Started on	Thursday, 2 February 2023, 11:38 PM
State	Finished
Completed on	Thursday, 2 February 2023, 11:45 PM
Time taken	7 mins 10 secs
Grade	**13.00** out of 13.00 (**100**%)

Information

Assume all necessary headers are included. Don't provide "doesn't compile" as a valid answer only because certain headers are NOT included in a code fragment.

When it's required to compile a code fragment, add all necessary headers and compile/link using the following compilation options:

```
-std=c++17 -pedantic-errors -Wall -Wextra -Werror
```

Question **1**

Correct

Mark 1.00 out of 1.00

C++ supports dynamic allocation and deallocation of objects using the `new` and `delete` operators. Although `new` and `delete` seem to have the same functionality as C standard library functions `malloc` and `free`, respectively, they differ in various ways:

1. The `new` operator not only allocates an object of user-defined type but also calls the object's constructor to initialize the allocated memory. On the other hand, function `malloc` just allocates the appropriate memory for storing the object. Fundamental data types have no constructors, but nevertheless they can be initialized with the `new` operator:

```
int n = new int{10};
```

2. The `new` operator returns the exact type of the allocated object, while function `malloc` returns `void*`, so typecasting is required. For example if `A` is a user-defined type, we define an object on the heap like this:

```
A *pa {new A};
```

   whereas the pointer returned by function `malloc` must be typecast to type `A`:

```
A *pb = static_cast<A*>(malloc(sizeof(A)));
```

3. Function `malloc` signals failure (for example, when program is out of memory) by returning a null pointer where as `new` operator throws an exception that requires the programmer to use C++ exception handling mechanisms to avoid program termination.
4. While function `free` does nothing more than returning the previously allocated memory back to the heap, the `delete` operator will first call the destructor function of the object's type followed by returning the memory back to the heap.

It should now be clear that C standard library functions `malloc`, `calloc`, `realloc`, and `free` must not be used in C++ programs except in exceptional cases. Such an exceptional scenario arises when a C library calls a C++ function that returns a pointer pointing to dynamically allocated memory that must be returned by the caller to avoid memory leaks.

If the following program doesn't compile, write **NC**. If the program has undefined behavior, write **RTE**. If the program prints unspecified values to standard output, write **UV**. Otherwise, write the values written to standard output.

```
#include <iostream>
#include <stdlib.h>

struct Vector {
  int x, y, z;
  Vector() : x{}, y{}, z{} {} // default constructor
};

int main() {
  Vector *pV {static_cast<Vector*>(malloc(sizeof(Vector)))};
  std::cout << pV->x << pV->y << pV->z;
  free(pV);
}
```

Answer: UV ✔

The correct answer is: UV

Question **2**

Correct

Mark 1.00 out of 1.00

In C++, structures behave differently than in C, for example, allowing not only data but also functions (called member functions) defined inside the structure block. There is an additional minor difference to know about structure types in C++ in comparison to C.

Suppose, we've defined the following structure:

```
struct Vector {
  int x, y, z;
};
```

Recall that in C, the type defined above is `struct Vector` and a variable of such a type would be defined like this:

```
struct Vector vec;
```

In C++, you can define variables of similar type with or without keyword `struct`:

```
struct Vector vec1; // variable of type struct Vector
Vector vec2; // ditto
```

In practice, it is a superficial difference that leads to a little less typing of C++ code. Now, compile the following code using a C [as in HLP1] compiler and then a C++ compiler.

```
#include <stdio.h>

struct Vector {
  int x, y, z;
};

int main(void) {
  Vector v = {10, 20, 30};
  printf("(%d, %d, %d)\n", v.x, v.y, v.z);
  return 0;
}
```

Which of the following best reflects the output of these compilations?

Select one:

- Successfully compiled only by a C compiler
- Successfully compiled only by a C++ compiler ✔
- Not compiled by both a C compiler and a C++ compiler
- Successfully compiled by both a C compiler and a C++ compiler

Well done!

The correct answer is: Successfully compiled only by a C++ compiler

Question **3**

Correct

Mark 1.00 out of 1.00

Given an expression:

```
n = 10
```

that has both rvalues and lvalues, a useful heuristic to determine whether an expression is an *lvalue* is to ask if you can take its address. If you can, it typically is an lvalue expression. If you can't, it's usually an *rvalue* expression. So, in the expression `n=10`, we have two rvalues expressions `10` and `n=10` and one lvalue expression `n`.

Just remember that in C, built-in pre-increment/decrement operators, comma operator, ternary or conditional `?:` operator evaluate to an *rvalue*. In C++ the same operators evaluate to an *lvalue*.

When the expression is a function or overloaded operator call, the answer to the question *what kind of value?* is not so obvious. If the definition or declaration is available, it's possible. For the rest, like the built-in increment operator `++i` you now know that it evaluates to an lvalue expression.

Consider the following C++ code. Write **NC** if the code doesn't compile. Write **RTE** if the code causes undefined behavior at run time. Write **UV** if an unspecified value is written to standard output. Otherwise, write the correct value written to standard output.

```
int main() {
  int i{};
  ++i = 5;
  std::cout << i;
}
```

Answer: 5 ✔

Since `++` operator has higher precedence than `? :` operator, the evaluation of expression `++i` results in an lvalue expression which is assigned rvalue expression `5`. Therefore, the entire expression `++i = 5` results in the assignment of `5` to variable `i` causing value `5` to standard output by the following statement.

The correct answer is: 5

Question **4**

Correct

Mark 1.00 out of 1.00

The following code fragment defines a `namespace` named `Lab` in a C++ source file:

```
namespace Lab {
  int id;
  struct Vector {
    int x,y,z;
  };
} // end namespace Lab
```

The code fragment resembles a C/C++ structure definition with both using curly braces to delimit scope and both declaring members, but they are quite different. Keyword `struct` is used to define a user-defined type that can be instantiated as an object stored in memory. On the other hand, keyword `namespace` is used to organize names (types, functions, variables) into logical groups to prevent name collisions. For example, `namespace Lab` adds scope `Lab` to the names `id` and `Vector` declared in that namespace.

Outside the namespace scope, members can be accessed by specifying a fully qualified name consisting of the namespace's name followed by the scope resolution operator `::` and then followed by the member's name. Therefore, type `Vector` in namespace `Lab` will be accessed outside the namespace like this: `Lab::Vector`. All names at namespace scope are visible to one another without qualification.

What is the output when function `foo` is invoked?

```
namespace A {
  char c = 'a';
}

char c = 'b';

void foo() {
  std::cout << A::c << c;
}
```

Select one:

- `ab` ✔
- `b`
- Code outputs nothing
- Code has a runtime error

Your answer is correct.

The correct answer is: `ab`

Question **5**

Correct

Mark 1.00 out of 1.00

In C++ we read characters from the standard input stream using global variable `std::cin` of C++ standard library type `std::istream` that encapsulates the standard input stream. Here, `std::cin` is the fully qualified name of object `cin` defined in namespace `std`. Class `std::istream` overloads the right-shift operator `>>` to read different types of arithmetic values and text from the keyboard. Format specification, which is the first parameter of function `scanf` is not required. Using overload operator `>>` instead of function `scanf` also allows us to read different values in a single "chained" statement. For example:

```
int x;
double y;
std::cin >> x >> y;
```

Supposing the user types characters `11.7.99` into the standard input stream, write the text written by the following code fragment to the stadard output stream. Ignore the lack of `includes` and functions such as `main` in the code fragment.

```
int x;
double y;
std::cin >> x >> y;
std::cout << x+static_cast<int>(y);
```

Answer: 11 ✔

Operator `>>` is left associative and therefore expression `std::cin >> i` will be evaluated first followed by expression `std::cout >> d`. In all evaluations, `std::cin` will peek at characters in the standard input stream and extract only those characters that can be used to define integral values. So characters `'1'` and `'1'` up until the period character `'.'` will be consumed to build integral value `11` that will then be assigned to variable `i`.

Next, expression `std::cout >> d` will be evaluated. Since the argument to overloaded `operator<<` function is a `double`, `std::cin` will peek at characters in the input stream that can be used to construct a legitimate floating-point value. This means the second period character `'.'` will be left behind in the stream and `std::cin` will construct a floating-point from characters `'.'` and `'7`. Hence, variable `d` of type `double` will be assigned the floating-point value equivalent to `.7`.

The correct answer is: 11

Question **6**

Correct

Mark 1.00 out of 1.00

`nullptr` is a keyword introduced in C++11 and is meant to replace the use of both `0` and the C standard library macro `NULL`. Although the integer literal `0` is a valid value to assign to pointers, it is hard for human readers to detect the use of `0` when used in the context of initialization or assignment to pointers. The use of `NULL` is problematic because the macro typecasts integer literal `0` to either `void*` or `int` depending on the particular compiler.

In contrast to macro `NULL` or integer literal `0`, `nullptr` provides a "type safe" value representing an empty pointer to prevent mixing of data types that are incompatible in your code.

Use the following definition to choose the best possible answer.

```
int *pi {nullptr};
```

Select one:

- `pi` is an object of type `nullptr`
- `pi` is a pointer that does not point to an object ✔
- `pi` is an object of type `int*` that cannot be initialized
- `pi` is a pointer of type `int*` that points to the `nullptr` object

Your answer is correct.

The correct answer is: `pi` is a pointer that does not point to an object

Question **7**

Correct

Mark 1.00 out of 1.00

A structure in C is a user defined data type that can be used to group multiple values under one name. These individual values are called *data members* and they have different names and possibly different types.

A structure in C++ is a more complex data type that provides facilities to easily implement the concepts of *data abstraction* and *encapsulation* [that were explained in class lectures]. In addition to data members, C++ structures can also contain type declarations and functions. *Member functions* of a C++ structure can be used by programmers to initialize, to retrieve, and modify the state of the data members of a variable of that structure type. For example, we can define a type `Student` like this:

```
#include <string>

struct Student {
  std::string name;
  int id;
  double gpa;
  Student(std::string const& a_name, int an_id, double a_gpa) : name{a_name}, id{an_id}, gpa{a_gpa} { }
  void GPA(double new_gpa) { gpa = new_gpa; } // 1st overloaded member function GPA
  double GPA() const { return gpa; }          // 2nd overloaded member function GPA
};
```

A member function that has the same name as the structure type is called a *constructor*. The purpose of a constructor is to initialize the data members of an object.

The first overloaded member function `GPA` is called a *modifier* because it changes the value of data member `gpa`. The second overloaded member function `GPA` is called an *accessor* because it retrieves the current value of data member `gpa`.

What is the output written to standard output by the following code fragment?

```
Student s{"Bart", 98, 2.1};
std::cout << s.name << ' ' << s.id << ' ' << s.GPA();
```

Answer: Bart 98 2.1 ✔

The correct answer is: Bart 98 2.1

Question **8**

Correct

Mark 1.00 out of 1.00

The preprocessor `#include` directive has the form `#include <FILENAME>`, where `FILENAME` is a string for the included header file.

Standard C library headers can be included in C++ source files exactly as in C source files. For example, both C and C++ source files can make references to functions `printf()`, `scanf()`, `fprintf()`<, and others by including header file `<stdio.h>`.

C++ also allows standard C library functions to be referenced using names qualified by namespace `std`. However, in that case, standard C library header filenames must be transformed by the following rule:

1. the header file name no longer maintains the `.h` extension, and
2. the header file name has to be preceded by a `c` character.

For example, `ctype.h` becomes `cctype`. Apply this rule to the following standard C library header filenames to match their corresponding C++ header file [so that functions declared in these standard C library headers can only be referenced in C++ code by qualifying their names with namespace `std`].

`math.h`	cmath ✔
`stdlib.h`	cstdlib ✔
`string.h`	cstring ✔
`stdio.h`	cstdio ✔

Apply the following rule to transform standard C library header filenames:

1. the header file name no longer maintains the `.h` extension, and
2. the header file name has to be preceded by a `c` character.

The correct answer is: `math.h` → cmath, `stdlib.h` → cstdlib, `string.h` → cstring, `stdio.h` → cstdio

Question **9**

Correct

Mark 1.00 out of 1.00

In C, we can have only a single name to reference an object. For example, the definition `int n;` associates name `n` with a block of memory of size 4 bytes. In C++, the concept of references allows us to associate multiple names to an object. For example, the following code fragment

```
int i{10};
int *pi{&i};
int &ri{i};

ri = 21;
std::cout << i << ' ' << ri << ' ' << *pi << '\n';
```

associates name `i` to an `int` object. Such a named object is called a *variable*. The second line in the code framgent defines a pointer variable `pi` and initializes it with the address of the object named `i`. We can *directly* refer to the object using name `i` or *indirectly* refer to the object using expression `*pi`. The third definition introduces name `ri` as a *reference* to variable `i`. C++ uses the term *reference* to mean that the name `ri` is an alias to the name `i`: anywhere you can use the name `i`, you can now use the name `ri`. That is, the object associated with named `i` has a second name `ri` associated with it. Therefore, the code fragment will write `21` thrice to standard output.

Although it seems that references are similar to pointers, they differ in several essential ways:

- Pointers can be uninitialized, while references must be initialized [because they must be an alias of something].
- Once a reference is initilaized to alias an object, it cannot be made to alias a different object. On the other hand, pointers can be reassigned to point to another object.
- There is no such thing as a null reference while we can have a null pointer [by initializing or assigning the `nullptr` value to a pointer variable].

Although every reference must be initialized to refer to something, it is still possible to have a *dangling reference*. Dangling references arise when a reference is initialized to alias a dynamically allocated object which is then subseqently deleted. Consider the following code fragment:

```
int *pi{new int{10};
int &ri {*pi}; // ri is now an alias to unnamed object pointed to by pi
delete pi;     // the object referenced by pi is no longer in existence!!!
               // note that physical memory still exists but not the dynamic object that
               // was earlier given storage there
               // therefore, ri is now a dangling reference
```

Which of the following fragments of code do NOT compile?

Select one or more:

-
```
int c = 10;
int &c = c;
```
✔

-
```
int c;
int &b = c;
```

-
```
int c = 10;
int &b;
b = c;
```
✔

-
```
int c = 10;
char &b = c;
```
✔

Your answer is correct.

The correct answers are:

```
int c = 10;
char &b = c;
```

,

```
int c = 10;
int &b;
b = c;
```

```
int c = 10;
int &c = c;
```

Question **10**

Correct

Mark 1.00 out of 1.00

In C++ we write characters to the standard output stream using global variable `std::cout` of a C++ standard library type `std::ostream` that encapsulates the standard output stream. Here, `std::cout` is the fully qualified name of object `cout` defined in `std` namespace. Class `std::ostream` overloads the left-shift operator `<<` to write the characters representing different type of values and text to the output stream. Format specification, which is the first parameter of function `printf` is not required. Using overloaded operator `<<` instead of function `printf` also allows us to write different values in a single "chained" statement. For example:

```
std::cout << "character: " << ch << std::endl;
```

`std::endl` represents a *stream manipulator* that inserts the end of line character followed by flushing of the standard output stream. You can keep using `'\n'`, but `endl` makes your code more portable [because we don't have to remember the newline character when a non-ASCII character set is used].

What is the result of compilation and execution of the following code? Type the *exact* output or type **NC** if the code doesn't compile.

```
#include <iostream>

int main() {
  cout << "Hello World!";
  return 0;
}
```

Answer: NC ✔

The correct answer is: NC

Question **11**

Correct

Mark 1.00 out of 1.00

Recall from HLP1 that *external variables* are variables that are defined outside of any function and by default such variables have *external linkage*. *External linkage* means that external variables will have their names exported by the compiler to the linker. This makes external variables defined in one source file accessible to functions defined in other source files. In contrast, *internal linkage* means names of external variables defined in a source file are not exported by the compiler to the linker and therefore such names will only be visible to functions in only that source file. Programmers can force external variables to have internal linkage by adding storage specifier keyword `static` to the definition of the external variable, as in:

```
static int global_variable;
```

In C11, read-only `const` external variables have external linkage by default. Determine whether such read-only external variables have internal linkage or external linkage in C++ by examining the following C++ code in a source file:

```
const int foo = 1;

int main() {
  return 0;
}
```

and the following code in a second source file:

```
const int foo = 4;
```

Which of the following best reflects the output of compilation of these two source files and the linkage of the corresponding two object files?

Select one:

- Compile error
- Compile warning
- Linker error
- Linker warning
- No error. Executable created. ✔

The correct answer is: No error. Executable created.

Question **12**

Correct

Mark 1.00 out of 1.00

Just as in C, type qualifier `const` is used in C++ to declare a variable that is designated as *read-only* so that the variable's value *cannot* be changed during program execution. Think about the output of the following code fragment.

```
#include <iostream>

int main() {
  int const value{10};
  std::cout << "value: " << value << "\n";
  value = 20;
  std::cout << "value: " << value << "\n";
}
```

In both languages, the rule is that a `const` variable must be initialized during its definition. What is your opinion of the following code?

```
#include <iostream>

int main() {
  int const value;
  value = 20;
  std::cout << "value: " << value << "\n";
}
```

A ***constant expression*** is an expression whose value cannot change and that can be evaluated at compile time. A literal is a constant expression. Additionally in C++, a `const` variable initialized from a constant expression is also a constant expression. For example, the following C++ code fragment

```
int const degree{4};
int poly[degree] {11,12,13,14};
```

establishes two levels of constancy for variable `degree`: variable `degree` cannot be changed during program execution [just as in C]; and variable `degree` is a constant expression and can therefore be used to specify the size of a static array [unlike in C]. Compare the following C program's behavior:

```
#include <stdio.h>

int main(void) {
  int const value = 4;
  int array[value] = {1, 2, 3, 4};
  printf("%d\n", array[1]);
  return 0;
}
```

with the behavior of this C++ program

```
#include <iostream>

int main() {
  int const value {4};
  int array[value] {1, 2, 3, 4};
  std::cout << array[1] << "\n";
}
```

to deduce that a `const` variable initialized from a constant expression [such as `value`] is also a constant expression only in C++ code but not in C code.

Clearly, a constant expression [that can be evaluated at compile time] has several advantages:

1. A constant expression can be evaluated at compile time and therefore saves CPU cycles from evaluating the expression at run time.
2. A constant expression provides the compiler deep insight into the code so that the compiler can potentially determine additional optimizations.
3. A constant expression makes the code implicitly thread safe which is difficult in multi-threaded expressions.

Whether a given variable or expression is a constant expression depends on the types and initializers:

```
int square(int n) {
  return n*n;
}

int main() {
  int const max_files{20};       // max_files is a constant expression
  int const limit {max_files};   // limit is a constant expression
  int size{27};                  // size is not a constant expression
  int const sq_size{square(2)};  // sq_size is not a constant expression
                                 // because the value of the initializer is not known until run time
  int array[square(2)]{1,2,3,4}; // illegal because square(2) is not a constant expression
  std::cout << array[2] << "\n";
}
```

Since C+11, the standard has introduced a new specifier that allows programmers to specify that the value of a variable [such as `size`] or an expression [such `square(2)`] must be evaluated at compile time. This keyword when used as a specifier for function `square` will enable `const` variable `sq_size` to be a constant expression and thus allow the compiler to define the static array `array` with size `4`.

Read Section 2 of the textbook to learn what this C++ keyword is and then write the exact keyword as your answer.

Answer: constexpr ✔

`constexpr`

The correct answer is: constexpr

Question **13**

Correct

Mark 1.00 out of 1.00

In C++, an array is dynamically allocated of an array using the `new[]` operator:

```
int *pi {new int[10]};
```

When dynamically allocating memory for a two-dimensional array, early C++ programmers make the common mistake of writing the following incorrect statement to allocate a matrix of 5 rows and 6 columns:

```
int *p2i {new int[5][6]}; // error
```

The simplest and most efficient solution is to rely on the fact that both C and C++ use a row-major order to conceptualize a two-dimensional array in the one-dimensional sequence of bytes that comprise physical memory. For example, a two-dimensional array with `5` rows and `6` columns is conceptualized as a one-dimensional array of 30 elements:

```
int *p2i {new int[5 * 6]};
```

An element located at the intersection of row `y` and column `x` in this conceptual two-dimensional array is accessed by mapping the offset of this two-dimensional element within the one-dimensional array pointed to by `p2i` [defined above]:

```
// map two-dimensional element at intersection of row y and column x
int offset = y*6 + x;
```

Now, read the following code fragment and determine the value written to standard output.

```
int const rows{3}, columns{4};
int *p2i { new int[rows*columns]{7,10,17,12,3,14,-5,16,18,19,11,8} };
int r{1}, c{3};
std::cout << *(p2i+r*columns+c);
```

Answer: 16 ✔

The correct answer is: 16

◄ Assignment 3: Problem Solving with C++ std::vector and std::string

Jump to...

Quiz 2: Review of C++ Functions [Part 1] ►