again
  if (students != nullptr)
  {
    delete [] students;
    students = nullptr;
  }
}

string Course::getCourseName() const
{
  return courseName; []
}

void Course::addStudent(const string& name)
{
  if (numberOfStudents >= capacity)
  {
    cout << "The maximum size of array exceeded" << endl;
    cout << "Program terminates now" << endl;
    exit(0);
  }

  students[numberOfStudents] = name;
  numberOfStudents++;
}

void Course::dropStudent(const string& name)
{
  // Left as an exercise
}

string* Course::getStudents() const
{
  return students;
}

int Course::getNumberOfStudents() const
{
  return numberOfStudents;
}

Course::Course(const Course& course) // Copy constructor
{
  courseName = course.courseName;
  numberOfStudents = course.numberOfStudents;
  capacity = course.capacity;
  students = new string[capacity];
  for (int i = 0; i < numberOfStudents; i++)
    students[i] = course.students[i];
}
```

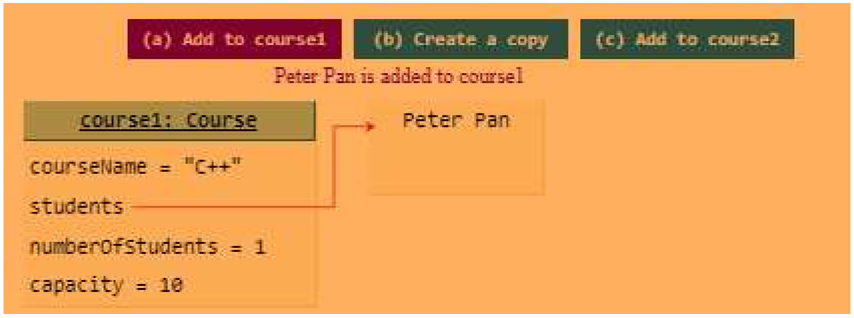
-

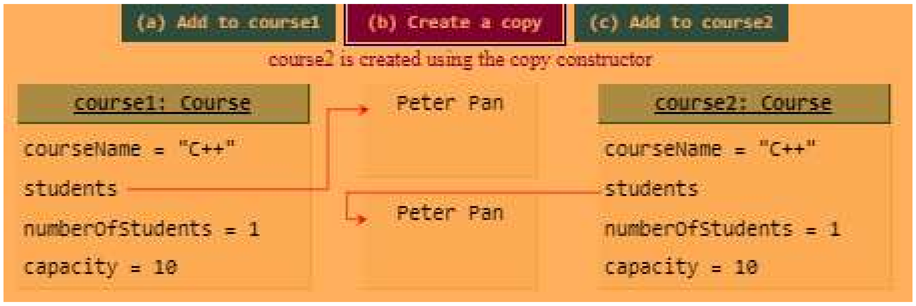
- 

```
#include <iostream>
#include "CourseWithCustomCopyConstructor.h"
using namespace std;

void printStudent(const string names[], int size)
{
  for (int i = 0; i < size; i++)
    cout << names[i] << (i < size - 1 ? ", " : " ");
}

int main()
{
  Course course1("C++", 10);
  course1.addStudent("Peter Pan"); // Add a student to course1

  Course course2(course1); // Create course2 as a copy of course1
  course2.addStudent("Lisa Ma"); // Add a student Lisa Ma to course2

  cout << "students in course1: ";
  printStudent(course1.getStudents(), course1.getNumberOfStudents());
  cout << endl;

  cout << "students in course2: ";
  printStudent(course2.getStudents(), course2.getNumberOfStudents());
  cout << endl;

  return 0;
}
```

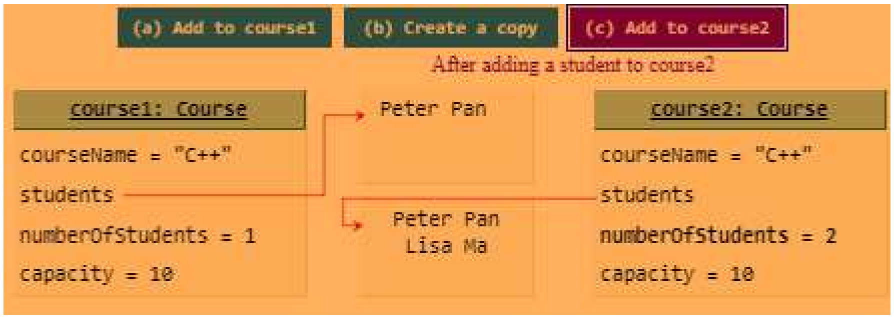
Execution Result:

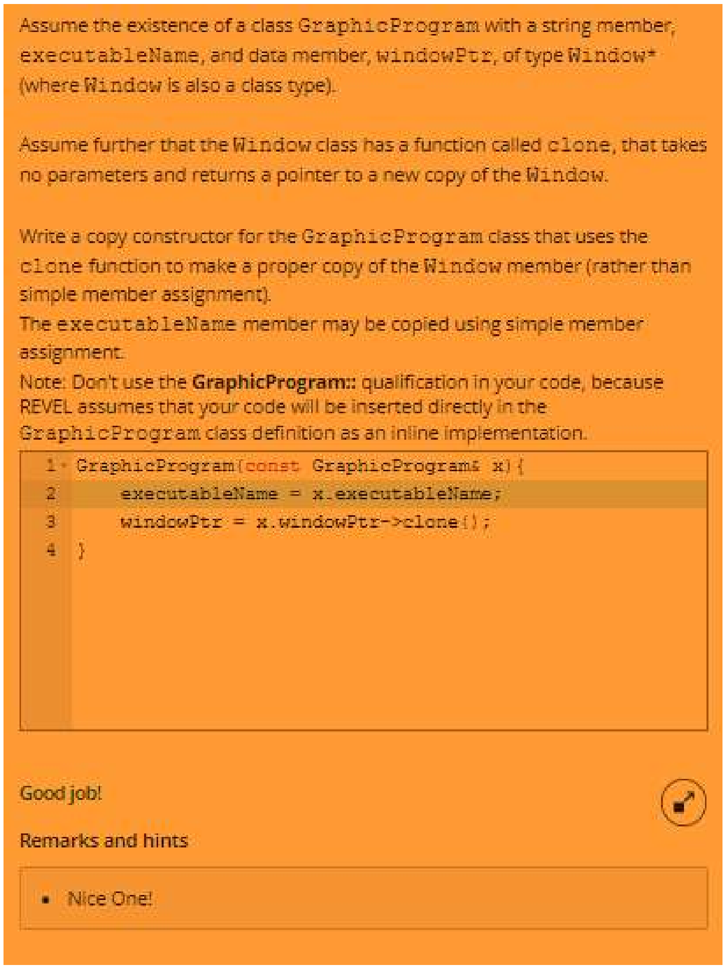
```
command>cl CustomCopyConstructorDemo.cpp
Microsoft C++ Compiler 2019
Compiled successful (cl is the VC++ compile/link command)

command>CustomCopyConstructorDemo
students in course1: Peter Pan
students in course2: Peter Pan, Lisa Ma

command>
```

- The pgrm makes course1 and adds a student "Peter Pan" to course1, the copy constructor makes a new array in course2 for stroing student names that is independent of the array in course1.
- A student "Lisa Ma" is added to course2, the first student in course1 is now "Peter Pan" and in course2 is "Peter Pan, Lisa Ma"
- 

- 

-

- 

- The custom copy constructor does not change the behavior of the memberwise copy operator = by default, later learn how to customize = operator
-
- 

Assume the existence of a class `GraphicProgram` with a string member, `executableName`, and data member, `windowPtr`, of type `Window*` (where `Window` is also a class type).

Assume further that the `Window` class has a function called `clone`, that takes no parameters and returns a pointer to a new copy of the `Window`.

Write a copy constructor for the `GraphicProgram` class that uses the `clone` function to make a proper copy of the `Window` member (rather than simple member assignment).
The `executableName` member may be copied using simple member assignment.
Note: Don't use the **GraphicProgram::** qualification in your code, because REVEL assumes that your code will be inserted directly in the `GraphicProgram` class definition as an inline implementation.

```
GraphicProgram(const GraphicProgram& x){
    executableName = x.executableName;
    windowPtr = x.windowPtr->clone();
}
```

Good job!

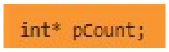
Remarks and hints

- Nice One!

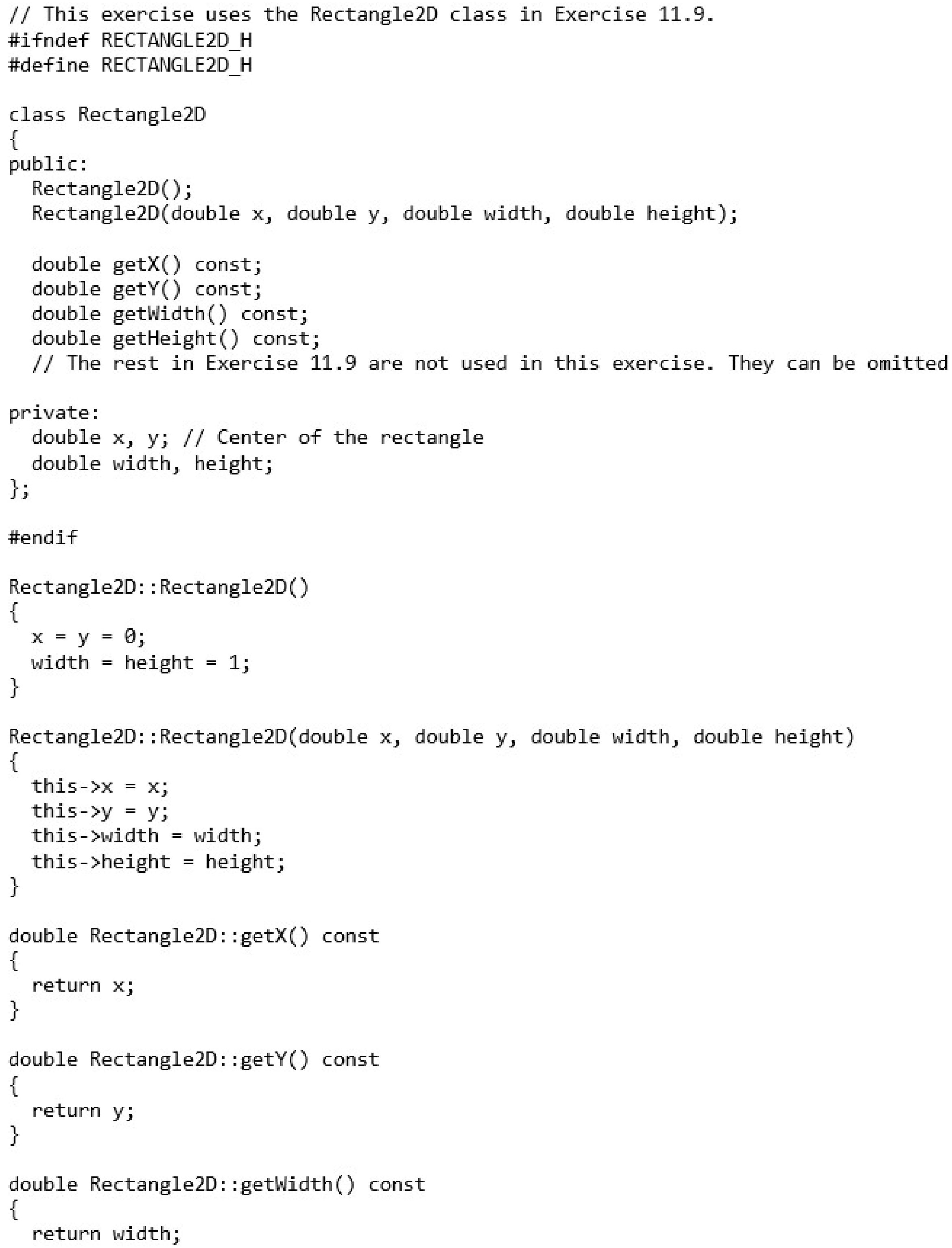
-
-

Chapter Summary

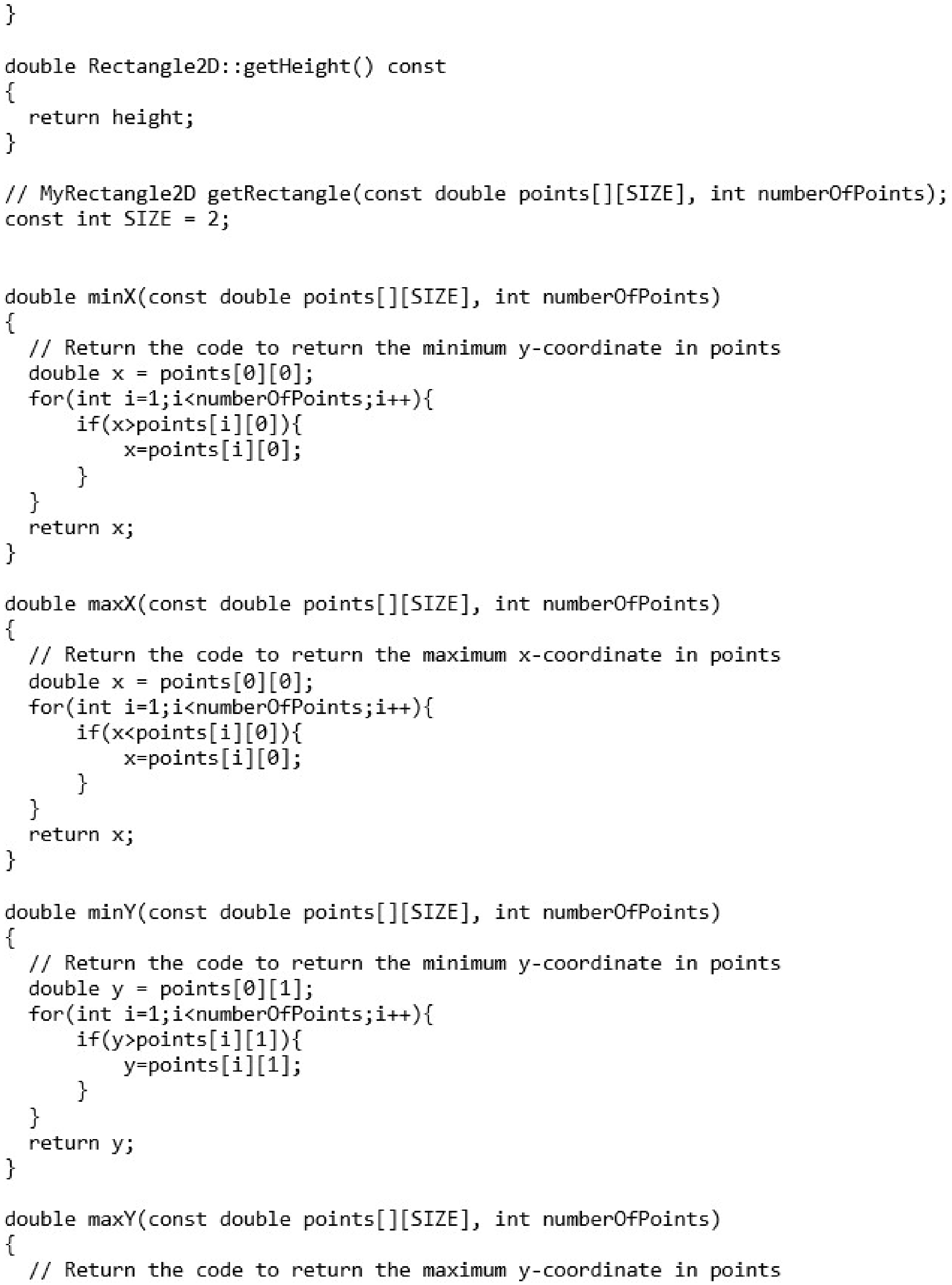
Thursday, March 23, 2023 9:31 PM

1. Pointers are variables that store the memory address of other variables.
2. The declaration
   a. 
   ```
   int* pCount;
   ```
   declares pCount to be a pointer that can point to an int variable.
3. The ampersand (&) symbol is called the address operator when placed in front of a variable. It is a unary operator that returns the address of the variable.
4. A pointer variable is declared with a type such as int or double. You have to assign it with the address of the variable of the same type.
5. Like a local variable, a local pointer is assigned an arbitrary value if you don't initialize it.
6. If a pointer does not reference to a value, it should be initialized to nullptr to prevent potential memory errors.
7. The asterisk (*) placed before a pointer is known as the indirection operator or dereference operator (dereference means indirect reference).
8. When a pointer is dereferenced, the value at the address stored in the pointer is retrieved.
9. The const keyword can be used to declare constant pointer and constant data.
10. An array name is actually a constant pointer that points to the starting address of the array.
11. You can access array elements using pointers or via index.
12. An integer may be added or subtracted from a pointer. The pointer is incremented or decremented by that integer times the size of the element to which the pointer points.
13. A pointer argument can be passed by value or by reference.
14. A pointer may be returned from a function. But you should not return the address of a local variable from a function, because a local variable is destroyed after the function is returned.
15. The new operator can be used to allocate persistent memory on the heap.
16. You should use the delete operator to release the memory created using the new operator, when the memory is no longer needed.
17. You can use pointers to reference an object and access object data fields and invoke functions.
18. You can create objects dynamically in a heap using the new operator.
19. The keyword this can be used as a pointer to the calling object.
20. Destructors are the opposite of constructors.
21. Constructors are invoked to create objects, and destructors are invoked automatically when objects are destroyed.
22. Every class has a default destructor, if the destructor is not explicitly defined.
23. The default destructor does not perform any operations.
24. Every class has a default copy constructor, if the copy constructor is not explicitly defined.
25. The default copy constructor simply copies each data field in one object to its counterpart in the other object.

```
// This exercise uses the Rectangle2D class in Exercise 11.9.
#ifndef RECTANGLE2D_H
#define RECTANGLE2D_H
```



```
// This exercise uses the Rectangle2D class in Exercise 11.9.
#ifndef RECTANGLE2D_H
#define RECTANGLE2D_H

class Rectangle2D
{
public:
  Rectangle2D();
  Rectangle2D(double x, double y, double width, double height);

  double getX() const;
  double getY() const;
  double getWidth() const;
  double getHeight() const;
  // The rest in Exercise 11.9 are not used in this exercise. They can be omitted

private:
  double x, y; // Center of the rectangle
  double width, height;
};

#endif

Rectangle2D::Rectangle2D()
{
  x = y = 0;
  width = height = 1;
}

Rectangle2D::Rectangle2D(double x, double y, double width, double height)
{
  this->x = x;
  this->y = y;
  this->width = width;
  this->height = height;
}

double Rectangle2D::getX() const
{
  return x;
}

double Rectangle2D::getY() const
{
  return y;
}

double Rectangle2D::getWidth() const
{
  return width;
```



```
}

double Rectangle2D::getHeight() const
{
  return height;
}

// MyRectangle2D getRectangle(const double points[][SIZE], int numberOfPoints);
const int SIZE = 2;


double minX(const double points[][SIZE], int numberOfPoints)
{
  // Return the code to return the minimum y-coordinate in points
  double x = points[0][0];
  for(int i=1;i<numberOfPoints;i++){
      if(x>points[i][0]){
          x=points[i][0];
      }
  }
  return x;
}

double maxX(const double points[][SIZE], int numberOfPoints)
{
  // Return the code to return the maximum x-coordinate in points
  double x = points[0][0];
  for(int i=1;i<numberOfPoints;i++){
      if(x<points[i][0]){
          x=points[i][0];
      }
  }
  return x;
}

double minY(const double points[][SIZE], int numberOfPoints)
{
  // Return the code to return the minimum y-coordinate in points
  double y = points[0][1];
  for(int i=1;i<numberOfPoints;i++){
      if(y>points[i][1]){
          y=points[i][1];
      }
  }
  return y;
}

double maxY(const double points[][SIZE], int numberOfPoints)
{
  // Return the code to return the maximum y-coordinate in points
  double y = points[0][1];
  for(int i=1;i<numberOfPoints;i++){
      if(y<points[i][1]){
          y=points[i][1];
      }
  }
  return y;
}

Rectangle2D* getRectanglePointer(const double points[][SIZE], int numberOfPoints)
{
  // Write your code to return a point to a Rectangle2D object that is the minimum
bounding rectangle for the points
  // Hint: invoke minX, minY, maxX, and maxY to find the minimum x, y and maximum x,
y for the bounding rectangle.
  double centerX = (maxX(points,numberOfPoints) + minX(points,numberOfPoints))/2;
  double centerY = (maxY(points,numberOfPoints) + minY(points,numberOfPoints))/2;
  double widthX = maxX(points,numberOfPoints) - minX(points,numberOfPoints);
  double heightY = maxY(points,numberOfPoints) - minY(points,numberOfPoints);
  Rectangle2D rectAdd = Rectangle2D(centerX, centerY, widthX, heightY);
  Rectangle2D *rectReturn = &rectAdd;
  return rectReturn;
}

Rectangle2D getRectangle(const double points[][SIZE], int numberOfPoints)
{
  // Write your code to return a Rectangle2D object that is the minimum bounding
rectangle for the points
  // Hint: invoke minX, minY, maxX, and maxY to find the minimum x, y and maximum x,
y for the bounding rectangle.
  double centerX = (maxX(points,numberOfPoints) + minX(points,numberOfPoints))/2;
  double centerY = (maxY(points,numberOfPoints) + minY(points,numberOfPoints))/2;
  double widthX = maxX(points,numberOfPoints) - minX(points,numberOfPoints);
  double heightY = maxY(points,numberOfPoints) - minY(points,numberOfPoints);
  Rectangle2D rectReturn = Rectangle2D(centerX, centerY, widthX, heightY);
  return rectReturn;
}

#include <iostream>
using namespace std;

int main()
{
  double points[5][2];
  cout << "Enter five points: ";
  for(int i=0; i<5;i++){
      for(int j=0; j<2; j++){
          cin>>points[i][j];
      }
  }
  Rectangle2D boundingRectangle = getRectangle(points, 5);
  cout << "The bounding rectangle's center (" << boundingRectangle.getX() << ", " <<

    boundingRectangle.getY() << "), width " << boundingRectangle.getWidth() <<
          ", height " << boundingRectangle.getHeight() << endl;

  Rectangle2D* boundingRectangleP = getRectanglePointer(points, 5);
  cout << "The bounding rectangle's center (" << boundingRectangleP->getX() << ", "
<<
          boundingRectangleP->getY() << "), width " <<
boundingRectangleP->getWidth() <<
          ", height " << boundingRectangleP->getHeight() << endl;

  return 0;
}
```