

**Instructions:** This homework assignment focuses on enumerated types, arrays and struct variables.

---

For questions 1 through 3, assume the type definition:

```
struct PersonRec {
    int Age;
    int Height;
    int Weight;
};
```

1) Which of the following is/are valid for creating and initializing a PersonRec variable?

- |   |                  |
|---|------------------|
| 1) PersonRec me;<br>me.Age = 19;<br>me.Height = 174;<br>me.Weight = 83;     | 4) All of these  |
| 2) PersonRec me = {19, 174, 83};  | 5) 1 and 2 only  |
| 3) PersonRec.Age = 19;<br>PersonRec.Height = 174;<br>PersonRec.Weight = 83; | 6) 1 and 3 only  |
|   | 7) 2 and 3 only  |
|   | 8) None of these |

2) Given the variable declarations: PersonRec You, Me;

and assuming the two variables have been properly initialized, which of the following is/are valid expressions or statements involving PersonRec variables?

- |                 |                  |
|-----------------|------------------|
| 1) You = Me;    | 5) 1 and 2 only  |
| 2) You == Me    | 6) 1 and 3 only  |
| 3) cout << You; | 7) 2 and 3 only  |
| 4) All of these | 7) None of these |

3) Given the variable declarations: PersonRec You, Me;

and assuming the two variables have been properly initialized, which of the following is/are valid expressions or statements involving PersonRec variables?

- |                      |                  |
|----------------------|------------------|
| 1) You.Age++;        | 5) 1 and 2 only  |
| 2) Me[2] = 4;        | 6) 1 and 3 only  |
| 3) Me.Age >= You.Age | 7) 2 and 3 only  |
| 4) All of these      | 8) None of these |

For questions 4 through 7, consider the declarations:

```
const int MaxEmployees = 1000;
struct IDType {
    string Name;
    int    IDNum;
    double HourlyPayRate;
};

IDType Emp5 = {"Joe Bob Hokie", 4242, 19.87},
        Emp7 = {"Haskell Hoo IV", 1024, 9.32};
IDType Personnel[MaxEmployees];
```

Assume that the array `Personnel[]` has been initialized so that all fields of all the array elements have logically correct values.

4) The name fields of the variables `Emp5` and `Emp7` could be compared by the expression(s):

- |  |                  |
|--|------------------|
| 1) <code>Emp5.Name == Emp7.Name</code>   | 5) 1 and 2 only  |
| 2) <code>Emp5.Name = Emp7.Name</code>    | 6) 1 and 3 only  |
| 3) <code>Emp7[Name] == Emp7[Name]</code> | 7) 2 and 3 only  |
| 4) All of these                          | 8) None of these |

5) The variables `Emp5` and `Emp7` could be compared by the expression(s):

- |                                 |                  |
|---------------------------------|------------------|
| 1) <code>Emp5 == Emp7</code>    | 5) 1 and 2 only  |
| 2) <code>Emp7 = Emp7</code>     | 6) 1 and 3 only  |
| 3) <code>Emp5 &lt;= Emp7</code> | 7) 2 and 3 only  |
| 4) All of the above             | 8) None of these |

6) The statement: `cout << Personnel[17].Name;`

- 1) prints the name of the employee with ID number 17
- 2) prints the name of the employee whose `IDType` record is stored at index 17
- 3) is syntactically illegal since you can't dump a `struct` variable into an output stream
- 4) None of these

7) The statement: `Personnel[17] = Emp5;`

- 1) copies the contents of the `IDType` record at index 17 into the variable `Emp5`
- 2) copies the contents of the variable `Emp5` into the `IDType` record at index 17
- 3) swaps the contents of the variable `Emp5` and the `IDType` record at index 17
- 4) is syntactically illegal
- 5) None of these

8) Suppose that a `struct` parameter is to be passed to a function, and that the function does not need to modify either the formal parameter or the actual parameter. What advantage(s) could be gained by passing the parameter by constant reference instead of passing it by value in this situation?

- |  |                  |
|--|------------------|
| 1) The syntax of the function call would be simpler. | 5) 1 and 2 only  |
| 2) This would require less space in memory.          | 6) 1 and 3 only  |
| 3) This would probably require less execution time.  | 7) 2 and 3 only  |
| 4) All of these                                      | 8) None of these |

9) Passing a `struct` parameter by constant reference instead of by reference:

- |  |                  |
|--|------------------|
| 1) reduces the amount of memory needed               | 5) 1 and 3 only  |
| 2) increases the amount of memory needed             | 6) 2 and 4 only  |
| 3) reduces the time required for the function call   | 7) None of these |
| 4) increases the time required for the function call |                  |

For questions 10 through 13, consider the declarations and function:

```

struct Bar {
    int Foo[5];
    int Size;
};

Bar x = {{2, 5, 9, 11, 17}, 5}, // This is a legal initialization.
      y = {{0, 3, 6}, 3};

Bar AddBar(const Bar x, const Bar y) {
    Bar NewBar;
    if (x.Size <= y.Size)
        NewBar.Size = x.Size;
    else
        NewBar.Size = y.Size;

    for (int Idx = 0; Idx < NewBar.Size; x++)
        NewBar.Foo[Idx] = x.Foo[Idx] + y.Foo[Idx];

    return NewBar;
}

```

10) The statement: `y = x;`

- 1) would copy the contents of the `Bar` variable `x` into the `Bar` variable `y`
- 2) is syntactically illegal because you can't assign an array to an array
- 3) is syntactically legal, but has a different effect than 1)
- 4) None of these

11) The statement: `y.Foo = x.Foo;`

- 1) would copy the contents of the `Foo` field of `x` into the `Foo` field of `y`
- 2) is syntactically illegal because you can't assign an array to an array
- 3) is syntactically legal, but has a different effect than 1)
- 4) None of these

12) Suppose the following statement is executed: `Bar z = AddBar(x, y);`

Then the value of `z.Foo[2]` would be:

- |      |      |                  |
|------|------|------------------|
| 1) 0 | 3) 8 | 5) 15            |
| 2) 2 | 4) 9 | 6) None of these |

13) Again, suppose the following statement is executed: `Bar z = AddBar(x, y);`

Then the value of `z.Size` would be:

- |      |      |                  |
|------|------|------------------|
| 1) 0 | 4) 3 | 7) None of these |
| 2) 1 | 5) 4 |                  |
| 3) 2 | 6) 5 |                  |

---

For questions 14 and 15 assume the following declarations:

```
enum WoodKind {ASH, CEDAR, OAK, PINE, POPLAR, WALNUT};

struct Size {
    int Length;
    int Width;
    int Thickness;
};

struct Board {
    Size Dimensions;
    WoodKind Kind;
    int smoothSurfaces;
};

Board oneBoard = {{2, 4, 8}, PINE, 4}; // Legal initialization.
```

14) The number of smooth surfaces of `oneBoard` could be printed by the statement(s):

- |  |                  |
|--|------------------|
| 1) <code>cout &lt;&lt; oneBoard.smoothSurfaces;</code> | 5) 1 and 2 only  |
| 2) <code>cout &lt;&lt; smoothSurfaces;</code>          | 6) 1 and 3 only  |
| 3) <code>cout &lt;&lt; Wood.smoothSurfaces;</code>     | 7) 2 and 3 only  |
| 4) All of these  | 8) None of these |

15) The width of `oneBoard` could be printed by the statement(s):

- |  |                  |
|--|------------------|
| 1) <code>cout &lt;&lt; oneBoard.Width;</code>            | 5) 1 and 2 only  |
| 2) <code>cout &lt;&lt; oneBoard.Size.Width;</code>       | 6) 1 and 3 only  |
| 3) <code>cout &lt;&lt; oneBoard.Dimensions.Width;</code> | 7) 2 and 3 only  |
| 4) All of the above                                      | 8) None of these |

For questions 16 and 17 assume the declarations given for questions 14 and 15, and also the declarations:

```
const int MAXBOARDS = 1000;
Board Lumber[MAXBOARDS];
```

Consider implementing a loop to print the dimensions of all OAK boards that occur in the array `Lumber[]`, assuming the array has been initialized to hold data about `numBoards` boards:

```
for (int Idx = 0; Idx < numBoards; Idx++) {
    if ( _____ == _____ ) {
        cout << "Length:    " << Lumber[Idx].Size.Length << endl
             << "Width:       " << Lumber[Idx].Size.Width << endl
             << "Thickness:  " << Lumber[Idx].Size.Thickness << endl;
    }
}
```

16) How should the first blank in the `if` condition be filled?

- |                                      |                               |
|--------------------------------------|-------------------------------|
| 1) <code>Lumber[Idx].WoodKind</code> | 4) <code>Kind</code>          |
| 2) <code>Lumber[Idx].Kind</code>     | 5) <code>None of these</code> |
| 3) <code>WoodKind</code>             |                               |

17) How should the second blank in the `if` condition be filled?

- |                                  |                               |
|----------------------------------|-------------------------------|
| 1) <code>"OAK"</code>            | 4) <code>Either 1 or 2</code> |
| 2) <code>" OAK "    "oak"</code> | 5) <code>None of these</code> |
| 3) <code>OAK</code>              |                               |

For questions 18 through 20 assume the following enumerated type declaration:

```
enum Status {NEGATIVE, OK, UNUSED, TOOBIG};
```

The function `checkIndex()` is intended to validate an array index, as you did in Project 8, and indicate to the caller what is determined:

```
_____ checkIndex(const int Idx,           // index to be checked
                  const int Used,         // # of used cells in array
                  const int Dimension) {  // dimension of array

    if (Idx < 0)
        return _____;           // line 1
    if (Idx < Used)
        return OK;
    if (Idx < Dimension)
        return UNUSED;
    return _____;               // line 2
}
```

18) What should the return type of the function be?

- |                        |                               |
|------------------------|-------------------------------|
| 1) <code>UNUSED</code> | 4) <code>Status</code>        |
| 2) <code>void</code>   | 5) <code>None of these</code> |
| 3) <code>int</code>    |                               |

19) How should the blank in line 1 be filled?

- 1) NEGATIVE
- 2) Status
- 3) -1

- 4) 0
- 5) None of these

**20)** How should the blank in line 2 be filled?

- 1) TOOBIG
- 2) OK
- 3) Status

- 4) It should be left blank.
  - 5) None of these
-