Chapter 1 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. Which of the following statements is/are **TRUE** regarding the integer types of `int32_t` and `uint32_t`?
   - **I.** The `uint32_t` data type cannot represent a negative number, while the `int32_t` data type can.
   - **II.** The largest integer value representable using the `int32_t` and `uint32_t` data types are the same.
   - **III.** Because a `uint32_t` needs to cover a wider range of potential values, it takes up more bytes in memory than a `int32_t`.

   A) I only
   B) II only
   C) III only
   D) I and II only
   E) I and III only

2. Which of the following statements is/are **TRUE** regarding the difference between using a `struct` and a `class` to create a custom type?
   A) A `struct` can support member functions, while a `class` cannot
   B) A `class` can support member functions, while a `struct` cannot
   C) By default, access to members in a `struct` is `public`, while access to members in a `class` is `private`
   D) By default, access to members in a `struct` is `private`, while access to members in a `class` is `public`
   E) More than one of the above

3. Consider the following code, which defines a derived class with several member variable types.

```
1   struct ObjectA {
2     ObjectA() { std::cout << "Object A initialized! "; }
3   };
4
5   struct ObjectB {
6     ObjectB() { std::cout << "Object B initialized! "; }
7   };
8
9   struct ObjectC {
10    ObjectC() { std::cout << "Object C initialized! "; }
11  };
12
13  struct Base {
14    ObjectA a;
15  };
16
17  struct Derived : public Base {
18    ObjectB b;
19    ObjectC c;
20  };
21
22  int main() {
23    Derived my_derived_class;
24  } // main()
```

   What is the output of this code? *(Note: The `std::cout` statement prints a message to the console output.)*
   A) `Object A initialized!  Object B initialized!  Object C initialized!`
   B) `Object B initialized!  Object C initialized!  Object A initialized!`
   C) `Object A initialized!  Object C initialized!  Object B initialized!`
   D) `Object C initialized!  Object B initialized!  Object A initialized!`
   E) The ordering of the constructor messages is indeterminate

4. Consider the following code, which declares an integer using the `const` keyword.

```
1   int main() {
2     int32_t v1 = 280;
3     int32_t v2 = 281;
4     int32_t* const ptr = &v1;
5     /* ??? */
6   } // main()
```

   Which of the following expressions, if added on line 5, would result in a compilation error?
   A) `++v1;`
   B) `++(*ptr);`
   C) `*ptr = v2;`
   D) `ptr = &v2;`
   E) More than one of the above

5. Consider the following class:

```
1  class Foo {
2    int32_t bar = 281;
3  public:
4    int32_t process_values(int32_t baz) const {
5      /* ??? */
6    } // process_values()
7  };
```

Which of the following statements is/are **TRUE** regarding this class definition?

I. The `process_values()` member function cannot change the value of the function argument `baz`.
II. The `process_values()` member function cannot change the value of the member variable `bar`.
III. If `bar` had been marked as `mutable` on line 2 (i.e., **`mutable`** `int32_t bar`), then the `process_values()` member function would be able to change the value of `bar`.

A) I only
B) II only
C) I and III only
D) II and III only
E) I, II, and III

6. Consider the following snippet of code:

```
1  class Foo {
2    int32_t bar = 281;
3  public:
4    void do_something() {
5      /* ??? */
6    } // do_something()
7
8    void do_something_const() const {
9      /* ??? */
10   } // do_something_const()
11 };
12
13 int main() {
14   Foo f1{};
15   const Foo f2{};
16
17 } // main()
```

Which of the following lines of code, when added on line 16, would result in a compilation error?

A) `f1.do_something();`
B) `f1.do_something_const();`
C) `f2.do_something();`
D) `f2.do_something_const();`
E) None of the above

7. Suppose you had the following one-argument constructor for a `Person` object:

```
1  class Person {
2    int32_t age;
3  public:
4    explicit Person(int32_t age_in) : age{age_in} {}
5  };
```

Which of the following lines of code would cause a compilation error in this case, but would have *run without issue* had the `Person` constructor not been marked as `explicit`?

A) `Person annie;`
B) `Person billy = billy(18);`
C) `Person cathy(17);`
D) `Person danny = 16;`
E) `Person eddie(15.62);`

8. Which of the following is/are valid advantages of using a `static_cast<>` to convert between two types instead of a C-style cast, provided that it is possible to do so?

I. Static casts are better able to identify incompatible type conversions at compile time.
II. Static casts will always be significantly faster than C-style casts for larger objects.
III. Static casts are less prone to dangerous or undefined behavior.

A) I only
B) II only
C) III only
D) I and III only
E) I, II, and III

9. What is the output of the following code?

```
1  struct Compare {
2    bool operator() (int32_t a, int32_t b) const {
3      return a < b;
4    } // operator()()
5  };
6
7  template <typename T>
8  void conditional_print(const std::vector<int32_t>& values, int32_t conditional) {
9    T comp;
10   for (const int32_t val : values) {
11     if (comp(conditional, val)) {
12       std::cout << val << ' ';
13     } // if
14   } // for
15 } // conditional_print()
16
17 int main() {
18   std::vector<int32_t> values = {183, 203, 280, 281, 370, 376};
19   conditional_print<Compare>(values, 281);
20 } // main()
```

A) `183 203 280`
B) `183 203 280 281`
C) `183 203 280 281 370 376`
D) `281 370 376`
E) `370 376`

10. Suppose your friend wrote the following function, which takes in a vector of integers, multiplies every value by a constant multiplier, and prints out the modified values.

```
1  void multiply_and_print_values(std::vector<int32_t>& vec, int32_t multiplier) {
2    for (auto val : vec) {
3      val *= multiplier;
4    } // for j
5
6    for (auto val : vec) {
7      std::cout << val << ' ';
8    } // for k
9  } // multiply_and_print_values()
```

To test this change, your friend ran the following function. However, to their surprise, the output of this code was `"1 2 3 4 5 "` instead of `"2 4 6 8 10 "`.

```
10 int main() {
11   std::vector<int32_t> values = {1, 2, 3, 4, 5};
12   multiply_and_print_values(values, 2);
13 } // main()
```

There is a bug in the function that is causing this unexpected output. On what line is this bug located?

A) Line 1
B) Line 2
C) Line 3
D) Line 6
E) None of the above

11. Consider the following definition of a `for` loop:

```
for (int32_t i = n; i >= 0; --i) { ... }
```

Which of the following `for` loop definitions exhibits the exact same behavior as the loop above?

A) `for (int32_t i = n; i-- > 0;) { ... }`
B) `for (int32_t i = n + 1; i-- > 0;) { ... }`
C) `for (int32_t i = n + 1; i > 0; i--) { ... }`
D) `for (int32_t i = n; --i > 0;) { ... }`
E) `for (int32_t i = n + 1; --i > 0;) { ... }`

12. Which of the following statements best defines the concept of abstraction with regard to object-oriented programming?

A) The act of hiding a component's internal implementation details from a user and only exposing relevant information for usage
B) The act of combining both data and methods into a single unit
C) The act of reusing the interface of a component when developing the functionality of another component
D) The act of substituting multiple components with shared behavior into the same position of a program's implementation
E) None of the above

13. Consider the following `for` loop, which is designed to print out all values from 281 down to 0 in descending order:

```
1  int main() {
2    for (size_t i = 281; i >= 0; --i) {
3      std::cout << i << std::endl;
4    } // for i
5  } // main()
```

Does this code work as intended? If not, what is the issue?

A) This code does not compile, since you cannot decrement a `size_t`
B) This code does not compile, since `main()` does not explicitly return a value
C) The `for` loop does not terminate, since `size_t` is unsigned, which causes `i >= 0` to always evaluate to true
D) More than one of the above
E) The code has no issues and correctly performs the intended behavior

14. Consider the following function `foo()`, which takes in an argument of type `int32_t`:

```
1  void foo(int32_t x) {
2    /* ...implementation here... */
3  } // foo()
```

Consider the following ways this function can be invoked:

I.
```
void bar(double x) {
  foo(x);
} // bar()
```

II.
```
void baz(int16_t x) {
  foo(x);
} // baz()
```

III.
```
void qux(uint32_t x) {
  foo(x);
} // qux()
```

Which of the above function invocations could result in a loss of precision?

A) I only
B) II only
C) I and III only
D) II and III only
E) I, II, and III

Chapter 1 Exercise Solutions

1. **The correct answer is (A).** Statement I is true because `uint32_t` is unsigned, so it cannot represent a negative number; meanwhile, `int32_t` is signed, so it can represent a negative number. Statement II is false because the `uint32_t` type has a larger maximum value, as it can use the same 32 bits to represent larger values beyond the largest value of a `int32_t` since it does not need to represent negative values. Statement III is false for the same reason: even though `uint32_t` is able to reach a larger maximum value, it is able to do so because it does not need to represent negative values, not because it uses more memory to represent these higher numbers.

2. **The correct answer is (C).** In C++, a `struct` and a `class` are essentially identical, with the main difference being the protection level of its members. For a `struct`, member variables default to `public`, while for a `class`, member variables default to `private`. Both `struct` and `class` are able to support member functions.

3. **The correct answer is (A).** Members in a custom object are initialized in the order they are listed. Members in a base class are initialized before members in classes that derive from it.

4. **The correct answer is (D).** The `const` placement on line 4 indicates that the pointer cannot be modified, but the integer it stores can. Only option (D) modifies the pointer and not the value it points to, which would cause a compilation error.

5. **The correct answer is (D).** The `const` that is applied to the member function on line 4 indicates that the member function is not allowed to modify the member variables of the `class` itself (in this case, the value of `bar`), provided that the member variable is not marked as `mutable` (which gives `const` member functions the permission to modify a variable). By this logic, statements II and III are true. Statement I is false because the function argument `baz` itself is not marked as `const` (e.g., `int32_t baz` instead of `const int32_t baz`), so the member function is still able to change its value.

6. **The correct answer is (C).** Non-`const` member functions can only be called by non-`const` objects (a `const` object cannot invoke a non-`const` member function, since such a function does not guarantee it does not modify any of the object's members). Thus, option (C) would result in a compilation error, since `f2` is `const`, but `do_something()` is not.

7. **The correct answer is (D).** The `explicit` keyword prevents the compiler from using a one-argument constructor to construct an object from an implicit type conversion. The `Person` object takes an `int32_t` as its only argument, so without the `explicit` keyword, the assignment of a `Person` object with an `int32_t` would cause the `int32_t` to be implicitly converted into an object of type `Person` using the one-argument constructor. Adding the `explicit` keyword prevents this. Options (A), (C), and (E) do not involve an implicit type conversion, and option (B) would not compile regardless.

8. **The correct answer is (D).** Static casts are better than C-style casts because they are better able to identify incompatible type conversions at compile time and are thereby less prone to undefined behavior. If you perform a cast incorrectly, using `static_cast<>` could catch this mistake during program compilation. However, C-style casts could still compile and perform the cast during runtime, which may be dangerous.

9. **The correct answer is (E).** The templated function here applies a comparator to each number in a container and prints it out if the comparator returns true. On line 19, we can see that the `Compare` comparator is used to conduct this comparison. With templates, we can essentially substitute `Compare` in place of `T` in the `conditional_print()` function on line 8. Since the overloaded `operator()` of `Compare` returns true if `a < b`, and `conditional` is passed into the value of `a` on line 11, the `if` statement on line 11 only evaluates to true if `conditional < val`. In our example, `conditional` has a value of 281, so the function invocation prints out all numbers in the container that are larger than 281.

10. **The correct answer is (B).** The issue with the code is that the container of numbers is never modified at all, despite the multiplication that is performed on line 3. This is because the loop that performs this multiplication is accessing each integer *by value* instead of by reference. As a result, the multiplication is performed on a local copy of each integer, and not the actual integer in the original container itself. The bug is therefore located on line 2 — to fix this bug, **`auto`** `val` should be replaced with **`auto`**`&` `val`.

11. **The correct answer is (B).** The best way to solve this problem is to identify what the value of `i` should be on the first and last iterations of the loop body. We can see that the first iteration of the loop runs with `i = n`, and the last iteration of the loop runs with `i = 0`. It should also be noted that the termination condition of the `for` loop is checked before the body of the loop runs, so it is also important to keep track of any changes that are made to `i` there as well. For (A) and (D), the value of `i` starts at `n` but gets decremented in the termination condition check before the body of the loop can run. As a result, the first iteration of the loop runs with `i = n - 1`, so these options cannot be correct. For (C), the value of `i` starts at `n + 1` but does not get modified in the termination condition, so the first iteration of the loop also begins at `n + 1`, which is not what we want either. Only options (B) and (E) run the first iteration of the loop body with a value of `i = n`. However, choice (E) is incorrect because it uses prefix decrementation (`--i` instead of `i--`) and subtracts from `i` before comparing with 0. As a result, the loop in (E) does not run an iteration of the loop body with `i = 0`, since `--i > 0` evaluates to false when `i = 1`. On the other hand, the loop in (B) compares the value of `i` with 0 before decrementing its value, so it is still able to run one iteration where `i = 0`, since the `i-- > 0` evaluates to true when `i = 1`.

12. **The correct answer is (A).** Abstraction is the idea of hiding a component's implementation from the user and only providing the relevant information needed to use the component. This essentially separates what a component does (the interface) from how it works (the implementation), and removes the need for a user to understand all the internal details of what they want to use. Choice (B) defines encapsulation, choice (C) defines inheritance, and choice (D) defines polymorphism.

13. **The correct answer is (C).** This is a common bug when working with loops involving unsigned values: `i >= 0` does not work as a termination condition because it is always true! To fix this, you can either cast the variable in the loop to a signed integer so that `i >= 0` can become false, or you can check that `i` does not wrap around to the maximum value of the unsigned type.

14. **The correct answer is (C).** Precision loss may occur if you perform a conversion from one type to a more restrictive type. In this case, `foo()` takes in an `int32_t`, so precision loss may occur if a value is passed into `foo()` that cannot be represented as a `int32_t`. This is true for I and III: if you pass a `double` into a `int32_t`, it gets truncated since `int32_t` values cannot handle floating point values (all decimal information therefore gets lost). Similarly, an unsigned `uint32_t` can store values that are larger than the maximum value representable with a signed `int32_t`, so precision loss may arise here as well. Option II does not result in precision loss since any value that can be represented with an `int16_t` can also be represented with an `int32_t`.