Data Races Solutions

Data Races 1

```
#include <thread>
#include <iostream>

using namespace std::literals;

void task() {
    for (int i = 0; i < 5; ++i) {
      std::cout << "I'm a task with ID " << std::this_thread::get_id() << "!" << std::endl;
      std::this_thread::sleep_for(50ms);
    }
}

int main() {
  std::thread t{task};
  std::thread t2{task};
  std::thread t3{task};
  t.join();
  t2.join();
  t3.join();
}
```

Data Races 1

- Sample output:

```
I'm a task with ID I'm a task with ID I'm a task with ID 2!
34!!

I'm a task with ID 4!
I'm a task with ID I'm a task with ID 32!!
```

- The output from the threads is interleaved due to a data race
  - Multiple threads modify a memory location (the variable cout)
  - The modifications are not atomic
  - There is no ordering of the modifying threads

Data Races 2

- Can a data race occur in the following code sample, when func1 and func2 are run as concurrent threads? Explain your answer

```
const int x{5};

int func1() {
   return 2*x;
}

int func2() {
   return 3*x;
}
```

- x is const, so no thread can modify it
- There is no possibility of conflicting accesses to x (unless a thread dangerously casts away const)
- The code shown is data-race free

Data Races 2

```
#include <thread>
#include <iostream>

using namespace std;

const int x{5};

int func1() {
   return 2*x;
}

int func2() {
   return 3*x;
}

int main() {
   thread f1{ func1 };
   thread f2{ func2 };
   f1.join();
   f2.join();
}
```

Data Races 3

- Can a data race occur in the following code sample, when func1 and func2 are run as concurrent threads? Explain your answer

```
int x{0}, y{0};

void func1() {
   if (x)
      y = 1;
}

void func2() {
   if (y)
      x = 1;
}
```

- In func1, x is always 0, so y is never set to 1. In func2, y is always 0, so x is never set to 1
- There is no possible execution path in which more than one thread tries to modify x or y
- The code is data-race free

Data Races 3

```
// #includes and main() as before
int x{0}, y{0};

void func1() {
    if (x) {
        y = 1;                                  // Never executed
        cout << "y set\n";
    }          }
}

void func2() {
    if (y) {
        x = 1;                                  // Never executed
        cout << "x set\n";
    }
}
```

Data Races 4

- Can a data race occur in the following code sample, when func1 and func2 are run as concurrent threads? Explain your answer

```
int x{0}, y{0};

void func1() {
   x = 1;
   int r1 = y;
}

void func2() {
   y = 1;
   int r2 = x;
}
```

- It is possible for func1 to read y while func2 is modifying it, and vice versa for x.
- The accesses are not atomic and are not ordered, so we have a data race

Data Races 5

- Can a data race occur in the following code sample, when func1 and func2 are run as concurrent threads? Explain your answer

```
int x{0};
bool done{false};

void func1() {
   std::this_thread::sleep_for(50ms);
   x = 42;
   done = true;
}

void func2() {
   std::this_thread::sleep_for(50ms);
   while (!done) {}
   std::cout << x << std::endl;
}
```

Data Races 5

- There are two data races, on x and done
- The compiler optimizes the loop in func2, because it does not know that done can be modified by func1
- It assumes done is a constant. This allows it to generate more efficient code, but causes the loop to run for ever
  - We can declare done as "volatile" to prevent this optimization
  - In Java and C#, the volatile keyword means that modifying done will be performed as an atomic operation. This removes the data race on done
  - However, in C++ we use a different keyword for atomic operations. The data race on done will still exist if it is declared volatile, even though the loop will now run correctly