Class Test 2024: 1 Hour 30 Minutes – 35 marks

Instructions

- Answer *all* questions. The questions do not carry equal weight.
- For questions which require you to write source code, note that:
  - You only need to specify `#include`'s if specifically asked.
  - For classes, you can give the implementation entirely in the header file, unless directed otherwise.
  - Marks are not awarded solely for functionality but also for good design, making appropriate use of library functions, following good coding practices, and using a modern, idiomatic C++ style.
  - Your code must be easily understandable or well commented.
  - You may use pencil but then you forfeit the right to query the marks.
- Reference sheets are provided.

Question 1 [Total Marks: 19]

a) Given a class `Rectangle`, what would the type of the variables `bounding` and `box` be? Explain the difference between each call.

(4 marks)

```
auto bounding = Rectangle{};
auto box = new Rectangle{};
```

b) Given the following class definition for Rectangle:

```
class Rectangle
{
public:
    Rectangle(int base = 1, int height = 2);
    int base() const;
    //...
};
```

Is it possible to instantiate a new object by calling:

```
auto bounding = Rectangle{};
```

Explain your answer. (2 marks)

c) Give the code for a `Rectangle` class that represents a rectangle with a base and height. Provide a default constructor which makes use of *in-class member initialization* to set default values for these dimensions. Additionally, provide a constructor that allows users to initialize these dimensions with custom values.

(5 marks)

d) Examine Listings 1 and 2 and give the output of the `main` program, assuming that no compiler optimisations take place. Explain why each line of the output is present.

(8 marks)

```cpp
class Person
{
public:
  Person(const string& name): name_{name}
  { cout << "Constructing a person" << endl; }

  Person(const Person& rhs): name_{rhs.name_}
  { cout << "Copying a person" << endl; }

  ~Person()
  { cout << "Destructing a person" << endl; }

private:
  string name_;
};
```

Listing 1: `Person` class definition

```cpp
Person newPerson()
{
   return Person{"Bobby"};
}

int main()
{
   newPerson();

   return 0;
}
```

Listing 2: Using the `Person` class

Question 2

[Total Marks: 8]

Provide the code for a class which models a cylinder. Client code should have the ability to independently set the radius of the base (r), and the height (h), of the cylinder. Clients must also be able to request the cylinder's volume (V) which is given by the formula:

$$V = \pi r^2 \times h$$

Minimise the number of times that the volume needs to be calculated by using a *lazy calculation* strategy. In other words, calculate the volume only when requested and store the result for future requests. Re-calculate only if necessary. Client code must be completely unaware of the calculation strategy that is being used.

Question 3 [Total Marks: 8]

Give the Git commands required for producing the Git history graph shown in Figure 2 from the initial state shown in Figure 1. In Figure 2, *feature-1*, *feature-2*, and *main* are branch names. *C0* to *C5* represent commits numbered in the order that they are made, and HEAD is pointing to the *main* branch.

You should only use the following commands:

- `git commit`
- `git switch`
- `git branch`
- `git merge`

You may assume that `git add` is called before each `git commit` command.

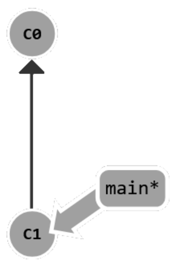

Figure 1: Git history: initial state

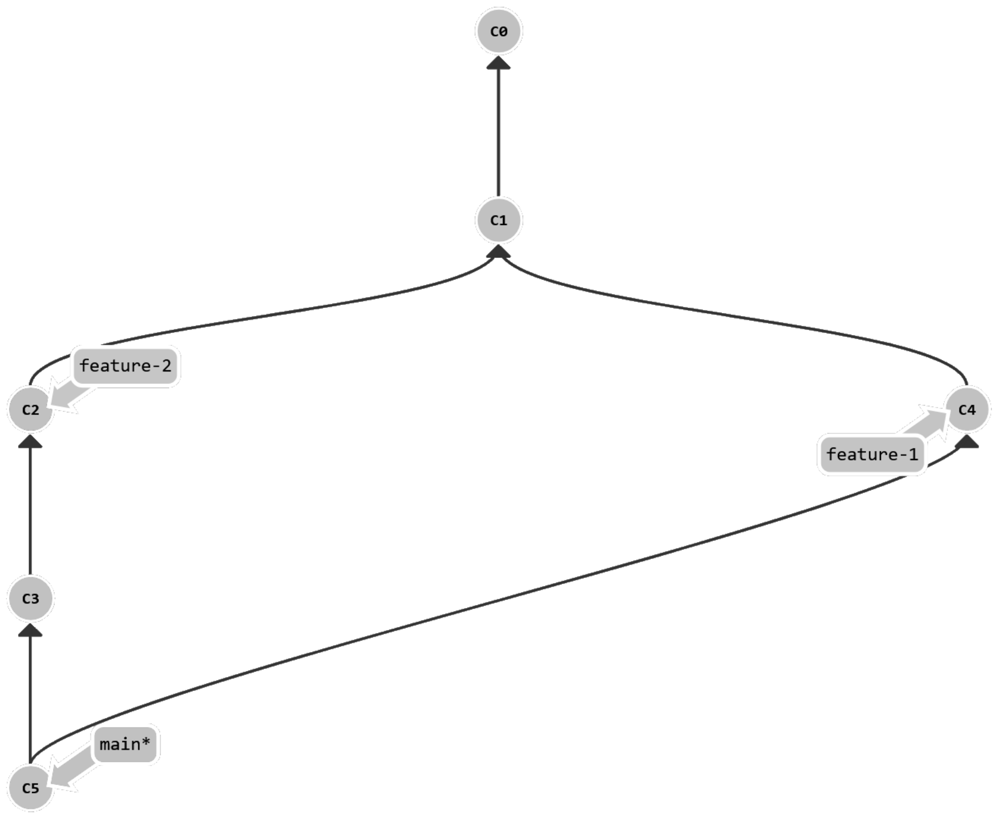

Figure 2: Git history: final state

[3 Questions — Available Marks: 35 — Full Marks: 35]

Please fill in the question numbers on the front page of your script.

<vector> class

Assume that `T` is some type (eg, int). Assume the following declarations:

```
T e;
vector<T> v, v1;
vector<T>::iterator iter, iter2, beg, end;
(use vector<T>::const_iterator or vector<T>::reverse_iterator if appropriate)
int i, n, size;
bool b
```

Methods and operators

Constructors and destructors

`vector<T> v;`	Creates an empty vector of T's.
`vector<T> v(n);`	Creates vector of n default values.
`vector<T> v(n, e);`	Creates vector of n copies of e.
`vector<T> v(beg, end);`	Creates vector with elements copied from range beg..end.
`v.~vector<T>();`	Destroys all elems and frees memory.

Size

`i = v.`**`size`**`();`	Number of elements.
`i = v.`**`capacity`**`();`	Max number of elements before reallocation.
`i = v.`**`max_size`**`();`	Implementation max number of elements.
`b = v.`**`empty`**`();`	True if empty. Same as `v.size()==0`
`v.`**`reserve`**`(`*`size`*`);`	Increases capacity to `size` before reallocation

Altering

`v = v1;`	Assigns *v1* to *v*.
`v`**`[`**`i`**`]`**` = e;`	Sets ith element. Subscripts from zero.
`v.`**`at`**`(i)= e;`	As subscription, but may throw `out_of_range`.
`v.`**`push_back`**`(e);`	Adds e to end of v. Expands v if necessary.
`v.`**`pop_back`**`();`	Removes last element of v.
`v.`**`clear`**`();`	Removes all elements.
`v.`**`assign`**`(n, e);`	Replaces existing elements with n copies of e.
`v.`**`assign`**`(beg, end);`	Replaces existing elements with copies from range beg..end.
`iter2 = v.`**`insert`**`(iter, e);`	Inserts a copy of e at iter position and returns its position.
`v.`**`insert`**`(iter, n, e);`	Inserts n copies of e starting at iter position.
`v.`**`insert`**`(iter, beg, end);`	Inserts all elements in range beg..end, starting at iter position.
`iter2 = v.`**`erase`**`(iter);`	Removes element at iter position and returns position of next element.
`iter2 = v.`**`erase`**`(beg, end);`	Removes range beg..end and returns position of next element.

Access

`e = v`**`[`**`i`**`]`**`;`	ith element. No range checking.
`e = v.`**`at`**`(i);`	As subscription, but may throw `out_of_range`.
`e = v.`**`front`**`();`	First element. No range checking.
`e = v.`**`back`**`();`	Last element. No range checking.

Iterators

`beg = v.`**`begin`**`();`	Returns iterator to first element.
`end = v.`**`end`**`();`	Returns iterator to *after* last element.
`beg = v.`**`rbegin`**`();`	Returns reverse iterator to first (in reverse order) element.

`end = v.rend();`	Returns reverse iterator to *after* last (in reverse order) element.

<string> class

Assume the following declarations:

```
string s, s1, s2;
char c;  char* cs;
string::size_type i, start, len, start1, len1, start2, len2, pos, newSize;
```

Methods and operators

Constructors and destructors

`string s;`	Creates a string variable.
`string s(s1);`	Creates s; initial value from s1.
`string s(cs);`	Creates s; initial value from cs.

Altering

`s1 = s2;`	Assigns *s2* to *s1*.
`s1 = cs;`	Assigns C-string *cs* to *s1*.
`s1 = c;`	Assigns char *c* to *s1*.
`s[i] = c;`	Sets ith character. Subscripts from zero.
`s.at(i) = c;`	As subscription, but throws `out_of_range` if *i* isn't in string.
`s.append(s2);`	Concatenates s2 on end of s. Same as s += s2;
`s.append(cs);`	Concatenates cs on end of s. Same as s += cs;
`s.assign(s2, start, len);`	Assigns s2[start..start+len-1] to s.
`s.clear();`	Removes all characters from s
`s.insert(start,s1);`	Inserts s1 into s starting at position *start*.
`s.erase(start,len);`	Deletes a substring from s. The substring starts at position *start* and is *len* characters in length.

Access

`cs = s.c_str();`	Returns the equivalent c-string.
`s1 = s.substr(start, len);`	s[start..start+len-1].
`c = s[i];`	ith character. Subscripts start at zero.
`c = s.at(i);`	As subscription, but throws `out_of_range` if *i* isn't in string.

Size

`i = s.length();`	Returns the length of the string.
`i = s.size();`	Same as s.length()
`i = s.capacity();`	Number of characters s can contain without reallocation.
`b = s.empty();`	True if empty, else false.
`i = s.resize(newSize, padChar);`	Changes size to *newSize*, padding with *padChar* if necessary.

Searching

All searches return `string::npos` on failure. The `pos` argument specifies the starting position for the search, which proceeds towards the end of the string (for "first" searches) or towards the beginning of the string (for "last" searches); if `pos` is not specified then the whole string is searched by default.

`i = s.find(c, pos);`	Position of leftmost occurrence of char *c*.
`i = s.find(s1, pos);`	Position of leftmost occurrence of *s1*.
`i = s.rfind(s1, pos);`	As find, but right to left.
`i = s.find_first_of(s1, pos);`	Position of first char in s which is in s1 set of chars.
`i = s.find_first_not_of(s1, pos);`	Position of first char of s not in s1 set of chars.
`i = s.find_last_of(s1, pos);`	Position of last char of s in s1 set of chars.
`i = s.find_last_not_of(s1, pos);`	Position of last char of s not in s1 set of chars.

Comparison

`i = s.compare(s1);`	<0 if s<s1, 0 if s==s1, or >0 if s>s1.
`i = s.compare(start1, len1, s1, start2, len2);`	Compares s[start1..start1+len1-1] to s1[start2..start2+len2-1]. Returns value as above.
`b = s1 == s2` also `> < >= <= !=`	The comparison operators work as expected.

Input / Output

`cin >> s;`	>> overloaded for string input.
`getline(cin, s);`	Next line (without newline) into s.
`cout << s;`	<< overloaded for string output.

Concatenation

The + operator is overloaded to concatentate two strings.

```
s = s1 + s2;
```

Similarly the += operator is overloaded to append strings.

```
s += s2;
s += cs;
s += c;
```

SCHOOL OF ELECTRICAL AND INFORMATION ENGINEERING
University of the Witwatersrand, Johannesburg
Software Development II

Reference Sheets

1 doctest Framework

```
TEST_CASE("This is a test case")
{
    CHECK(expression);          // assertion passes if expression is true
    CHECK_FALSE(expression);    // assertion passes if expression is false
    CHECK_THROWS(expression);   // assertion passes if an exception of any
        type is thrown by expression
    CHECK_NOTHROW(expression);  // assertion passes if no exception is thrown
        by expression
    CHECK_THROWS_AS(expression, exception_type); // assertion passes if
        expression throws an exception of exception_type
    CHECK(doctest::Approx(left) == right);  // assertion passes if left is
        approximately equal to right (floating point comparison)
}
```

Listing 1: doctest framework: syntax and assertions

2 Algorithms

The following tables provide information on some of the algorithms which are available in `<algorithm>`. Arguments which are repeatedly used in the function signatures are explained below. All of this information has been adapted from: `http://www.cplusplus.com/reference/algorithm/`.

Function Arguments	
`first` and `last`	Represent a pair of iterators which specify a range. The range specified is `[first,last)`, which contains all the elements between `first` and `last`, including the element pointed to by `first` but not the element pointed to by `last`.
`result`	Represents an iterator pointing to the start of the output range.
`val`, `old_value`, `new_value`	Represent elements which are of the same type as those contained in the range.
`pred`	Represents a function which accepts an element in the range as its only argument. The function returns either `true` or `false` indicating whether the element fulfills the condition that is checked. The function shall not modify its argument. `pred` can either be a function pointer or a function object.

Non-Modifying Sequence Operations	
`all_of(first, last, pred)`	Returns `true` if `pred` returns `true` for all the elements in the specified range or if the range is empty, and `false` otherwise.
`any_of(first, last, pred)`	Returns `true` if `pred` returns `true` for any of the elements in the specified range, and `false` otherwise.
`none_of(first, last, pred)`	Returns `true` if `pred` returns false for all the elements in the specified range or if the range is empty, and `false` otherwise.
`for_each(first, last, fn)`	Applies function `fn` to each of the elements in the specified range. `fn` accepts an element in the range as its argument. Its return value, if any, is ignored. `fn` can either be a function pointer or a function object.
`find(first, last, val)`	Returns an iterator to the first element in the specified range that compares equal to `val`. If no such element is found, the function returns `last`. The function uses `operator==` to compare the individual elements to `val`.
`find_if(first, last, pred)`	Returns an iterator to the first element in the specified range for which `pred` returns `true`. If no such element is found, the function returns `last`.
`find_first_of(first1, last1, first2, last2)`	Returns an iterator to the first element in the range [`first1`,`last1`) that matches any of the elements in [`first2`,`last2`). If no such element is found, the function returns `last1`. The elements in [`first1`,`last1`) are sequentially compared to each of the values in [`first2`,`last2`) using `operator==`.
`count(first, last, val)`	Returns the number of elements in the specified range that compare equal to `val`. The function uses `operator==` to compare the individual elements to `val`.
`equal(first1, last1, first2)`	Compares the elements in the range [`first1`,`last1`) with those in the range beginning at `first2`, and returns `true` if all of the elements in both ranges match, and `false` otherwise. The elements are compared using `operator==`.
`search_n(first, last, count, val)`	Searches the specified range for a sequence of successive `count` elements, each comparing equal to `val`. The function returns an iterator to the first of such elements, or `last` if no such sequence is found.
`binary_search(first, last, val)`	Returns `true` if any element in the specified range is equivalent to `val`, and `false` otherwise. The elements are compared using `operator<`. Two elements, `a` and `b` are considered equivalent if `(!(a<b) && !(b<a))`. The elements in the range *shall already be sorted* according to this same criterion (operator<). The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is especially efficient for random-access iterators.
`min_element(first, last)`	Returns an iterator pointing to the element with the smallest value in the specified range. The comparisons are performed using `operator<`. An element is the smallest if no other element compares less than it. If more than one element fulfills this condition, the iterator returned points to the first of such elements.
`max_element(first, last)`	Returns an iterator pointing to the element with the largest value in the specified range. The comparisons are performed using `operator<`. An element is the largest if no other element does not compare less than it. If more than one element fulfills this condition, the iterator returned points to the first of such elements.

Modifying Sequence Operations	
`copy(first, last, result)`	Copies the elements in the range [`first`,`last`) into the range beginning at `result`. The function returns an iterator to the end of the destination range (which points to the element following the `last` element copied). The ranges shall not overlap in such a way that `result` points to an element in the range [`first`,`last`).
`transform(first, last, result, op)`	Applies the function `op` to each of the elements in the specified range and stores the value returned by `op` in the range that begins at `result`. `op` can either be a function pointer or a function object. The `transform` function allows for the destination range to be the same as the input range to make transformations *in place*. `transform` returns an iterator pointing to the element that follows the `last` element written in the `result` sequence.
`replace(first, last, old_value, new_value)`	Assigns `new_value` to all the elements in the specified range that compare equal to `old_value`. The function uses `operator==` to compare the individual elements to `old_value`. No value is returned.
`replace_if(first, last, pred, new_value)`	Assigns `new_value` to all the elements in the specified range for which `pred` returns `true`. No value is returned.
`fill(first, last, val)`	Assigns `val` to all the elements in the specified range. No value is returned.
`remove(first, last, val)`	Transforms the specified range into a range with all the elements that compare equal to `val` removed, and returns an iterator to the new end of that range. The function does not alter the size of the container containing the range of elements. The removal is done by replacing the elements that compare equal to `val` by the next element that does not, and signalling the new size of the shortened range by returning an iterator to the element that should be considered its new past-the-end element. The relative order of the elements not removed is preserved, while the elements between the returned iterator and `last` are left in a valid but unspecified state. The function uses `operator==` to compare the individual elements to `val`.
`remove_if(first, last, pred)`	Transforms the specified range into a range with all the elements for which `pred` returns `true` removed, and returns an iterator to the new end of that range. The function does not alter the size of the container containing the range of elements. The removal is done by replacing the elements for which `pred` returns `true` by the next element that does not, and signalling the new size of the shortened range by returning an iterator to the element that should be considered its new past-the-end element. The relative order of the elements not removed is preserved, while the elements between the returned iterator and `last` are left in a valid but unspecified state.
`unique(first, last)`	Removes all but the first element from every consecutive group of equivalent elements in the specified range. The function does not alter the size of the container containing the range of elements. The removal is done by replacing the duplicate elements by the next element that is not a duplicate, and signalling the new size of the shortened range by returning an iterator to the element that should be considered its new past-the-end element. The relative order of the elements not removed is preserved, while the elements between the returned iterator and `last` are left in a valid but unspecified state. The function uses `operator==` to compare the pairs of elements.

Modifying Sequence Operations	
`reverse(first, last)`	Reverses the order of the elements in the specified range. There is no return value.
`sort(first, last)`	Sorts the elements in the specified range into ascending order. The elements are compared using `operator<`. There is no return value.

<locale> Members

The <locale> header file includes functions for character classification. These are listed below.

`bool isalnum(char c)`	Returns true if the character tested is alphanumeric; false if it is not.
`bool isalpha(char c)`	Returns true if the character tested is alphabetic; false if it is not.
`bool iscntrl(char c)`	Returns true if the character tested is a control character; false if it is not.
`bool isdigit(char c)`	Returns true if the character tested is a numeric; false if it is not.
`bool isgraph(char c)`	Returns true if the character tested is alphanumeric or a punctuation character; false if it is not.
`bool isupper(char c)`	Returns true if the character tested is uppercase; false if it is not.
`bool islower(char c)`	Returns true if the character tested is lowercase; false if it is not.
`bool isprint(char c)`	Returns true if the character tested is a printable; false if it is not.
`bool ispunct(char c)`	Returns true if the character tested is a punctuation character; false if it is not.
`bool isspace(char c)`	Returns true if the character tested is a whitespace; false if it is not.
`bool isxdigit(char c)`	Returns true if the character tested is a character used to represent a hexadecimal number; false if it is not.
`char tolower(char c)`	Returns the character converted to lower case.
`char toupper(char c)`	Returns the character converted to upper case.