Started on	Tuesday, 7 February 2023, 11:15 PM
State	Finished
Completed on	Tuesday, 7 February 2023, 11:36 PM
Time taken	21 mins 47 secs
Grade	**100.00** out of 100.00

Information

Don't choose "doesn't compile" because standard library headers are missing from certain code fragments. Instead, evaluate and analyze these code fragments [they're called fragments not programs] in terms of the behavior and state of variables.

Question **1**

Correct

Mark 2.00 out of 2.00

Given these declarations of function `foo`:

```
void foo(int);      // A
int foo(int);       // B
double foo(int);    // C
double foo(double); // D
```

which overload of function `foo` will be invoked in the following code fragment? Choose labels associated with each function declaration to identify a specific function. Choose NC if the overload resolution fails.

```
void boo() {
  double d = foo(5);
}
```

Select one:

- A
- NC ✔
- C
- B
- D

Your answer is correct.

Functions labeled `A`, `B`, and `C` have the same signatures but different return types. These declarations are considered illegal and therefore the answer is NC.

The correct answer is: NC

Question **2**

Correct

Mark 6.00 out of 6.00

Do the following in the specified order:

1. Define functions **add**, **subtract**, **multiply**, and **divide** that take two **int** parameters and return a corresponding **int** value. That is, function **add** will return the sum of its two parameters; function **subtract** will return the value obtained by subtracting the second parameter from the first parameter; and so on.
2. Provide an alias declaration that declares name **PTRFN** as an alias for type ***pointer*** to the type specified by any of the four functions defined earlier: **add**, **subtract**, **multiply**, and **divide**.
3. Define function **create_calc** that returns a **std::vector** container containing pointers to functions **add**, **subtract**, **multiply**, and **divide** [in that order].

DON'T ADD STANDARD LIBRARY HEADERS - ALL NECESSARY HEADERS ARE INCLUDED!!!

```
// Don't include any standard library headers - the necessary ones are included for you ...
// Implement the three steps in the order described in the specs ...
// 1
int add(int lhs, int rhs) { return lhs+rhs; }
int subtract(int lhs, int rhs) { return lhs-rhs; }
int multiply(int lhs, int rhs) { return lhs*rhs; }
int divide(int lhs, int rhs) { return lhs/rhs; }
// 2
using PTRFN = int (*)(int, int);
// 3
std::vector<PTRFN> create_calc() {
return std::vector{add, subtract, multiply, divide};
}
```

Correct

Evaluation details:

Evaluation:

-Summary of tests

```
>+----------------------------+
>|  6 tests run/ 6 tests passed |
>+----------------------------+
```

```
// 1
int add(int lhs, int rhs)      { return lhs+rhs; }
int subtract(int lhs, int rhs) { return lhs-rhs; }
int multiply(int lhs, int rhs) { return lhs*rhs; }
int divide(int lhs, int rhs)   { return lhs/rhs; }

// 2
using PTRFN = int (*)(int, int);

// 3
std::vector<PTRFN> create_calc() {
  return std::vector{add, subtract, multiply, divide};
}
```

Possible solution:

```
// 1
int add(int lhs, int rhs)      { return lhs+rhs; }
int subtract(int lhs, int rhs) { return lhs-rhs; }
int multiply(int lhs, int rhs) { return lhs*rhs; }
int divide(int lhs, int rhs)   { return lhs/rhs; }

// 2
using PTRFN = int (*)(int, int);

// 3
std::vector<int (*)(int, int)> create_calc() {
  return std::vector{add, subtract, multiply, divide};
}
```

Question **3**

Correct

Mark 2.00 out of 2.00

Write the exact values [without extraneous whitespace characters] written to the standard output stream by the following code fragment.

```
// in function main ...
std::string s ("0123456789");
std::cout << s.substr(6);
```

Answer: 6789 ✔

The correct answer is: 6789

Question **4**

Correct

Mark 3.00 out of 3.00

Given declaration:

```
char* init(int ht = 640, int wd = 480, char bgnd = '@');
```

what are the values of the three arguments in the call to function **init**:

```
char *cursor = init('#');
```

Argument 1: '#' ✔

Argument 2: 480 ✔

Argument 3: '@' ✔

Your answer is correct.

The call is legal because **'#'** is a **char** with an ASCII value **35**, and a **char** can be converted to the type of the left-most parameter. That parameter is an **int** type. In this call, the **char** argument is implicitly converted to **int** and the value **35** is passed as the argument to parameter **ht**. Since the call specifies no other arguments, the compiler will use the default parameter value **480** for parameter **wd** and the default parameter value **'@'** for parameter **bgnd**.

The correct answer is: Argument 1: → '#', Argument 2: → 480, Argument 3: → '@'

Question **5**

Correct

Mark 10.00 out of 10.00

Define function `unique_words` that parses a `std::string` parameter for "words" and returns a `std::vector` container containing unique words that are lexicographically sorted in ascending order. A "word" is delimited by any of these ASCII whitespace characters: `' '`, `'\t'`, `'\n'`, `'\r'` [carriage return], `'\v'` [vertical tab], `'\f'` [formfeed]. **Don't include any header files - all necessary header files [including `<algorithm>`] are included!!!**

Here's a use case:

```
std::string const line {"    today is a good tomorrow will be a better day    "};
std::vector<std::string> total_words = unique_words(line);
std::cout << total_words.size() << ": ";
for (std::string const& x : total_words) {
  std::cout << x << ' ';
}
std::cout << "\n";
```

will print the following text to standard output:

```
9: a be better day good is today tomorrow will
```

A second use case:

```
std::string const line {"        "};
std::vector<std::string> total_words = unique_words(line);
std::cout << total_words.size() << ": ";
for (std::string const& x : total_words) {
  std::cout << x << ' ';
}
std::cout << "\n";
```

will print the following text to standard output:

```
0:
```

```
7  // Do not include any standard library headers - the necessary header files [including <algorithm>] are already included!!!
8  // Define function unique_words here.
9  std::vector<std::string> unique_words(std::string const& line) {
10 // there are six possible whitespace characters in ASCII that delimit a "word"
11 static std::string const whitespace{" \t\n\v\r\f"};
12 std::vector<std::string> words;
13 std::string::size_type i{0};
14 while (i < line.size()) {
15 // ignore leading whitespace characters ...
16 i = line.find_first_not_of(whitespace, i);
17 // find end of next word ...
18 std::string::size_type j = line.find_first_of(whitespace, i);
19 // if we found some non-whitespace characters, we've a new "word"
20 if (i != j) {
```

Correct

Evaluation details:

Evaluation:

-Summary of tests

```
>+------------------------------+
>|  3 tests run/ 3 tests passed |
>+------------------------------+
```

```
std::vector<std::string> unique_words(std::string const& line) {
  // there are six possible whitespace characters in ASCII that delimit a "word"
  static std::string const whitespace{" \t\n\v\r\f"};
  std::vector<std::string> words;
  std::string::size_type i{0};
  while (i < line.size()) {
    // ignore leading whitespace characters ...
    i = line.find_first_not_of(whitespace, i);
    // find end of next word ...
    std::string::size_type j = line.find_first_of(whitespace, i);
    // if we found some non-whitespace characters, we've a new "word"
    if (i != j) {
      words.push_back(line.substr(i, j-i));
      i = j;
    }
  }

  std::sort(std::begin(words), std::end(words));
  std::vector<std::string>::iterator unique_it = std::unique(std::begin(words), std::end(words));
  words.erase(unique_it, std::end(words));
  return words;
}
```

Possible solution:

```
std::vector<std::string> unique_words(std::string const& line) {
  // there are six possible whitespace characters in ASCII that delimit a "word"
  static std::string const whitespace{" \t\n\v\r\f"};
  std::vector<std::string> words;
  std::string::size_type i{0};
  while (i < line.size()) {
    // ignore leading whitespace characters ...
    i = line.find_first_not_of(whitespace, i);
    // find end of next word ...
    std::string::size_type j = line.find_first_of(whitespace, i);
    // if we found some non-whitespace characters, we've a new "word"
    if (i != j) {
      words.push_back(line.substr(i, j-i));
      i = j;
    }
  }

  std::sort(std::begin(words), std::end(words));
  std::vector<std::string>::iterator unique_it = std::unique(std::begin(words), std::end(words));
  words.erase(unique_it, std::end(words));
  return words;
}
```

Question **6**

Correct

Mark 7.00 out of 7.00

This program takes command-line arguments and the program's function `main` calls function `foo` with the same arguments supplied to function `main`. Define function `foo` so that it returns a value of type `std::string` containing the concatenated command-line arguments except for the name of the program. Add a single space character after each command-line argument except the last argument. For example, if the program is called like this:

```
./a.out 1 2 3
```

function `foo` will return a `std::string` value encapsulating string `"1 2 3"`.

If the program is called like this:

```
./a.out today is a good day
```

function foo will return a `std::string` value encapsulating string `"today is a good day"`.

DON'T INCLUDE ANY STANDARD HEADERS - THE NECESSARY ONES ARE INCLUDED FOR YOU!!!

```
7  // Don't include any standard library headers - the necessary ones are included for you!!!
8  // Define function foo here.
9  std::string foo(int argc, char *argv[]) {
10 std::string cat;
11 for (int i{1}; i < argc; ++i) {
12 cat += argv[i];
13 cat += (i == argc-1) ? "" : " ";
14 }
15 return cat;
16 }
```

Correct

Evaluation details:

Evaluation:

-Summary of tests

```
>+------------------------------+
>|  3 tests run/ 3 tests passed |
>+------------------------------+
```

```
std::string foo(int argc, char *argv[]) {
  std::string cat;
  for (int i{1}; i < argc; ++i) {
    cat += argv[i];
    cat += (i == argc-1) ? "" : " ";
  }
  return cat;
}
```

Possible solution:

```
std::string foo(int argc, char *argv[]) {
  std::string cat;
  for (int i{1}; i < argc; ++i) {
    cat += argv[i];
    cat += (i == argc-1) ? "" : " ";
  }
  return cat;
}
```

Question **7**

Correct

Mark 2.00 out of 2.00

What is printed to standard output stream by the following code fragment? If the code doesn't compile, write NC as your answer.

```
class A {
  int n;
public:
  A(int m) : n(m) { std::cout << "A";  }
  ~A()            { std::cout << "~A"; }
  int N() const   { return n; }
};

void foo(A const& a) {
  std::cout << a.N();
}

int main() {
  foo(5);
}
```

Answer: A5~A ✔

Consider the call to function `foo`:

```
foo(5);
```

Argument `5` in the call to `foo` will be used as the argument to constructor `A::A(int)` that will initialize an unnamed, temporary object of type `A` on the stack. This will cause character `'A'` to be printed to standard output. Further, `foo`'s reference parameter `a` is initialized to reference this unnamed, temporary object. In function `foo`, the value of data member `n` of the object referenced by `a` - which is `5` - will be printed to standard output. When function `foo` terminates, the unnamed, temporary object will go out of scope causing the automatic execution of its destructor. The destructor's body will write C-style string `"~A"` to standard output stream.

The correct answer is: A5~A

Question **8**

Correct

Mark 2.00 out of 2.00

Write the exact values [without extraneous whitespace characters] written to the standard output stream by the following code fragment. Type NC if the code fragment does not compile.

```
int main() {
  char& b;
  int c{10};
  b = c;
  b = 20;
  std::cout << c;
}
```

Answer: NC ✔

The correct answer is: NC

Question **9**

Correct

Mark 2.00 out of 2.00

Write the exact values [without extraneous whitespace characters] written to standard output stream by the following code fragment. Write NC if the code doesn't compile.

```
// in function main() ....
std::string const s ("Hello world!");
for (char& ch : s)
  ch = std::tolower(ch);
std::cout <<s;
```

Answer: NC ✔

The correct answer is: NC

Question **10**

Correct

Mark 2.00 out of 2.00

The Battleship game requires a 2D grid to represent an ocean. The purpose of the partially defined function `create_ocean` is to dynamically allocate such a 2D grid with `M` columns [$x-$extent] and `N` rows [$y-$extent]. Unlike the lab assignment, we wish to access each element of the 2D grid using 2D array syntax. For example, an ocean location with coordinates (x, y) is accessed as `ocean[y][x]`. Complete the definition of the partially implemented function `create_ocean`. **Don't include any standard library headers - the necessary headers are included and trying to include other will prevent your code from compiling!!!**

```
4  // x-size (columns): M, y-size (rows) : N
5  // x-size (columns): M, y-size (rows) : N
6  int** create_ocean(size_t M, size_t N) {
7  int **pp { new int* [N] };
8  for (size_t i{0}; i < N; ++i) { pp[i] = new int [M]; }
9  return pp;
10 }
11
```

Correct

Evaluation details:

```
Evaluation:
-Summary of tests
>+------------------------------+
>|  1  test run/ 1  test passed |
>+------------------------------+
```

```
// x-size (columns): M, y-size (rows) : N
int** create_ocean(size_t M, size_t N) {
  int **pp { new int* [N] };
  for (size_t i{0}; i < N; ++i) { pp[i] = new int [M]; }
  return pp;
}
```

Possible solution:

```
// x-size (columns): M, y-size (rows) : N
int** create_ocean(size_t M, size_t N) {
  int **pp { new int* [N] };
  for (size_t i{0}; i < N; ++i) { pp[i] = new int [M]; }
  return pp;
}
```

Question **11**

Correct

Mark 5.00 out of 5.00

Define function `create_node` that takes a data value and returns a pointer to a dynamically allocated `Node` object.

```
struct Node {
  int value;
  struct Node *next;
};

// Don't include any headers - your code will not compile!!!
// Now, define function create_node
Node* create_node(int v) {
return new Node {v, nullptr};
}
```

Correct

Evaluation details:

Evaluation:

-Summary of tests

```
>+------------------------------+
>|  1  test run/ 1  test passed |
>+------------------------------+
```

```
struct Node {
  int value;
  struct Node *next;
};

Node* create_node(int v) {
  return new Node {v, nullptr};
}
```

Possible solution:

```
struct Node {
  int value;
  struct Node *next;
};

Node* create_node(int v) {
  return new Node {v, nullptr};
}
```

Question **12**

Correct

Mark 5.00 out of 5.00

Which of the following can be used as a default function argument?

Select one or more:

- [x] variables declared in anonymous namespaces ✔
- [x] global variables ✔
- [x] literals ✔
- [x] heap objects as in `void foo(int a = *(new int{10}));` ✔
- [ ] local variables
- [x] function call (as long as the function being called is declared before its use in the declaration) ✔

The correct answers are: global variables, function call (as long as the function being called is declared before its use in the declaration), literals, heap objects as in `void foo(int a = *(new int{10}));`, variables declared in anonymous namespaces

Question **13**

Correct

Mark 4.00 out of 4.00

Which, if any, of the following definitions are illegal? Use the label on the declaration statement to indicate an illegal definition.

```
int i = 1.01, *pi = &i;
int& ri = 1.01;     // A
int &ri1, i2;       // B
int &ri2 = i;       // C
int* &ri3 = &i;     // D
int &ri4 = pi;      // E
int& ri5 = *pi;     // F
int& *pri = pi;     // G
const int& rci = 1; // H
```

Select one or more:

- [x] A ✔
- [x] B ✔
- [ ] C
- [x] D ✔
- [x] E ✔
- [ ] F
- [x] G ✔
- [ ] H

Your answer is correct.

The correct answers are: A, B, D, E, G

Question **14**

Correct

Mark 2.00 out of 2.00

Write the values printed to standard output stream by the following code fragment.

```
struct Obj {
  Obj operator+(Obj const&) { std::cout << "binary+"; return Obj(); }
  Obj operator+()           { std::cout << "unary+"; return Obj(); }
};

Obj operator*(Obj const& x, Obj const& y) {
  std::cout << "binary-mul";
  return Obj();
}

// in function main ...
Obj o1, o2;
o1+o2;
+o2;
o1*o2;
```

Answer: binary+unary+binary-mul ✔

The correct answer is: binary+unary+binary-mul

Question **15**

Correct

Mark 5.00 out of 5.00

Define a function `foo` to swap two parameters of type `int*`. **Do not include any header files!!!**

```
// NOTE: Including standard library headers will prevent your code from compiling!!!
// Define function foo with required parameters here.
void foo(int* &lhs, int* &rhs) {
int *ptmp {lhs};
lhs = rhs;
rhs = ptmp;
}
```

Correct

Evaluation details:

Evaluation:

-Summary of tests

```
>+------------------------------+
>|  2 tests run/ 2 tests passed |
>+------------------------------+
```

```
void foo(int* &lhs, int* &rhs) {
  int *ptmp {lhs};
  lhs = rhs;
  rhs = ptmp;
}
```

Possible solution:

```
void foo(int* &lhs, int* &rhs) {
  int *ptmp {lhs};
  lhs = rhs;
  rhs = ptmp;
}
```

Question **16**

Correct

Mark 2.00 out of 2.00

The *copy constructor* is a constructor which creates an object by initializing it with an object of the same class which has been created previously.

If a copy constructor is not defined in a class, the compiler itself defines "shallow" or member-wise copy constructor. This means that each member of the class individually. When classes are simple (e.g. do not contain any dynamically allocated memory), this works very well. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a custom made copy constructor for the class.

Given the definition of type `A`:

```
class A {
  int n;
public:
  A() {n = 10;}                   // default ctor
  A(A const& a) { n = a.n + 10; } // copy ctor
  int N() const { return n; }     // accessor
};
```

what is the output of the following code fragment? Write NC if the code fragment doesn't compile.

```
// in function main ...
A a, b(a);
std::cout << a.N() << ',' << b.N() << ',' << A(b).N();
```

Answer: 10,20,30 ✔

In the statement

```
std::cout << a.N() << ',' << b.N() << ',' << A(b).N();
```

the expression `A(b).N()` evaluates to an unnamed temporary object which calls member function `N`.

The correct answer is: 10,20,30

Question **17**

Correct

Mark 2.00 out of 2.00

Which, if any, of the following definitions are illegal?

```
int i = -1;
const int i2 = i;            // A
const int * pi1 = &i;        // B
int * const pi2 = &i2;       // C
const int * const pi3 = &i2; // D
```

Select one or more:

- [ ] A
- [ ] B
- [x] C ✔
- [ ] D

Your answer is correct.

The correct answer is: C

Question **18**

Correct

Mark 2.00 out of 2.00

We wish to define function `square` as an `inline` function. Which of the following definitions is syntactically correct?

Select one:

- `double square(double x) { return x * x; }`
- `double square(double x) { return x * x; } inline`
- `inline double square(double x) { return x * x; }` ✔
- `double square(double x) inline { return x * x; }`

Your answer is correct.

The correct answer is: `inline double square(double x) { return x * x; }`

Question **19**

Correct

Mark 5.00 out of 5.00

Define function `find_char` that returns the position of the first occurrence of a given character in a `std::string`. and updates an out parameter with the count of occurrences of the character. **Don't include standard library headers - all necessary header files [except `<algorithm>`] are included.** Here are some use cases:

```
std::string const s{"aeiouAEIOU"};
int count;

// find_char will return std::string::npos and count is 0
std::string::size_type pos {find_char(s, 'Z', count)};

// find_char will return 0 and count is 1
pos = find_char(s, 'a', count);

// find_char will return 1 and count is 2
pos = find_char("seattle", 'e', count);
```

```
4  // NOTE: Don't include any standard library headers - all necessary headers [except <algorithm>] are already included!!!
5  // Define function find_char here ...
6  // Version that uses string's method find_first_of
7  std::string::size_type find_char(std::string const& s, char to_srch, int& cnt) {
8  cnt = 0;
9  for (char ch : s) {
10 cnt = (ch == to_srch) ? cnt+1 : cnt;
11 }
12 return s.find_first_of(to_srch);
13 }
```

Correct

Evaluation details:

Evaluation:

-Summary of tests

```
>+------------------------------+
>|  4 tests run/ 4 tests passed |
>+------------------------------+
```

```
// Version that uses string's method find_first_of
std::string::size_type find_char(std::string const& s, char to_srch, int& cnt) {
  cnt = 0;
  for (char ch : s) {
    cnt = (ch == to_srch) ? cnt+1 : cnt;
  }
  return s.find_first_of(to_srch);
}
```

Possible solution:

```
// Version that uses string's method find_first_of
std::string::size_type find_char(std::string const& s, char to_srch, int& cnt) {
  cnt = 0;
  for (char ch : s) {
    cnt = (ch == to_srch) ? cnt+1 : cnt;
  }
  return s.find_first_of(to_srch);
}
```

Question **20**

Correct

Mark 1.00 out of 1.00

The synthesized default constructor for the following class does nothing.

```
class Y {
public:
  // no constructors are declared
private:
  int  i;
  std::string s;
  std::vector<int> v;
};
```

Select one:

- True
- False ✔

A constructor has an ***initialization part*** and a ***function body***. The *initialization part* initializes each data member before the *function body* is executed, and members are initialized in the same order as they are declared in the class. The initialization part uses a *constructor member initializer list* to initialize data members. If the programmer doesn't author a constructor member initializer list, the compiler will synthesize a constructor member initializer list that will default initialize the data members. Default initialization of a data member of built-in type means the data member will have an unspecified value. Default initialization of a data member of user-defined type will take place by calling its default constructor or by synthesizing a default constructor for the data member. If the user-defined type doesn't define a default constructor and the compiler is unable to synthesize one, the compiler will flag an error.

Consider the following class `Y` that doesn't declare any constructors:

```
class Y {
public:
  // no constructors are declared
private:
  int  i;
  std::string s;
  std::vector<int> v;
};
```

When an object of type `Y` is defined

```
Y y;
```

the compiler will synthesize a default constructor that will have an initialization part and a function body. The initialization part will consist of the following ordered steps:

1. Default initialize data member `i` which means nothing is done since `i` is a built-in data type.
2. Default initialize data member `s` using default constructor `std::string::string()` of class `std::string`.
3. Default initialize data member `v` using default constructor `std::vector<int>::vector<int>()` of class `std::vector<int>`.

The function body of the synthesized default constructor will be empty.

Now, consider the following definitions of classes `X` and `Y`:

```
class X {
public:
  X(double x);
  // no other constructors are declared
private:
  double d;
};

class Y {
public:
  // no constructors nor a destructor are declared
private:
  int  i;
  std::string s;
  std::vector<int> v;
  X x;
};
```

When an object of type `Y` is defined

```
Y y;
```

the compiler will synthesize a default constructor that will have an initialization part and a function body. The initialization part will consist of the following ordered steps:

1. Default initialize data member `i` with an unspecified value since `i` is a built-in data type.
2. Default initialize data member `s` using default constructor `std::string::string()` of class `std::string`.
3. Default initialize data member `v` using default constructor `std::vector<int>::vector<int>()` of class `std::vector<int>`.
4. Default initialize data member `x` using default constructor `X::X()` of user-defined type `X`. However, class `X` doesn't declare a default constructor. In addition, the compiler is unable to synthesize a default constructor because class `X` declares a constructor `X::X(double)`.

Since the compiler is unable to default initialize data member `x`, the compiler will flag the definition

```
Y y;
```

as a compile-time error.

Main takeaway: The constructor body doesn't directly initialize the members themselves. Members are initialized as part of the initialization phase that precedes the constructor body. A constructor body executes in addition to the member-wise initialization that takes place as part of an object's initialization. In short, if a constructor body is empty, it is not true that the constructor is not doing anything.

The correct answer is 'False'.

Question **21**

Correct

Mark 15.00 out of 15.00

Consider the partial definition of type `StopWatch`:

```
class StopWatch {
public:
  StopWatch();             // default ctor: to be defined by you
  StopWatch(int seconds);  // conversion ctor: to be defined by you
  StopWatch(int hours, int minutes, int seconds); // non-default ctor: to be defined by you

  // accessor and mutator
  int Seconds() const;    // to be defined by you
  void Seconds(int secs); // to be defined by you

  // Missing declarations: 10 operator overload member functions are missing!!!
  // These 10 member functions are declared by me but you can't see these declarations!!!
  // This is an exercise in understanding class design and implementation
  // exactly as explained in class lectures and lab

  // tell compiler and other programmers that you're using default semantics
  StopWatch(StopWatch const&) = default;
  StopWatch& operator=(StopWatch const&) = default;
  ~StopWatch() = default;
private:
  int seconds;
};

// Missing declarations: 10 non-member non-friend operator overload function declarations are missing!!!
// These 10 non-member non-friend functions are declared by me but you can't see them!!!
// This is an exercise in understanding class design and implementation
// exactly as explained in class lectures and lab
```

Function `main` illustrates the 25 interface methods that `StopWatch` clients require from the designers of this type. Define these 25 interface functions [3 constructors + 2 accessors/modifiers + 10 member functions that overload operators + 10 non-member, non-friend functions that overload operators]. Since you don't have access to the definition of type `StopWatch`, you must define these functions outside the class definition. Header files `<iostream>` and `<iomanip>` are included and you can't include any other standard library headers.

```
int main() {
  StopWatch sw1(10);         // sw1 is 10 seconds
  StopWatch sw2(10, 11, 12); // sw2 is 36'672 seconds or 10:11:12
  StopWatch sw3;             // sw3 is 0 seconds

  (sw1 += 19)++;             // sw1 should be 30 seconds
  ++(sw2 -= 36663);          // sw2 is 10 seconds

  sw3 = sw1 - sw2;           // sw3 should be 20 seconds
  sw3 = sw1 + sw2;           // sw3 should be 40 seconds
  (sw3 += sw1)--;            // sw3 should be 69 seconds or 00:01:09
  (++--(sw3 -= sw1))++;      // sw3 should be 40 seconds
  sw3 = sw1 + 10;            // sw3 should be 40 seconds
  sw3 = 10 + sw1;            // sw3 should be 40 seconds
  sw3 = sw1 - 10;            // sw3 should be 20 seconds
  sw3 = 10 - sw2;            // sw3 should be 0 seconds
  sw3.Seconds(20);           // sw3 should be 20 seconds
  sw3 *= 16;                 // sw3 should be 320 seconds or 00:05:20
  sw3 *= sw1;                // sw3 should be 9600 seconds or 02:40:00
  sw3 = sw1 * sw2;           // sw3 should be 300 seconds or 00:05:00
  sw3 = sw1 * 11;            // sw3 should be 330 seconds or 00::05::30
  sw3 = 11 * sw1;            // sw3 should be 330 seconds or 00::05::30

  sw3 = ----sw2;             // sw3 and sw2 should be 8 seconds
  sw3 *= sw2*1000;           // sw3 should be 64'000 seconds or 17:46:40 hrs
  std::cout << sw3;          // prints seconds in with 8 characters to output stream: 17:46:40
}
```

```
48  // Header files <iostream> and <iomanip> are included!!! You can't include any other standard library headers ...
49  // Provide definitions of 25 functions:
50  // [3 ctors + 2 accessor/mutator + 10 member overloads + 10 non-member, non-friend overloads] ...
51  // declarations of 10 non-member, non-friend operator overloads ...
52  StopWatch operator+(StopWatch const& lhs, StopWatch const& rhs);
53  StopWatch operator+(StopWatch const& lhs, int rhs);
54  StopWatch operator+(int lhs, StopWatch const& rhs);
55  StopWatch operator-(StopWatch const& lhs, StopWatch const& rhs);
56  StopWatch operator-(StopWatch const& lhs, int rhs);
57  StopWatch operator-(int lhs, StopWatch const& rhs);
58  StopWatch operator*(StopWatch const& lhs, StopWatch const& rhs);
59  StopWatch operator*(StopWatch const& lhs, int rhs);
60  StopWatch operator*(int lhs, StopWatch const& rhs);
61  std::ostream& operator<<(std::ostream& os, StopWatch const& rhs);
```

Correct

Evaluation details:

Evaluation:

-Summary of tests

```
>+------------------------------+
>|  1  test run/ 1  test passed |
>+------------------------------+
```

```
class StopWatch {
public:
  StopWatch();            // default ctor: to be defined by you
  StopWatch(int seconds); // conversion ctor: to be defined by you
  StopWatch(int hours, int minutes, int seconds); // non-default ctor: to be defined by you

  // mutator + accessor
  void Seconds(int secs); // mutator: to be defined by you
  int Seconds() const;    // accessor: to be defined by you

  // missing declarations: 10 operator overload member functions are missing!!!
  // these 10 member functions are declared by me but you can't see these declarations!!!
  // this is an exercise in understanding class design and implemmentation
  // exactly as explained in class lectures and lab
  StopWatch& operator+=(StopWatch const&);
  StopWatch& operator+=(int);
  StopWatch& operator-=(StopWatch const&);
  StopWatch& operator-=(int);
  StopWatch& operator*=(StopWatch const&);
  StopWatch& operator*=(int);
  StopWatch& operator++();
  StopWatch const operator++(int);
  StopWatch& operator--();
  StopWatch const operator--(int);

  // tell other programmers and compiler that you want default semantics
  StopWatch(StopWatch const&) = default;
  StopWatch& operator=(StopWatch const&) = default;
  ~StopWatch() = default;
private:
       int seconds;
};

// declarations of 10 non-member, non-friend operator overloads ...
StopWatch operator+(StopWatch const& lhs, StopWatch const& rhs);
StopWatch operator+(StopWatch const& lhs, int rhs);
StopWatch operator+(int lhs, StopWatch const& rhs);
StopWatch operator-(StopWatch const& lhs, StopWatch const& rhs);
StopWatch operator-(StopWatch const& lhs, int rhs);
StopWatch operator-(int lhs, StopWatch const& rhs);
StopWatch operator*(StopWatch const& lhs, StopWatch const& rhs);
StopWatch operator*(StopWatch const& lhs, int rhs);
StopWatch operator*(int lhs, StopWatch const& rhs);
std::ostream& operator<<(std::ostream& os, StopWatch const& rhs);

// member function definitions ...

// 3 ctors
StopWatch::StopWatch() : seconds{} {}
StopWatch::StopWatch(int secs) : seconds{secs} {}
StopWatch::StopWatch(int hours, int minutes, int secs) : seconds{hours*60*60+minutes*60+secs} {}

// mutator + accessor
void StopWatch::Seconds(int secs) { seconds = secs; }
int StopWatch::Seconds() const { return seconds; }

// 10 member functions of operator overloads
StopWatch& StopWatch::operator++() { ++seconds; return *this; }
StopWatch const StopWatch::operator++(int) { StopWatch old{*this}; ++(*this); return old; }
StopWatch& StopWatch::operator--() { --seconds; return *this; }
StopWatch const StopWatch::operator--(int) { StopWatch old{*this}; --(*this); return old; }

StopWatch& StopWatch::operator+=(StopWatch const& rhs) { seconds += rhs.seconds; return *this; }
StopWatch& StopWatch::operator+=(int rhs) { seconds += rhs; return *this; }
StopWatch& StopWatch::operator-=(StopWatch const& rhs) { seconds -= rhs.seconds; return *this; }
StopWatch& StopWatch::operator-=(int rhs) { seconds -= rhs; return *this; }
StopWatch& StopWatch::operator*=(StopWatch const& rhs) { seconds *= rhs.seconds; return *this; }
StopWatch& StopWatch::operator*=(int rhs) { seconds *= rhs; return *this; }

// 10 non-member, non-friend definitions of operator overloads
StopWatch operator+(StopWatch const& lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp += rhs; return tmp; }
StopWatch operator+(StopWatch const& lhs, int rhs) { StopWatch tmp{lhs}; tmp += rhs; return tmp; }
StopWatch operator+(int lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp += rhs; return tmp; }

StopWatch operator-(StopWatch const& lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp -= rhs; return tmp; }
StopWatch operator-(StopWatch const& lhs, int rhs) { StopWatch tmp{lhs}; tmp -= rhs; return tmp; }
StopWatch operator-(int lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp -= rhs; return tmp; }

StopWatch operator*(StopWatch const& lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp *= rhs; return tmp; }
StopWatch operator*(StopWatch const& lhs, int rhs) { StopWatch tmp{lhs}; tmp *= rhs; return tmp; }
StopWatch operator*(int lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp *= rhs; return tmp; }

std::ostream& operator<<(std::ostream& os, StopWatch const& rhs) {
  int seconds = rhs.Seconds();
  os << std::setfill('0') << std::setw(2) << seconds/3600 << ":";
  seconds %= 3600;
  os << std::setw(2) << seconds/60 << ":";
  seconds %= 60;
  os << std::setw(2) << seconds << "\n";
  return os;
}
```

Possible solution:

```
// declarations of 10 non-member, non-friend operator overloads ...
StopWatch operator+(StopWatch const& lhs, StopWatch const& rhs);
StopWatch operator+(StopWatch const& lhs, int rhs);
StopWatch operator+(int lhs, StopWatch const& rhs);
StopWatch operator-(StopWatch const& lhs, StopWatch const& rhs);
StopWatch operator-(StopWatch const& lhs, int rhs);
StopWatch operator-(int lhs, StopWatch const& rhs);
StopWatch operator*(StopWatch const& lhs, StopWatch const& rhs);
StopWatch operator*(StopWatch const& lhs, int rhs);
StopWatch operator*(int lhs, StopWatch const& rhs);
std::ostream& operator<<(std::ostream& os, StopWatch const& rhs);

// member function definitions ...

// 3 ctors
StopWatch::StopWatch() : seconds{} {}
StopWatch::StopWatch(int secs) : seconds{secs} {}
StopWatch::StopWatch(int hours, int minutes, int secs) : seconds{hours*60*60+minutes*60+secs} {}

// mutator + accessor
void StopWatch::Seconds(int secs) { seconds = secs; }
int StopWatch::Seconds() const { return seconds; }

// 10 member functions of operator overloads
StopWatch& StopWatch::operator++() { ++seconds; return *this; }
StopWatch const StopWatch::operator++(int) { StopWatch old{*this}; ++(*this); return old; }
StopWatch& StopWatch::operator--() { --seconds; return *this; }
StopWatch const StopWatch::operator--(int) { StopWatch old{*this}; --(*this); return old; }

StopWatch& StopWatch::operator+=(StopWatch const& rhs) { seconds += rhs.seconds; return *this; }
StopWatch& StopWatch::operator+=(int rhs) { seconds += rhs; return *this; }
StopWatch& StopWatch::operator-=(StopWatch const& rhs) { seconds -= rhs.seconds; return *this; }
StopWatch& StopWatch::operator-=(int rhs) { seconds -= rhs; return *this; }
StopWatch& StopWatch::operator*=(StopWatch const& rhs) { seconds *= rhs.seconds; return *this; }
StopWatch& StopWatch::operator*=(int rhs) { seconds *= rhs; return *this; }

// 10 non-member, non-friend definitions of operator overloads
StopWatch operator+(StopWatch const& lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp += rhs; return tmp; }
StopWatch operator+(StopWatch const& lhs, int rhs) { StopWatch tmp{lhs}; tmp += rhs; return tmp; }
StopWatch operator+(int lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp += rhs; return tmp; }

StopWatch operator-(StopWatch const& lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp -= rhs; return tmp; }
StopWatch operator-(StopWatch const& lhs, int rhs) { StopWatch tmp{lhs}; tmp -= rhs; return tmp; }
StopWatch operator-(int lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp -= rhs; return tmp; }

StopWatch operator*(StopWatch const& lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp *= rhs; return tmp; }
StopWatch operator*(StopWatch const& lhs, int rhs) { StopWatch tmp{lhs}; tmp *= rhs; return tmp; }
StopWatch operator*(int lhs, StopWatch const& rhs) { StopWatch tmp{lhs}; tmp *= rhs; return tmp; }

std::ostream& operator<<(std::ostream& os, StopWatch const& rhs) {
  int seconds = rhs.Seconds();
  os << std::setfill('0') << std::setw(2) << seconds/3600 << ":";
  seconds %= 3600;
  os << std::setw(2) << seconds/60 << ":";
  seconds %= 60;
  os << std::setw(2) << seconds << "\n";
  return os;
}
```

Question **22**

Correct

Mark 10.00 out of 10.00

Define function **push_back** that inserts an element to the end of a singly-linked list. The function takes two parameters: a pointer to the first element of a singly-linked list and a pointer to the element that is to be inserted into the list. The function returns a pointer to the first element of the list. **Do not include any headers file - none are required!!!**

```
11  // Assume that Node is defined as follow:
12  // struct Node {
13  //   int value;
14  //   struct Node *next;
15  // };
16  // Do not include any standard library headers - none are required!!!
17  // Now, define function push_back ...
18  // in C++ (but not in C), conditional operator evaluates to lvalue
19  Node* push_back(Node *phead, Node *pnew) {
20  Node *ptmp {phead};
21  while (ptmp && ptmp->next) {
22  ptmp = ptmp->next;
23  }
24  // in C++ (but not in C), conditional operator evaluates to lvalue
```

Correct

Evaluation details:

Evaluation:

-Summary of tests

```
>+------------------------------+
>|  1  test run/ 1  test passed |
>+------------------------------+
```

```
Node* push_back(Node *phead, Node *pnew) {
  Node *ptmp {phead};
  while (ptmp && ptmp->next) {
    ptmp = ptmp->next;
  }

  // in C++ (but not in C), conditional operator evaluates to lvalue
  (ptmp ? ptmp->next : phead) = pnew;
  return phead;
}
```

Possible solution:

```
Node* push_back(Node *phead, Node *pnew) {
  Node *ptmp {phead};
  while (ptmp && ptmp->next) {
    ptmp = ptmp->next;
  }

  // in C++ (but not in C), conditional operator evaluates to lvalue
  (ptmp ? ptmp->next : phead) = pnew;
  return phead;
}
```

Question **23**

Correct

Mark 2.00 out of 2.00

Rewrite the indicated statement in the following code fragment so that the overloaded function is called directly. Write your answer without using whitespace and ensure that it is syntactically correct. Otherwise, the grader will mark your answer as incorrect.

```
struct Obj {
  Obj operator+(Obj const&) { std::cout << "binary+"; }
  Obj operator+()           { std::cout << "unary+"; }
};

Obj operator*(Obj const& x, Obj const& y) {
  std::cout << "binary-mul";
}

// in function main ...
Obj o2;
+o2; // rewrite this statement so that overloaded function is called directly
```

Answer: o2.operator+(); ✔

Ordinarily, we call an overloaded operator function indirectly by using the operator on arguments of the appropriate type. However, we can also call an overloaded operator function directly in the same way that we call an ordinary function. We name the function and pass an appropriate number of arguments of the appropriate type:

```
+o2;            // normal expression
o2.operator+(); // equivalent function call
```

The correct answer is: o2.operator+();

Question **24**

Correct

Mark 2.00 out of 2.00

Write the exact values [without extraneous whitespace characters] written to the standard output stream by the following code fragment. Type NC if the code fragment does not compile.

```
void inc(int c, int& b) {
  c++;
  b++;
}

int main() {
  int c{10}, b{20};
  inc(c, b);
  std::cout << c << b;
}
```

Answer: 1021 ✔

The correct answer is: 1021

◄ Assignment 5: Rule of Three

Jump to...

Static Class Members ►