karlina_object_winter_2023_abridged_print_pages_0_to_17

[Note that this Portable Format Document (to print out onto pieces of white paper which are each 8.5 inches wide and 11 inches tall using black ink, sans-serif font, and 11 point font size) contains plain-text content only and that not all the content which is featured on the website named Karlina Object dot WordPress dot Com is featured also in this document].

The final draft version of this document was published on 21_OCTOBER_2023.

KARLINA_OBJECT

Creativity And The Cosmos

KARLINA OBJECT dot WordPress dot Com

---

image_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/romanesco_broccoli_19_march_2021.jpeg

---

INTRODUCTION

This website is an expression of a person’s interest and expertise in topics such as metaphysics, physics, computer science, mathematics, logic, and digital media (and the name of that person is karbytes).

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/karlina_object_wordpress_logo_orange_box_drawing_2020.png

---

WEBSITE_DIRECTORY

PAGE_28: KNOWLEDGE

PAGE_27: TRIANGLE_GRAPHING

PAGE_26: WORD_COUNTER

PAGE_25: PROBABILITY

PAGE_24: SOUND_TRACK_LOOP_COUNTER

PAGE_23: AGENCY

PAGE_22: BASE_CONVERTER

PAGE_21: PI_APPROXIMATION

PAGE_20: COUNT_DOWN_TIMER

PAGE_19: BITS_AND_BYTES

PAGE_18: HEXIDECIMAL_COLOR_CODES

PAGE_17: MULTIVERSE

PAGE_16: HASH_TABLE

PAGE_15: LINKED_LIST

PAGE_14: POLYGON

PAGE_13: TRIANGLE

PAGE_12: POINT

PAGE_11: POINTERS_AND_ARRAYS

PAGE_10: EXPONENTIATION

PAGE_9: NATURE

PAGE_8: EULERS_NUMBER_APPROXIMATION

PAGE_7: CUBE_ROOT_APPROXIMATION

---

image_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/dimensions_02_october_2019.jpg

---

AUTHOR_DETAILS

Name: karbytes (self-assigned identifier), Karlina Ray Beringer (legal identifier)

Species: HUMAN (originally), CYBORG (currently)

Birth_Date: 13_JANUARY_1990

Birth_Location: California, United States of America, Planet Earth

Profession: Software Developer, Consciousness Researcher

AUTHOR_NETWORKING

Email_0: karlina.ray.beringer@protonmail.com

Email_1: stardustsstructures@protonmail.com

Email_2: karlinaberinger99@gmail.com

Email_3: karbytesforlife@protonmail.com

KARBYTES: https://karbytesforlifeblog.wordpress.com/

GitHub: https://github.com/karlinarayberinger/

LinkedIn: https://www.linkedin.com/in/kar-beringer-0a6684187/

Patreon: https://www.patreon.com/karbytes

Instagram: https://www.instagram.com/karbytes_anew/

Twitter: https://twitter.com/karbytes

Minds: https://www.minds.com/karbytes/

DONATIONS_PORTAL

Visitors to this website are invited to contribute a variable amount of money to the author of this website using the following PayPal donation portal link:

https://www.paypal.com/donate/?hosted_button_id=6CZQQJLS74TN4

(The suggested amount of money to donate is $5).

WEBSITE_CHRONOLOGY

Backed up versions of each one of the web pages in this website are available to view at Archive dot Org (and were saved to Archive dot Org using the WayBack Machine).

This web page was last updated on 21_SEPTEMBER_2023. The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

This website was established on 03_APRIL_2020.

[End of abridged plain-text content from START_PAGE]

PUBLIC_DOMAIN

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/nested_yellow_boxes_03_july_2022.jpg

---

The following terms, the respective definition of each of those terms, and the specifications for coloring hyperlinks describe the intellectual property which comprises this website (i.e. the website named Karlina Object dot WordPress dot Com). Note that the following definitions might not pertain to their respective terms used outside the scope of this website.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

FILE: a named sequence of some natural number of binary digits which represents a verbatim transmissible piece of information.

Each web page which comprises this website is a Hyper-Text-Markup-Language (HTML) file.

The image which is displayed on this web page (which depicts nested yellow boxes whose back interior surfaces are colored black) is a Joint-Photographic-Experts-Group (JPEG) file.

INTELLECTUAL_PROPERTY: a collection of some natural number of files.

OPEN_SOURCE: a classification of intellectual property which specifies that the respective intellectual property is licensed by its original author as being open source and is, hence, (a) legal for any person to make indefinitely many verbatim copies of, (b) legal for any person to freely distribute or else sell verbatim copies of, (c) legal for any person to modify verbatim copies of, (d) legal for any person to freely distribute or else sell modified copies of, and (e) legal for any person to claim any of those verbatim or modified copy files as that person’s own intellectual property.

(Note that, within the context of this web page, the term author is interchangeable with the term creator).

OPEN_SOURCE: a classification of intellectual property which specifies that the respective intellectual property is licensed by its original author as being open source and is, hence, (a) legal for any person to make indefinitely many verbatim copies of, (b) legal for any person to freely distribute or else sell verbatim copies of, (c) legal for any person to modify verbatim copies of, (d) legal for any person to freely distribute or else sell modified copies of, and (e) legal for any person to claim any of those verbatim or modified copy files as that person's own intellectual property.

(Note that, within the context of this web page, the term author is interchangeable with the term creator).

PUBLIC_DOMAIN: open source intellectual property which is legally determined to be ownerless after a specific date and, forever after that date, part of the public World Wide Web.

The author of this website has saved the source code of each web page of this website and each source code file or media file which that web page displays (and references via hyperlink) in a public GitHub repository which the author of this website created.

The author of this website has attempted to save each web page of this website multiple times to the WayBack Machine such that each of those WayBack Machine saves of Karlina Object dot WordPress dot Com web pages can be retrieved indefinitely many times from any terminal of the World Wide Web (without having to enter login credentials nor pay money) by using the Uniform Resource Locator (URL) of the respective Karlina Object dot WordPress dot Com web page as a search term to look up its corresponding saves in the WayBack Machine database.

For each unique URL which is successfully saved to the WayBack Machine, there exists a collection of some natural number of corresponding WayBack Machine saves. Each of those saves depicts its respective web page at the time that web page was captured by the WayBack Machine. For each unique URL which was saved to the WayBack Machine, the WayBack Machine has a corresponding database featuring a calendar interface with hyperlinked days in which saves of the respective web page was made. Users can click on those hyperlinked days of the calendar to view the saves of the respective web page which were made on the respective day.

Each of the web pages which are part of this website and each of the code files and each of the media files which are embedded in the web pages of this website is hereby designated by the author of this website to forever be public domain intellectual property. The original author of each of those aforementioned files is exactly one person named karbytes.

HYPERLINK_COLORING_CONVENTIONS

The following specifications pertain to each hyperlink which is displayed on any web page of this website:

Each of the hyperlinks which is displayed on this website has a background color or a text color which is either black (i.e. #000000), green (i.e. #00ff00), or orange (i.e. #ff9000).

A hyperlink whose background color is orange and whose text color is black refers to a file which belongs to a website which karbytes did not create.

A hyperlink whose background color is green and whose text color is black refers to a GitHub repository which karbytes created or to a GitHub account which karbytes created.

(Note that karbytes did not create the GitHub website. Instead, karbytes is a user of GitHub (and karbytes is a GitHub user who has created private and public GitHub repositories using the GitHub website to host those repositories (and each of those repositories contains zero or more files which karbytes uploaded and which karbytes is the creator of))).

A hyperlink whose background color is black refers to a GitHub web page displaying exactly one file which karbytes is the original author of (and that file is either a source code file or else a media file).

An example of a source code file which karbytes created and uploaded to GitHub is the file named fibonacci.cpp.

An example of a media file which karbytes created and uploaded to GitHub is the file named cube_comprised_of_eight_equally_sized_cubes_03_may_2023.jpeg.

A hyperlink whose background color is black and whose text color is green refers to (a) a source code file which karbytes is the creator of or else (b) to a web page of a WordPress website which karbytes is the creator of and such that the referenced web page is considered by karbytes to be more code-dominant than prose-dominant.

A hyperlink whose background color is black and whose text color is orange refers to (a) a media file which karbytes is the creator of or else (b) to a web page of a WordPress website which karbytes is the creator of and such that the referenced web page is considered by karbytes to be more prose-dominant than code-dominant.

ASSUMPTIONS

The entity (or entities) which create(s) a particular piece of intellectual property is (or are) implicitly, automatically, and immediately assigned sole ownership of that piece of intellectual property as soon as that piece of intellectual property is created.

After a piece of intellectual property is created, the creator (and, hence, the original owner) of that piece of intellectual property can assign a particular license to that piece of intellectual property which specifies how the creator of that intellectual property wants other people to use that intellectual property. For instance, the original owner of a piece of intellectual property might license that piece of intellectual property as being illegal to access without first purchasing an access code through the vendor website to download a copy of the files which constitute that piece of intellectual property.

(In the aforementioned example, it is presumably illegal for a person who legally purchases the access code to download proprietary intellectual property to distribute copies of that downloaded content to other people because doing so is, according to the license assigned to that downloaded content, stealing money from the original proprietor of that (non open source and non public domain) intellectual property).

In addition to assuming that the original creators of a piece of intellectual property are initially the sole proprietors of that piece of intellectual property, that piece of intellectual property is also assumed to initially be private instead of public. Intellectual property which is private instead of part of the public domain is illegal to share without that intellectual property owner's explicit permission for each instance of enabling some entity other than the owner of that intellectual property access to that intellectual property. By contrast, intellectual property which is part of the public domain is legal for any entity to access and to claim ownership of (but claiming ownership of a piece of intellectual property which is copied wholly or partially from some piece of intellectual property which was licensed as public domain or as open source does not automatically make that copy also public domain (but the copy may be assigned more exclusive access conditions which may or may not be physically enforceable)).

// The following pseudocode elaborates on what was discussed in the previous four paragraphs.

Let time_0 be a point in time which occurs before the point in time named time_1.

Let time_1 be a point in time which occurs before the point in time named time_2.

Let person_A and person_B each be unique end users of the same digital content distribution network.

Let ip_X and ip_Y each be unique instances of intellectual property which can be verbatim encoded as a particular finite sequence of binary digits within the digital computers which comprise that digital content distribution network which person_A and person_B use.

Suppose person_A creates a piece of intellectual property named ip_X at time_0.

Then, at time_0, ip_X is privately owned exclusively by person_A.

At time_0 and before time_1, it is illegal for person_B to own a copy of ip_X.

Then, at time_1, person_A assigns the license of public_domain to ip_X.

At time_1 and after time_1, it is legal for person_B to own a copy of ip_X.

Then, at time_2, person_B makes a copy of ip_X and names that copy ip_Y.

At time_2 and after time_2, it is illegal for person_A to own a copy of ip_Y but it is not illegal for person_A to own a copy of ip_X.

Finally, a piece of intellectual property which is (a) distributed sufficiently many (and arbitrarily many) times and is (b) substantiated as digital files in sufficiently many (and arbitrarily many) digital storage mediums (especially web content hosting servers for hosting content which is accessible to the general public) simultaneously for a (c) sufficiently long (and arbitrarily long) period of time after the most recent sole proprietor of that intellectual property dies (or if the most recent sole proprietor is an artificial intelligence without legally recognized personhood) is released to the public domain or else that piece of intellectual property is assumed to be entirely noumenal and to be treated as legally having never phenomenally instantiated at any prior point in time (due to the fact that presumably zero physical records currently exist which encode that particular piece of intellectual property).

It is implied by the previous paragraph that the same piece of intellectual property could potentially be created multiple times within the same universe (provided that those instantiation periods are non-overlapping) and each instantiation would legally be treated as the first time the associated piece of intellectual property ever emerged.

[End of abridged plain-text content from PUBLIC_DOMAIN]

FACTORIAL

The C++ program featured in this tutorial web page computes N factorial (N!) using recursion and using iteration. If N is a natural number, then N! is the product of exactly one instance of each unique natural number which is less than or equal to N. If N is zero, then N! is one.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

0! := 1. // base case: when N is zero
N! := N * (N - 1)! // recursive case: when N is any natural number

---

SOFTWARE_APPLICATION_COMPONENTS

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/factorial.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/factorial_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ source code into a new text editor document and save that document as the following file name:

factorial.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ factorial.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: After running the g++ command, run the executable file using the following command:

./app

STEP_5: Once the application is running, the following prompt will appear:

Enter a nonnegative integer which is no larger than 12:

STEP_6: Enter a value for N using the keyboard.

STEP_7: Observe program results on the command line terminal and in the output file.

PROGRAM_SOURCE_CODE

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/factorial.cpp

---

```
/**
 * file: factorial.cpp
 * type: C++ (source file)
 * date: 14_JUNE_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#include < iostream > // library for defining objects which handle command line input and
command line output
#include < fstream > // library for defining objects which handle file input and file output
```

```
#define MAXIMUM_N 12 // constant which represents maximum N value

/* function prototypes */
int compute_factorial_of_N_using_recursion(int N, std::ostream & output);
int compute_factorial_of_N_using_iteration(int N, std::ostream & output);

/**
 * Compute N factorial (N!) using a recursive algorithm.
 *
 * Assume that N is an integer value and that output is an output stream object.
 *
 * For each compute_factorial_of_N_using_recursion function call,
 * print an algebraic expression which represents N factorial.
 *
 * 0! := 1. // base case: when N is smaller than 1 or when N is larger than MAXIMUM_N.
 * N! := N * (N - 1)! // recursive case: when N is larger than or equal to 1 and when N is smaller
than or equal to MAXIMUM_N.
 */
int compute_factorial_of_N_using_recursion(int N, std::ostream & output)
{
	// base case: if N is smaller than 1 or if N is larger than MAXIMUM_N, return 1.
	if ((N < 1) || (N > MAXIMUM_N))
	{
	output << "\n\nfactorial(" << N << ") = 1. // base case";
	return 1;
	}
	// recursive case: if N is larger than or equal to 1 and if N is smaller than or equal to
MAXIMUM_N, return N multiplied by (N - 1) factorial.
	else
	{
	output << "\n\nfactorial(" << N << ") = " << N << " * factorial(" << N - 1 << "). // recursive
case" ;
	return N * compute_factorial_of_N_using_recursion(N - 1, output);
	}
}

/**
 * Compute N factorial using an iterative algorithm.
 *
 * Assume that N is an integer value and that output is an output stream object.
 *
 * For each while loop iteration, i,  print the ith multiplicative term of N factorial.
 *
```

```
 *
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
 * If N is a larger than or equal to 1 and if N is smaller than or equal to MAXIMUM_N,
 * N! is the product of exactly one instance of each unique natural number which is smaller than
or equal to N.
 *
 * N! := N * (N - 1) * (N - 2) * (N - 3) * ... * 3 * 2 * 1. // if N is an arbitrarily large natural number
(which, in this line, is equal to or larger than 7)
 *
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
 *
 * If N is zero, then N! is one.
 *
 * 0! := 1.
 *
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------
 */
int compute_factorial_of_N_using_iteration(int N, std::ostream & output)
{
	int i = 0, F = 0;
	i = ((N > 0) && (N <= MAXIMUM_N)) ? N : 0;
	F = (N > 0) ? N : 1;
	output << "\n\nfactorial(" << i << ") = ";
	while (i > 0) // Execute the code block encapsulated by the while loop while the condition
"i > 0" is true.
	{
	output << i << " * "; // Print the value of i followed by " * " to the output stream.
	if (i > 1) F *= i - 1; // If i is larger than 1, multiply F by (i - 1).
	i -= 1; // Decrement i by 1.
	}
	output << "1.";
	return F;
}

/* program entry point */
int main()
{
	// Declare three int type variables and set each of their initial values to 0.
	int N = 0, A = 0, B = 0;

	// Declare a file output stream object.
```

```
std::ofstream file;

/**
* If factorial_output.txt does not already exist in the same directory as factorial.cpp,
* create a new file named factorial_output.txt.
*
* Open the plain-text file named factorial_output.txt
* and set that file to be overwritten with program data.
*/
file.open("factorial_output.txt");

// Print an opening message to the command line terminal.
std::cout << "\n\n--------------------------------";
std::cout << "\nStart Of Program";
std::cout << "\n--------------------------------";

// Print an opening message to the file output stream.
file << "--------------------------------";
file << "\nStart Of Program";
file << "\n--------------------------------";

// Print "Enter a nonnegative integer which is no larger than {MAXIMUM_N}: " to the command line terminal.
std::cout << "\n\nEnter a nonnegative integer which is no larger than " << MAXIMUM_N << ": ";

// Scan the command line terminal for the most recent keyboard input value.
std::cin >> N;

// Print "The value which was entered for N is {N}." to the command line terminal.
std::cout << "\nThe value which was entered for N is " << N << ".";

// Print "The value which was entered for N is {N}." to the file output stream.
file << "\n\nThe value which was entered for N is " << N << ".";

// If N is smaller than 0 or if N is larger than MAXIMUM_N, set N to 0.
N = ((N < 0) || (N > MAXIMUM_N)) ? 0 : N; // A tertiary operation (using the tertiary operator (?)) is an alternative to using if-else statements.

// Print "N := {N}." to the command line terminal.
std::cout << "\n\nN := " << N << ".";

// Print "N := {N}." to the file output stream.
file << "\n\nN := " << N << ".";
```

```
	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print "Computing factorial N using recursion:" to the command line terminal.
	std::cout << "\n\nComputing factorial N using recursion:";

	// Print "Computing factorial N using recursion:" to the file output stream.
	file << "\n\nComputing factorial N using recursion:";

	// Compute N factorial using recursion, store the result in A, and print each function call in the recursive function call chain to the command line terminal.
	A = compute_factorial_of_N_using_recursion(N, std::cout);

	// Compute N factorial using recursion and print each function call in the recursive function call chain to the file output stream.
	compute_factorial_of_N_using_recursion(N, file);

	// Print the value of A to the command line terminal.
	std::cout << "\n\nA = factorial(" << N << ") = " << A << ". // " << N << "! = " << A << ".";

	// Print the value of A to the file output stream.
	file << "\n\nA = factorial(" << N << ") = " << A << ". // " << N << "! = " << A << ".";

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print "Computing factorial N using iteration:" to the command line terminal.
	std::cout << "\n\nComputing factorial N using iteration:";

	// Print "Computing factorial N using iteration:" to the file output stream.
	file << "\n\nComputing factorial N using iteration:";

	// Compute N factorial using iteration and print each multiplicative term of N! to the command line terminal.
	B = compute_factorial_of_N_using_iteration(N, std::cout);
```

```
	// Compute N factorial using iteration and print each multiplicative term of N! to the file output stream.
	compute_factorial_of_N_using_iteration(N, file);

	// Print the value of B to the command line terminal.
	std::cout << "\n\nB = factorial(" << N << ") = " << B << ". // " << N << "! = " << B << ".";

	// Print the value of B to the file output stream.
	file << "\n\nB = factorial(" << N << ") = " << B << ". // " << N << "! = " << B << ".";

	// Print a closing message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nEnd Of Program";
	std::cout << "\n--------------------------------\n\n";

	// Print a closing message to the file output stream.
	file << "\n\n--------------------------------";
	file << "\nEnd Of Program";
	file << "\n--------------------------------";

	// Close the file output stream.
	file.close();

	// Exit the program.
	return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/factorial_output.txt

---

```
--------------------------------
Start Of Program
--------------------------------
```

The value which was entered for N is 12.

N := 12.

Computing factorial N using recursion:

factorial(12) = 12 * factorial(11). // recursive case

factorial(11) = 11 * factorial(10). // recursive case

factorial(10) = 10 * factorial(9). // recursive case

factorial(9) = 9 * factorial(8). // recursive case

factorial(8) = 8 * factorial(7). // recursive case

factorial(7) = 7 * factorial(6). // recursive case

factorial(6) = 6 * factorial(5). // recursive case

factorial(5) = 5 * factorial(4). // recursive case

factorial(4) = 4 * factorial(3). // recursive case

factorial(3) = 3 * factorial(2). // recursive case

factorial(2) = 2 * factorial(1). // recursive case

factorial(1) = 1 * factorial(0). // recursive case

factorial(0) = 1. // base case

A = factorial(12) = 479001600. // 12! = 479001600.

Computing factorial N using iteration:

factorial(12) = 12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 * 1.

B = factorial(12) = 479001600. // 12! = 479001600.

End Of Program

This web page was last updated on 17_OCTOBER_2023. The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

[End of abridged plain-text content from FACTORIAL]

FIBONACCI_NUMBERS

The C++ program featured in this tutorial web page computes the Nth term of the Fibonacci Sequence using recursion and using iteration. If N is a natural number which is larger than or equal to two, then fibonacci(N) is the sum of fibonacci(N – 2) and fibonacci(N – 1). If N is either zero or else one, then fibonacci(N) is one.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

```
fibonacci(0) := 1. // The first term of the Fibonacci Sequence is 1.
fibonacci(1) := 1. // The second term of the Fibonacci Sequence is 1.
fibonacci(i) := fibonacci(i - 2) + fibonacci(i - 1). // i is a natural number which is larger than or equal to 2.
```

SOFTWARE_APPLICATION_COMPONENTS

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/fibonacci_numbers.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/fibonacci_numbers_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ source code into a new text editor document and save that document as the following file name:

fibonacci_numbers.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ fibonacci_numbers.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: After running the g++ command, run the executable file using the following command:

./app

STEP_5: Once the application is running, the following prompt will appear:

Enter a nonnegative integer which is no larger than 45:

STEP_6: Enter a value for N using the keyboard.

STEP_7: Observe program results on the command line terminal and in the output file.

PROGRAM_SOURCE_CODE

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/fibonacci_numbers.cpp

---

```
/**
 * file: fibonacci_numbers.cpp
 * type: C++ (source file)
 * date: 14_JUNE_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#include < iostream > // standard input (std::cin), standard output (std::cout)
#include < fstream > // file input, file output
#define MAXIMUM_N 45 // constant which represents maximum N value

/* function prototypes */
int compute_Nth_fibonacci_sequence_term_using_recursion(int N, std::ostream & output, int &
C);
int compute_Nth_fibonacci_sequence_term_using_iteration(int N, std::ostream & output);

/**
 * Compute the Nth term of the Fibonacci Sequence using a recursive algorithm.
 *
 * Assume that N is an integer value and that output is an output stream object.
 *
 * Assume that C is a reference to an int type variable whose initial value is zero.
 *
 * C is assumed to represent the total number of times this function is called during
 * a particular function call chain which is initiated when this function is called
 * inside the scope in which C is declared.
 *
 * If this function is going to be called more than one time from inside of the same
 * scope in which C is declared, C will need to be reset to 0 before each of those
 * function calls is implemented to ensure that C stores the correct number of time this
 * function calls itself during a particular
compute_Nth_fibonacci_sequence_term_using_recursion
 * function call from within C's program scope.
 *
 * For each compute_Nth_fibonacci_sequence_term_using_recursion function call,
 * print an algebraic expression which represents the Nth term of the Fibonacci Sequence.
```

```
 *
 * --------------------------------------------------------------------------------------------------
 *
 * The first term of the Fibonacci Sequence is one.
 *
 * fibonacci(0) := 1.
 *
 * --------------------------------------------------------------------------------------------------
 *
 * The second term of the Fibonacci Sequence is one.
 *
 * fibonacci(1) := 1.
 *
 * --------------------------------------------------------------------------------------------------
 *
 * If N is a natural number larger than or equal to two,
 * the Nth term of the Fibonacci Sequence is the sum
 * of the previous two terms of the Fibonacci Sequence.
 *
 * fibonacci(N) := fibonacci(N - 2) + fibonacci(N - 1).
 *
 * --------------------------------------------------------------------------------------------------
 */
int compute_Nth_fibonacci_sequence_term_using_recursion(int N, std::ostream & output, int &
C)
{
	// base case: if N is smaller than 2 or if N is larger than MAXIMUM_N, return 1.
	if ((N < 2) || (N > MAXIMUM_N))
	{
	C += 1;
	output << "\n\nfibonacci(" << N << ") = 1. // base case (C = " << C << ")";
	return 1;
	}
	// recursive case: if N is larger than 2 and if N is smaller than or equal to MAXIMUM_N,
	// return the sum of the (N - 2)th term of the Fibonacci Sequence
	// and the (N - 1)nth term of the Fibonacci Sequence.
	else
	{
	C += 1;
	output << "\n\nfibonacci(" << N << ") = fibonacci(" << N - 2 << ") + fibonacci(" << N - 1 <<
"). // recursive case (C = " << C << ")" ;
	return compute_Nth_fibonacci_sequence_term_using_recursion(N - 2, output, C) +
compute_Nth_fibonacci_sequence_term_using_recursion(N - 1, output, C);
	}
```

```
}

/**
 * Compute the Nth term of the Fibonacci Sequence using an iterative algorithm.
 *
 * Assume that N is an integer value and that output is an output stream object.
 *
 * For each while loop iteration, i,
 * print an algebraic expression which represents the ith term of the Fibonacci Sequence.
 *
 * fibonacci(0) := 1. // The first term of the Fibonacci Sequence is 1.
 * fibonacci(1) := 1. // The second term of the Fibonacci Sequence is 1.
 * fibonacci(i) := fibonacci(i - 2) + fibonacci(i - 1). // if i is a natural number larger than 1
 */
int compute_Nth_fibonacci_sequence_term_using_iteration(int N, std::ostream & output)
{
	// Define four int type variables for storing whole number values which increment zero or more times during any compute_Nth_fibonacci_sequence_term_using_iteration function call.
	int i = 0, A = 1, B = 1, C = 0;

	// Print the value of the first term of the Fibonacci Sequence (i.e. fibonacci(0)) to the output stream.
	output << "\n\nfibonacci(" << i << ") = 1. // i = " << i; // i = 0

	// If N is smaller than 1 or if N is larger than MAXIMUM_N, return 1.
	if ((N < 1) || (N > MAXIMUM_N)) return 1;

	// Increment the value of i by one.
	i += 1;

	// Print the value of the second term of the Fibonacci Sequence (i.e. fibonacci(1)).
	output << "\n\nfibonacci(" << i << ") = 1. // i = " << i; // i = 1

	// If N is equal to 1, return 1.
	if (N == 1) return 1;

	// If N is larger than 2, return the sum of the (N - 2)th term of the Fibonacci Sequence and the (N - 1)nth term of the Fibonacci Sequence.
	while (i < N)
	{
	i += 1;
	C = A;
	A = B;
	B += C;
```

```
        output << "\n\nfibonacci(" << i << ") = ";
        output << B << " = fibonacci(" << i - 2 << ") + fibonacci(" << i - 1 << ") = ";
        output << C << " + " << A;
        output << ". // i = " << i;
        }

        // Return the value of fibonacci(N).
        return B;
}

/* program entry point */
int main()
{
        // Declare four int type variables and set each of their initial values to 0.
        int N = 0, A = 0, B = 0, C = 0;

        // Declare a file output stream object.
        std::ofstream file;

        /**
        * If fibonacci_numbers_output.txt does not already exist in the same directory as
fibonacci_numbers.cpp,
        * create a new file named fibonacci_numbers_output.txt.
        *
        * Open the plain-text file named fibonacci_numbers_output.txt
        * and set that file to be overwritten with program data.
        */
        file.open("fibonacci_numbers_output.txt");

        // Print an opening message to the command line terminal.
        std::cout << "\n\n--------------------------------";
        std::cout << "\nStart Of Program";
        std::cout << "\n--------------------------------";

        // Print an opening message to the file output stream.
        file << "--------------------------------";
        file << "\nStart Of Program";
        file << "\n--------------------------------";

        // Print a warning message to the command line terminal.
        std::cout << "\n\nWARNING: the recursive function execution time increases
exponentially as the value of N increases.";

        // Print a warning message to the file output stream.
```

```
	file << "\n\nWARNING: the recursive function execution time increases exponentially as the value of N increases.";

	// Print "Enter a nonnegative integer which is no larger than {MAXIMUM_N}: " to the command line terminal.
	std::cout << "\n\nEnter a nonnegative integer which is no larger than " << MAXIMUM_N << ": ";

	// Scan the command line terminal for the most recent keyboard input value.
	std::cin >> N;

	// Print "The value which was entered for N is {N}." to the command line terminal.
	std::cout << "\nThe value which was entered for N is " << N << ".";

	// Print "The value which was entered for N is {N}." to the file output stream.
	file << "\n\nThe value which was entered for N is " << N << ".";

	// If N is smaller than 0 or if N is larger than MAXIMUM_N, set N to 0.
	N = ((N < 0) || (N > MAXIMUM_N)) ? 0 : N; // A tertiary operation (using the tertiary operator (?)) is an alternative to using if-else statements.

	// Print "N := {N}." to the command line terminal.
	std::cout << "\n\nN := " << N << ".";

	// Print "N := {N}." to the file output stream.
	file << "\n\nN := " << N << ".";

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print "Computing the Nth term of the Fibonacci using recursion:" to the command line terminal.
	std::cout << "\n\nComputing the Nth term of the Fibonacci Sequence using recursion:";

	// Print "Computing the Nth term of the Fibonacci using recursion:"to the file output stream.
	file << "\n\nComputing the Nth term of the Fibonacci Sequence using recursion:";

	// Compute the Nth term of the Fibonacci Sequence using recursion, store the result in A, and print each function call in the recursive function call chain to the command line terminal.
	A = compute_Nth_fibonacci_sequence_term_using_recursion(N, std::cout, C);
```

// Reset the value of C to zero.
C = 0;

// Compute the Nth term of the Fibonacci Sequence using recursion and print each function call in the recursive function call chain to the file output stream.
compute_Nth_fibonacci_sequence_term_using_recursion(N, file, C);

// Print the value of A to the command line terminal.
std::cout << "\n\nA = fibonacci(" << N << ") = " << A << ".";

// Print the value of A to the file output stream.
file << "\n\nA = fibonacci(" << N << ") = " << A << ".";

// Print "The number of times which the recursive Fibonacci Sequence term function was called during this program runtime instance is {C}." to the command line terminal.
std::cout << "\n\nThe number of times which the recursive Fibonacci Sequence term function was called during this program runtime instance is " << C << ".";

// Print "The number of times which the recursive Fibonacci Sequence term function was called during this program runtime instance is {C}." to the file output stream.
file << "\n\nThe number of times which the recursive Fibonacci Sequence term function was called during this program runtime instance is " << C << ".";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Print "Computing the Nth term of the Fibonacci using iteration:" to the command line terminal.
std::cout << "\n\nComputing the Nth term of the Fibonacci Sequence using iteration:";

// Print "Computing the Nth term of the Fibonacci using iteration:" to the file output stream.
file << "\n\nComputing the Nth term of the Fibonacci Sequence using iteration:";

// Compute the Nth term of the Fibonacci Sequence using iteration and print each additive term of fibonacci(N) to the command line terminal.
B = compute_Nth_fibonacci_sequence_term_using_iteration(N, std::cout);

// Compute the Nth term of the Fibonacci Sequence using iteration and print each additive term of fibonacci(N) to the file output stream.

```
	compute_Nth_fibonacci_sequence_term_using_iteration(N, file);

	// Print the value of B to the command line terminal.
	std::cout << "\n\nB = fibonacci(" << N << ") = " << B << ".";

	// Print the value of B to the file output stream.
	file << "\n\nB = fibonacci(" << N << ") = " << B << ".";

	// Print a closing message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nEnd Of Program";
	std::cout << "\n--------------------------------\n\n";

	// Print a closing message to the file output stream.
	file << "\n\n--------------------------------";
	file << "\nEnd Of Program";
	file << "\n--------------------------------";

	// Close the file output stream.
	file.close();

	// Exit the program.
	return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/fibonacci_numbers_output.txt

---

```
--------------------------------
Start Of Program
--------------------------------

WARNING: the recursive function execution time increases exponentially as the value of N increases.
```

The value which was entered for N is 10.

N := 10.

Computing the Nth term of the Fibonacci Sequence using recursion:

fibonacci(10) = fibonacci(8) + fibonacci(9). // recursive case (C = 1)

fibonacci(8) = fibonacci(6) + fibonacci(7). // recursive case (C = 2)

fibonacci(6) = fibonacci(4) + fibonacci(5). // recursive case (C = 3)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 4)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 5)

fibonacci(0) = 1. // base case (C = 6)

fibonacci(1) = 1. // base case (C = 7)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 8)

fibonacci(1) = 1. // base case (C = 9)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 10)

fibonacci(0) = 1. // base case (C = 11)

fibonacci(1) = 1. // base case (C = 12)

fibonacci(5) = fibonacci(3) + fibonacci(4). // recursive case (C = 13)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 14)

fibonacci(1) = 1. // base case (C = 15)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 16)

fibonacci(0) = 1. // base case (C = 17)

fibonacci(1) = 1. // base case (C = 18)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 19)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 20)

fibonacci(0) = 1. // base case (C = 21)

fibonacci(1) = 1. // base case (C = 22)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 23)

fibonacci(1) = 1. // base case (C = 24)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 25)

fibonacci(0) = 1. // base case (C = 26)

fibonacci(1) = 1. // base case (C = 27)

fibonacci(7) = fibonacci(5) + fibonacci(6). // recursive case (C = 28)

fibonacci(5) = fibonacci(3) + fibonacci(4). // recursive case (C = 29)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 30)

fibonacci(1) = 1. // base case (C = 31)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 32)

fibonacci(0) = 1. // base case (C = 33)

fibonacci(1) = 1. // base case (C = 34)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 35)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 36)

fibonacci(0) = 1. // base case (C = 37)

fibonacci(1) = 1. // base case (C = 38)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 39)

fibonacci(1) = 1. // base case (C = 40)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 41)

fibonacci(0) = 1. // base case (C = 42)

fibonacci(1) = 1. // base case (C = 43)

fibonacci(6) = fibonacci(4) + fibonacci(5). // recursive case (C = 44)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 45)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 46)

fibonacci(0) = 1. // base case (C = 47)

fibonacci(1) = 1. // base case (C = 48)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 49)

fibonacci(1) = 1. // base case (C = 50)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 51)

fibonacci(0) = 1. // base case (C = 52)

fibonacci(1) = 1. // base case (C = 53)

fibonacci(5) = fibonacci(3) + fibonacci(4). // recursive case (C = 54)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 55)

fibonacci(1) = 1. // base case (C = 56)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 57)

fibonacci(0) = 1. // base case (C = 58)

fibonacci(1) = 1. // base case (C = 59)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 60)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 61)

fibonacci(0) = 1. // base case (C = 62)

fibonacci(1) = 1. // base case (C = 63)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 64)

fibonacci(1) = 1. // base case (C = 65)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 66)

fibonacci(0) = 1. // base case (C = 67)

fibonacci(1) = 1. // base case (C = 68)

fibonacci(9) = fibonacci(7) + fibonacci(8). // recursive case (C = 69)

fibonacci(7) = fibonacci(5) + fibonacci(6). // recursive case (C = 70)

fibonacci(5) = fibonacci(3) + fibonacci(4). // recursive case (C = 71)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 72)

fibonacci(1) = 1. // base case (C = 73)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 74)

fibonacci(0) = 1. // base case (C = 75)

fibonacci(1) = 1. // base case (C = 76)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 77)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 78)

fibonacci(0) = 1. // base case (C = 79)

fibonacci(1) = 1. // base case (C = 80)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 81)

fibonacci(1) = 1. // base case (C = 82)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 83)

fibonacci(0) = 1. // base case (C = 84)

fibonacci(1) = 1. // base case (C = 85)

fibonacci(6) = fibonacci(4) + fibonacci(5). // recursive case (C = 86)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 87)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 88)

fibonacci(0) = 1. // base case (C = 89)

fibonacci(1) = 1. // base case (C = 90)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 91)

fibonacci(1) = 1. // base case (C = 92)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 93)

fibonacci(0) = 1. // base case (C = 94)

fibonacci(1) = 1. // base case (C = 95)

fibonacci(5) = fibonacci(3) + fibonacci(4). // recursive case (C = 96)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 97)

fibonacci(1) = 1. // base case (C = 98)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 99)

fibonacci(0) = 1. // base case (C = 100)

fibonacci(1) = 1. // base case (C = 101)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 102)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 103)

fibonacci(0) = 1. // base case (C = 104)

fibonacci(1) = 1. // base case (C = 105)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 106)

fibonacci(1) = 1. // base case (C = 107)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 108)

fibonacci(0) = 1. // base case (C = 109)

fibonacci(1) = 1. // base case (C = 110)

fibonacci(8) = fibonacci(6) + fibonacci(7). // recursive case (C = 111)

fibonacci(6) = fibonacci(4) + fibonacci(5). // recursive case (C = 112)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 113)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 114)

fibonacci(0) = 1. // base case (C = 115)

fibonacci(1) = 1. // base case (C = 116)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 117)

fibonacci(1) = 1. // base case (C = 118)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 119)

fibonacci(0) = 1. // base case (C = 120)

fibonacci(1) = 1. // base case (C = 121)

fibonacci(5) = fibonacci(3) + fibonacci(4). // recursive case (C = 122)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 123)

fibonacci(1) = 1. // base case (C = 124)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 125)

fibonacci(0) = 1. // base case (C = 126)

fibonacci(1) = 1. // base case (C = 127)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 128)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 129)

fibonacci(0) = 1. // base case (C = 130)

fibonacci(1) = 1. // base case (C = 131)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 132)

fibonacci(1) = 1. // base case (C = 133)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 134)

fibonacci(0) = 1. // base case (C = 135)

fibonacci(1) = 1. // base case (C = 136)

fibonacci(7) = fibonacci(5) + fibonacci(6). // recursive case (C = 137)

fibonacci(5) = fibonacci(3) + fibonacci(4). // recursive case (C = 138)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 139)

fibonacci(1) = 1. // base case (C = 140)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 141)

fibonacci(0) = 1. // base case (C = 142)

fibonacci(1) = 1. // base case (C = 143)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 144)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 145)

fibonacci(0) = 1. // base case (C = 146)

fibonacci(1) = 1. // base case (C = 147)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 148)

fibonacci(1) = 1. // base case (C = 149)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 150)

fibonacci(0) = 1. // base case (C = 151)

fibonacci(1) = 1. // base case (C = 152)

fibonacci(6) = fibonacci(4) + fibonacci(5). // recursive case (C = 153)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 154)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 155)

fibonacci(0) = 1. // base case (C = 156)

fibonacci(1) = 1. // base case (C = 157)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 158)

fibonacci(1) = 1. // base case (C = 159)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 160)

fibonacci(0) = 1. // base case (C = 161)

fibonacci(1) = 1. // base case (C = 162)

fibonacci(5) = fibonacci(3) + fibonacci(4). // recursive case (C = 163)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 164)

fibonacci(1) = 1. // base case (C = 165)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 166)

fibonacci(0) = 1. // base case (C = 167)

fibonacci(1) = 1. // base case (C = 168)

fibonacci(4) = fibonacci(2) + fibonacci(3). // recursive case (C = 169)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 170)

fibonacci(0) = 1. // base case (C = 171)

fibonacci(1) = 1. // base case (C = 172)

fibonacci(3) = fibonacci(1) + fibonacci(2). // recursive case (C = 173)

fibonacci(1) = 1. // base case (C = 174)

fibonacci(2) = fibonacci(0) + fibonacci(1). // recursive case (C = 175)

fibonacci(0) = 1. // base case (C = 176)

fibonacci(1) = 1. // base case (C = 177)

A = fibonacci(10) = 89.

The number of times which the recursive Fibonacci Sequence term function was called during this program runtime instance is 177.

Computing the Nth term of the Fibonacci Sequence using iteration:

fibonacci(0) = 1. // i = 0

fibonacci(1) = 1. // i = 1

fibonacci(2) = 2 = fibonacci(0) + fibonacci(1) = 1 + 1. // i = 2

fibonacci(3) = 3 = fibonacci(1) + fibonacci(2) = 1 + 2. // i = 3

fibonacci(4) = 5 = fibonacci(2) + fibonacci(3) = 2 + 3. // i = 4

fibonacci(5) = 8 = fibonacci(3) + fibonacci(4) = 3 + 5. // i = 5

fibonacci(6) = 13 = fibonacci(4) + fibonacci(5) = 5 + 8. // i = 6

fibonacci(7) = 21 = fibonacci(5) + fibonacci(6) = 8 + 13. // i = 7

fibonacci(8) = 34 = fibonacci(6) + fibonacci(7) = 13 + 21. // i = 8

fibonacci(9) = 55 = fibonacci(7) + fibonacci(8) = 21 + 34. // i = 9

fibonacci(10) = 89 = fibonacci(8) + fibonacci(9) = 34 + 55. // i = 10

B = fibonacci(10) = 89.

End Of Program

This web page was last updated on 17_OCTOBER_2023. The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

[End of abridged plain-text content from FIBONACCI_NUMBERS]

GOLDEN_RATIO_APPROXIMATION

The C++ program featured in this tutorial web page computes the approximate value of the Golden Ratio by dividing the Nth term of the Fibonacci Sequence by the (N – 1)th term of the Fibonacci Sequence. Note that, in the previous sentence, N represents any natural number.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

---

golden_ratio := (1 + square_root(2)) / 5.
fibonacci(i) := 1. // i is any nonnegative integer which is smaller than 2.
fibonacci(k) := fibonacci(k - 2) + fibonacci(k - 1). // k is a natural number which is larger than or equal to 2.
golden_ratio_approximation(N) := fibonacci(N) / fibonacci(N - 1). // N is any natural number.

SOFTWARE_APPLICATION_COMPONENTS

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/golden_ratio_approximation.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/golden_ratio_approximation_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ source code into a new text editor document and save that document as the following file name:

golden_ratio_approximation.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ golden_ratio_approximation.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: After running the g++ command, run the executable file using the following command:

./app

STEP_5: Once the application is running, the following prompt will appear:

Enter a natural number which is no larger than 92:

STEP_6: Enter a value for N using the keyboard.

STEP_7: Observe program results on the command line terminal and in the output file.

PROGRAM_SOURCE_CODE

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/golden_ratio_approximation.cpp

---

```
/**
 * file: golden_ratio_approximation.cpp
 * type: C++ (source file)
 * date: 14_JUNE_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#include < iostream > // standard input (std::cin), standard output (std::cout)
#include < fstream > // file input, file output
#define MAXIMUM_N 92 // constant which represents maximum N value

/* function prototypes */
unsigned long long compute_Nth_fibonacci_sequence_term_using_iteration(int N);
long double golden_ratio_approximation(int N, std::ostream & output);

/**
 * Compute the Nth term of the Fibonacci Sequence using an iterative algorithm.
 *
 * Assume that N is an integer value.
 *
 * For each while loop iteration, i,
 * print an algebraic expression which represents the ith term of the Fibonacci Sequence.
 *
 * fibonacci(0) := 1. // The first term of the Fibonacci Sequence is 1.
 * fibonacci(1) := 1. // The second term of the Fibonacci Sequence is 1.
 * fibonacci(i) := fibonacci(i - 2) + fibonacci(i - 1). // if i is a natural number larger than 1
 */
unsigned long long compute_Nth_fibonacci_sequence_term_using_iteration(int N)
{
	int i = 0;
	unsigned long long A = 1, B = 1, C = 0;
	if ((N < 2) || (N > MAXIMUM_N)) return 1;
	for (i = 1; i < N; i += 1)
	{
	C = A;
```

```
        A = B;
        B += C;
        }
        return B;
}

/**
 * Compute the approximate value of the Golden Ratio by dividing the Nth term of the Fibonacci
Sequence by the (N - 1)th term of the Fibonacci Sequence.
 *
 * Assume that N is an integer value and that output is an output stream object.
 *
 * For each Golden Ratio approximation, i,
 * print an algebraic expression which represents the ith Golden Ratio approximation
 * (and the ith Golden Ratio approximation is produced by dividing fibonacci(i) by fibonacci(i -
1)).
 *
 * golden_ratio := (1 + square_root(2)) / 5.
 * golden_ratio_approximation(N) := fibonacci(N) / fibonacci(N - 1).
 */
long double golden_ratio_approximation(int N, std::ostream & output)
{
        unsigned long long A = 0, B = 0;
        long double C = 0.0;
        if ((N < 0) || (N > MAXIMUM_N)) N = 0;
        A = compute_Nth_fibonacci_sequence_term_using_iteration(N);
        B = compute_Nth_fibonacci_sequence_term_using_iteration(N - 1);
        C = (long double) A / B;
        output << "\n\ngolden_ratio_approximation(" << N << ") = fibonacci(" << N << ") /
fibonacci(" << N - 1 << ").";
        output << "\ngolden_ratio_approximation(" << N << ") = " << A << " / " << B << ".";
        output << "\ngolden_ratio_approximation(" << N << ") = " << C << ".";
        return C;
}

/* program entry point */
int main()
{
        // Declare two int type variables for storing whole numbers and set their initial values to
0.
        int N = 0, i = 0;

        // Declare a long double type variable for storing floating-point numbers and set its initial
value to 0.
```

```
	long double G = 0.0;

	// Declare a file output stream object.
	std::ofstream file;

	// Set the number of digits of floating-point numbers which are printed to the command
line terminal to 100 digits.
	std::cout.precision(100);

	// Set the number of digits of floating-point numbers which are printed to the file output
stream to 100 digits.
	file.precision(100);

	/**
	* If golden_ratio_approximation_output.txt does not already exist in the same directory
as golden_ratio_approximation.cpp,
	* create a new file named golden_ratio_approximation_output.txt .
	*
	* Open the plain-text file named golden_ratio_approximation_output.txt
	* and set that file to be overwritten with program data.
	*/
	file.open("golden_ratio_approximation_output.txt");

	// Print an opening message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nStart Of Program";
	std::cout << "\n--------------------------------";

	// Print an opening message to the file output stream.
	file << "--------------------------------";
	file << "\nStart Of Program";
	file << "\n--------------------------------";

	// Print "The following statements describe the data capacities of various primitive C++
data types:" to the command line terminal.
	std::cout << "\n\nThe following statements describe the data capacities of various
primitive C++ data types:";

	// Print "The following statements describe the data capacities of various primitive C++
data types:" to the file output stream.
	file << "\n\nThe following statements describe the data capacities of various primitive
C++ data types:";

	// Print the data size of an int type variable to the command line terminal.
```

```
	std::cout << "\n\nsizeof(int) = " << sizeof(int) << " byte(s).";

	// Print the data size of an int type variable to the file output stream.
	file << "\n\nsizeof(int) = " << sizeof(int) << " byte(s).";

	// Print the data size of an unsigned int type variable to the command line terminal.
	std::cout << "\n\nsizeof(unsigned int) = " << sizeof(unsigned int) << " byte(s).";

	// Print the data size of an unsigned int type variable to the file output stream.
	file << "\n\nsizeof(unsigned int) = " << sizeof(unsigned int) << " byte(s).";

	// Print the data size of a long type variable to the command line terminal.
	std::cout << "\n\nsizeof(long) = " << sizeof(long) << " byte(s).";

	// Print the data size of a long type variable to the file output stream.
	file << "\n\nsizeof(long) = " << sizeof(long) << " byte(s).";

	// Print the data size of an unsigned long type variable to the command line terminal.
	std::cout << "\n\nsizeof(unsigned long) = " << sizeof(unsigned long) << " byte(s).";

	// Print the data size of an unsigned long type variable to the file output stream.
	file << "\n\nsizeof(unsigned long) = " << sizeof(unsigned long) << " byte(s).";

	// Print the data size of a long long type variable to the command line terminal.
	std::cout << "\n\nsizeof(long long) = " << sizeof(long long) << " byte(s).";

	// Print the data size of a long long type variable to the file output stream.
	file << "\n\nsizeof(long long) = " << sizeof(long long) << " byte(s).";

	// Print the data size of an unsigned long long  type variable to the command line 
terminal.
	std::cout << "\n\nsizeof(unsigned long long) = " << sizeof(unsigned long long) << " 
byte(s).";

	// Print the data size of an unsigned long long type variable to the file output stream.
	file << "\n\nsizeof(unsigned long long) = " << sizeof(unsigned long long) << " byte(s).";

	// Print the data size of a bool type variable to the command line terminal.
	std::cout << "\n\nsizeof(bool) = " << sizeof(bool) << " byte(s).";

	// Print the data size of a bool type variable to the file output stream.
	file << "\n\nsizeof(bool) = " << sizeof(bool) << " byte(s).";

	// Print the data size of a char type variable to the command line terminal.
```

```
std::cout << "\n\nsizeof(char) = " << sizeof(char) << " byte(s).";

// Print the data size of a char type variable to the file output stream.
file << "\n\nsizeof(char) = " << sizeof(char) << " byte(s).";

// Print the data size of a float type variable to the command line terminal.
std::cout << "\n\nsizeof(float) = " << sizeof(float) << " byte(s).";

// Print the data size of a float type variable to the file output stream.
file << "\n\nsizeof(float) = " << sizeof(float) << " byte(s).";

// Print the data size of a double type variable to the command line terminal.
std::cout << "\n\nsizeof(double) = " << sizeof(double) << " byte(s).";

// Print the data size of a double type variable to the file output stream.
file << "\n\nsizeof(double) = " << sizeof(double) << " byte(s).";

// Print the data size of a long double type variable to the command line terminal.
std::cout << "\n\nsizeof(long double) = " << sizeof(long double) << " byte(s).";

// Print the data size of a long double type variable to the file output stream.
file << "\n\nsizeof(long double) = " << sizeof(long double) << " byte(s).";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Print "Enter a natural number which is no larger than {MAXIMUM_N}: " to the command line terminal.
std::cout << "\n\nEnter a natural number which is no larger than " << MAXIMUM_N << ": ";

// Scan the command line terminal for the most recent keyboard input value.
std::cin >> N;

// Print "The value which was entered for N is {N}." to the command line terminal.
std::cout << "\nThe value which was entered for N is " << N << ".";

// Print "The value which was entered for N is {N}." to the file output stream.
file << "\n\nThe value which was entered for N is " << N << ".";

// If N is smaller than 1 or if N is larger than MAXIMUM_N, set N to 1.
```

```
N = ((N < 1) || (N > MAXIMUM_N)) ? 1 : N; // A tertiary operation (using the tertiary operator (?)) is an alternative to using if-else statements.

// Print "N := {N}." to the command line terminal.
std::cout << "\n\nN := " << N << ".";

// Print "N := {N}." to the file output stream.
file << "\n\nN := " << N << ".";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Print "Computing the first N Golden Ratio approximations by dividing adjacent terms of the Fibonacci Sequence:" to the command line terminal.
std::cout << "\n\nComputing the first N Golden Ratio approximations by dividing adjacent terms of the Fibonacci Sequence:";

// Print "Computing the first N Golden Ratio approximations by dividing adjacent terms of the Fibonacci Sequence:" to the file output stream.
file << "\n\nComputing the first N Golden Ratio approximations by dividing adjacent terms of the Fibonacci Sequence:";

// Print the first N Golden Ratio approximations to the command line terminal and to the file output stream.
for (i = 1; i <= N; i += 1)
{
G = golden_ratio_approximation(i, std::cout); // Print comments to the command line terminal.
golden_ratio_approximation(i, file); // Print comments to the file output stream.
std::cout << "\nG = golden_ratio_approximation(" << i << ") = " << G << ".";
file << "\nG = golden_ratio_approximation(" << i << ") = " << G << ".";
}

// Print a closing message to the command line terminal.
std::cout << "\n\n--------------------------------";
std::cout << "\nEnd Of Program";
std::cout << "\n--------------------------------\n\n";

// Print a closing message to the file output stream.
file << "\n\n--------------------------------";
file << "\nEnd Of Program";
```

```
	file << "\n--------------------------------";

	// Close the file output stream.
	file.close();

	// Exit the program.
	return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/golden_ratio_approximation_output.txt

---

```
--------------------------------
Start Of Program
--------------------------------

The following statements describe the data capacities of various primitive C++ data types:

sizeof(int) = 4 byte(s).

sizeof(unsigned int) = 4 byte(s).

sizeof(long) = 8 byte(s).

sizeof(unsigned long) = 8 byte(s).

sizeof(long long) = 8 byte(s).

sizeof(unsigned long long) = 8 byte(s).

sizeof(bool) = 1 byte(s).

sizeof(char) = 1 byte(s).
```

sizeof(float) = 4 byte(s).

sizeof(double) = 8 byte(s).

sizeof(long double) = 16 byte(s).

The value which was entered for N is 92.

N := 92.

Computing the first N Golden Ratio approximations by dividing adjacent terms of the Fibonacci Sequence:

golden_ratio_approximation(1) = fibonacci(1) / fibonacci(0).
golden_ratio_approximation(1) = 1 / 1.
golden_ratio_approximation(1) = 1.
G = golden_ratio_approximation(1) = 1.

golden_ratio_approximation(2) = fibonacci(2) / fibonacci(1).
golden_ratio_approximation(2) = 2 / 1.
golden_ratio_approximation(2) = 2.
G = golden_ratio_approximation(2) = 2.

golden_ratio_approximation(3) = fibonacci(3) / fibonacci(2).
golden_ratio_approximation(3) = 3 / 2.
golden_ratio_approximation(3) = 1.5.
G = golden_ratio_approximation(3) = 1.5.

golden_ratio_approximation(4) = fibonacci(4) / fibonacci(3).
golden_ratio_approximation(4) = 5 / 3.
golden_ratio_approximation(4) =
1.666666666666666666630526594250483185533084906637668609619140625.
G = golden_ratio_approximation(4) =
1.666666666666666666630526594250483185533084906637668609619140625.

golden_ratio_approximation(5) = fibonacci(5) / fibonacci(4).
golden_ratio_approximation(5) = 8 / 5.
golden_ratio_approximation(5) =
1.600000000000000000021684043449710088680149056017398834228515625.

G = golden_ratio_approximation(5) =
1.6000000000000000021684043449710088680149056017398834228515625.

golden_ratio_approximation(6) = fibonacci(6) / fibonacci(5).
golden_ratio_approximation(6) = 13 / 8.
golden_ratio_approximation(6) = 1.625.
G = golden_ratio_approximation(6) = 1.625.

golden_ratio_approximation(7) = fibonacci(7) / fibonacci(6).
golden_ratio_approximation(7) = 21 / 13.
golden_ratio_approximation(7) =
1.615384615384615384623724632096042341800057329237461090087890625.
G = golden_ratio_approximation(7) =
1.615384615384615384623724632096042341800057329237461090087890625.

golden_ratio_approximation(8) = fibonacci(8) / fibonacci(7).
golden_ratio_approximation(8) = 34 / 21.
golden_ratio_approximation(8) =
1.619047619047619047624210486535645259209559299051761627197265625.
G = golden_ratio_approximation(8) =
1.619047619047619047624210486535645259209559299051761627197265625.

golden_ratio_approximation(9) = fibonacci(9) / fibonacci(8).
golden_ratio_approximation(9) = 55 / 34.
golden_ratio_approximation(9) =
1.617647058823529411758328222514791150388191454112529754638671875.
G = golden_ratio_approximation(9) =
1.617647058823529411758328222514791150388191454112529754638671875.

golden_ratio_approximation(10) = fibonacci(10) / fibonacci(9).
golden_ratio_approximation(10) = 89 / 55.
golden_ratio_approximation(10) =
1.618181818181818181824095648213557296912767924368381500244140625.
G = golden_ratio_approximation(10) =
1.618181818181818181824095648213557296912767924368381500244140625.

golden_ratio_approximation(11) = fibonacci(11) / fibonacci(10).
golden_ratio_approximation(11) = 144 / 89.
golden_ratio_approximation(11) =
1.61797752808988764042751051785984373054816387593746185302734375.
G = golden_ratio_approximation(11) =
1.61797752808988764042751051785984373054816387593746185302734375.

golden_ratio_approximation(12) = fibonacci(12) / fibonacci(11).

golden_ratio_approximation(12) = 233 / 144.
golden_ratio_approximation(12) =
1.6180555555555555555073687923339775807107798755168914794921875.
G = golden_ratio_approximation(12) =
1.6180555555555555555073687923339775807107798755168914794921875.

golden_ratio_approximation(13) = fibonacci(13) / fibonacci(12).
golden_ratio_approximation(13) = 377 / 233.
golden_ratio_approximation(13) =
1.618025751072961373414754726596243017411325126886367797851562 5.
G = golden_ratio_approximation(13) =
1.6180257510729613734147547265962430174113251268863677978515625.

golden_ratio_approximation(14) = fibonacci(14) / fibonacci(13).
golden_ratio_approximation(14) = 610 / 377.
golden_ratio_approximation(14) =
1.61803713527851458883928537080265641634468920528888702392578125.
G = golden_ratio_approximation(14) =
1.61803713527851458883928537080265641634468920528888702392578125.

golden_ratio_approximation(15) = fibonacci(15) / fibonacci(14).
golden_ratio_approximation(15) = 987 / 610.
golden_ratio_approximation(15) =
1.6180327868852459016453873563712306804518448188900947570800781 25.
G = golden_ratio_approximation(15) =
1.618032786885245901645387356371230680451844818890094757080078125.

golden_ratio_approximation(16) = fibonacci(16) / fibonacci(15).
golden_ratio_approximation(16) = 1597 / 987.
golden_ratio_approximation(16) =
1.6180344478216818642803132011209754637093283236026763916015625.
G = golden_ratio_approximation(16) =
1.6180344478216818642803132011209754637093283236026763916015625.

golden_ratio_approximation(17) = fibonacci(17) / fibonacci(16).
golden_ratio_approximation(17) = 2584 / 1597.
golden_ratio_approximation(17) =
1.61803381340012523482464745772091418984928168356418609619140625.
G = golden_ratio_approximation(17) =
1.61803381340012523482464745772091418984928168356418609619140625.

golden_ratio_approximation(18) = fibonacci(18) / fibonacci(17).
golden_ratio_approximation(18) = 4181 / 2584.

golden_ratio_approximation(18) =
1.61803405572755417958334678285581276213633827865123748779296875.
G = golden_ratio_approximation(18) =
1.61803405572755417958334678285581276213633827865123748779296875.

golden_ratio_approximation(19) = fibonacci(19) / fibonacci(18).
golden_ratio_approximation(19) = 6765 / 4181.
golden_ratio_approximation(19) =
1.618033963166706529538362013820318452417268417775630950927734375.
G = golden_ratio_approximation(19) =
1.618033963166706529538362013820318452417268417775630950927734375.

golden_ratio_approximation(20) = fibonacci(20) / fibonacci(19).
golden_ratio_approximation(20) = 10946 / 6765.
golden_ratio_approximation(20) =
1.618033998521803399858222383134176425301120616495609283447265625.
G = golden_ratio_approximation(20) =
1.618033998521803399858222383134176425301120616495609283447265625.

golden_ratio_approximation(21) = fibonacci(21) / fibonacci(20).
golden_ratio_approximation(21) = 17711 / 10946.
golden_ratio_approximation(21) =
1.6180339850173579389902567271519728819839656352996826171875.
G = golden_ratio_approximation(21) =
1.6180339850173579389902567271519728819839656352996826171875.

golden_ratio_approximation(22) = fibonacci(22) / fibonacci(21).
golden_ratio_approximation(22) = 28657 / 17711.
golden_ratio_approximation(22) =
1.618033990175597086548682501661033938944456167519092559814453125.
G = golden_ratio_approximation(22) =
1.618033990175597086548682501661033938944456167519092559814453125.

golden_ratio_approximation(23) = fibonacci(23) / fibonacci(22).
golden_ratio_approximation(23) = 46368 / 28657.
golden_ratio_approximation(23) =
1.6180339882053250515070441650777866016142070293426513671875.
G = golden_ratio_approximation(23) =
1.6180339882053250515070441650777866016142070293426513671875.

golden_ratio_approximation(24) = fibonacci(24) / fibonacci(23).
golden_ratio_approximation(24) = 75025 / 46368.
golden_ratio_approximation(24) =
1.618033988957902001375697975671386075191549025475978851318359375.

G = golden_ratio_approximation(24) =
1.618033988957902001375697975671386075191549025475978851318359375.

golden_ratio_approximation(25) = fibonacci(25) / fibonacci(24).
golden_ratio_approximation(25) = 121393 / 75025.
golden_ratio_approximation(25) =
1.618033988670443185618752490739780114381574094295501708984375.
G = golden_ratio_approximation(25) =
1.618033988670443185618752490739780114381574094295501708984375.

golden_ratio_approximation(26) = fibonacci(26) / fibonacci(25).
golden_ratio_approximation(26) = 196418 / 121393.
golden_ratio_approximation(26) =
1.6180339887802426828040947004438976364326663315296173095703125.
G = golden_ratio_approximation(26) =
1.6180339887802426828040947004438976364326663315296173095703125.

golden_ratio_approximation(27) = fibonacci(27) / fibonacci(26).
golden_ratio_approximation(27) = 317811 / 196418.
golden_ratio_approximation(27) =
1.61803398873830300689659333901460058768861927092075347900390625.
G = golden_ratio_approximation(27) =
1.61803398873830300689659333901460058768861927092075347900390625.

golden_ratio_approximation(28) = fibonacci(28) / fibonacci(27).
golden_ratio_approximation(28) = 514229 / 317811.
golden_ratio_approximation(28) =
1.61803398875432253765059564809547509867115877568721771240234375.
G = golden_ratio_approximation(28) =
1.61803398875432253765059564809547509867115877568721771240234375.

golden_ratio_approximation(29) = fibonacci(29) / fibonacci(28).
golden_ratio_approximation(29) = 832040 / 514229.
golden_ratio_approximation(29) =
1.6180339887482036212960900822821486144675873219966888427734375.
G = golden_ratio_approximation(29) =
1.6180339887482036212960900822821486144675873219966888427734375.

golden_ratio_approximation(30) = fibonacci(30) / fibonacci(29).
golden_ratio_approximation(30) = 1346269 / 832040.
golden_ratio_approximation(30) =
1.6180339887505408393887640361441526692942716181278228759765625.
G = golden_ratio_approximation(30) =
1.6180339887505408393887640361441526692942716181278228759765625.

golden_ratio_approximation(31) = fibonacci(31) / fibonacci(30).
golden_ratio_approximation(31) = 2178309 / 1346269.
golden_ratio_approximation(31) =
1.618033988749648101573667957620017432418535463511943817138671875.
G = golden_ratio_approximation(31) =
1.618033988749648101573667957620017432418535463511943817138671875.

golden_ratio_approximation(32) = fibonacci(32) / fibonacci(31).
golden_ratio_approximation(32) = 3524578 / 2178309.
golden_ratio_approximation(32) =
1.618033988749989097034702456578969531619804911315441131591796875.
G = golden_ratio_approximation(32) =
1.618033988749989097034702456578969531619804911315441131591796875.

golden_ratio_approximation(33) = fibonacci(33) / fibonacci(32).
golden_ratio_approximation(33) = 5702887 / 3524578.
golden_ratio_approximation(33) =
1.618033988749858848358274820977698027490987442433834075927734375.
G = golden_ratio_approximation(33) =
1.618033988749858848358274820977698027490987442433834075927734375.

golden_ratio_approximation(34) = fibonacci(34) / fibonacci(33).
golden_ratio_approximation(34) = 9227465 / 5702887.
golden_ratio_approximation(34) =
1.618033988749908598926523228822560440676170401275157928466796875.
G = golden_ratio_approximation(34) =
1.618033988749908598926523228822560440676170401275157928466796875.

golden_ratio_approximation(35) = fibonacci(35) / fibonacci(34).
golden_ratio_approximation(35) = 14930352 / 9227465.
golden_ratio_approximation(35) =
1.618033988749889595898205640889244705249438993632793426513671875.
G = golden_ratio_approximation(35) =
1.618033988749889595898205640889244705249438993632793426513671875.

golden_ratio_approximation(36) = fibonacci(36) / fibonacci(35).
golden_ratio_approximation(36) = 24157817 / 14930352.
golden_ratio_approximation(36) =
1.618033988749896854414909996844329498344450257718563079833984375.
G = golden_ratio_approximation(36) =
1.618033988749896854414909996844329498344450257718563079833984375.

golden_ratio_approximation(37) = fibonacci(37) / fibonacci(36).

golden_ratio_approximation(37) = 39088169 / 24157817.
golden_ratio_approximation(37) =
1.618033988749894081893114516912390854486147873103618621826171875.
G = golden_ratio_approximation(37) =
1.618033988749894081893114516912390854486147873103618621826171875.

golden_ratio_approximation(38) = fibonacci(38) / fibonacci(37).
golden_ratio_approximation(38) = 63245986 / 39088169.
golden_ratio_approximation(38) =
1.618033988749895140941796600753121992966043762862682342529296875.
G = golden_ratio_approximation(38) =
1.618033988749895140941796600753121992966043762862682342529296875.

golden_ratio_approximation(39) = fibonacci(39) / fibonacci(38).
golden_ratio_approximation(39) = 102334155 / 63245986.
golden_ratio_approximation(39) =
1.6180339887498947364259660464114176647854037582874298095703125.
G = golden_ratio_approximation(39) =
1.6180339887498947364259660464114176647854037582874298095703125.

golden_ratio_approximation(40) = fibonacci(40) / fibonacci(39).
golden_ratio_approximation(40) = 165580141 / 102334155.
golden_ratio_approximation(40) =
1.618033988749894890924775625595799510847427882254123687744140625.
G = golden_ratio_approximation(40) =
1.618033988749894890924775625595799510847427882254123687744140625.

golden_ratio_approximation(41) = fibonacci(41) / fibonacci(40).
golden_ratio_approximation(41) = 267914296 / 165580141.
golden_ratio_approximation(41) =
1.618033988749894831944177442384358300841995514929294586181640625.
G = golden_ratio_approximation(41) =
1.618033988749894831944177442384358300841995514929294586181640625.

golden_ratio_approximation(42) = fibonacci(42) / fibonacci(41).
golden_ratio_approximation(42) = 433494437 / 267914296.
golden_ratio_approximation(42) =
1.6180339887498948543871624128343000847962684929370880126953125.
G = golden_ratio_approximation(42) =
1.6180339887498948543871624128343000847962684929370880126953125.

golden_ratio_approximation(43) = fibonacci(43) / fibonacci(42).
golden_ratio_approximation(43) = 701408733 / 433494437.

golden_ratio_approximation(43) =
1.6180339887498948458219652501988150561373913660645484924316406625.
G = golden_ratio_approximation(43) =
1.6180339887498948458219652501988150561373913660645484924316406625.

golden_ratio_approximation(44) = fibonacci(44) / fibonacci(43).
golden_ratio_approximation(44) = 1134903170 / 701408733.
golden_ratio_approximation(44) =
1.618033988749894849074571767655328358159749768674373626708984375.
G = golden_ratio_approximation(44) =
1.618033988749894849074571767655328358159749768674373626708984375.

golden_ratio_approximation(45) = fibonacci(45) / fibonacci(44).
golden_ratio_approximation(45) = 1836311903 / 1134903170.
golden_ratio_approximation(45) =
1.6180339887498948478819493779212734807515516877174377441406 25.
G = golden_ratio_approximation(45) =
1.618033988749894847881949377921273480751551687717437744140625.

golden_ratio_approximation(46) = fibonacci(46) / fibonacci(45).
golden_ratio_approximation(46) = 2971215073 / 1836311903.
golden_ratio_approximation(46) =
1.6180339887498948483156302469154752543545328080654144287109375.
G = golden_ratio_approximation(46) =
1.6180339887498948483156302469154752543545328080654144287109375.

golden_ratio_approximation(47) = fibonacci(47) / fibonacci(46).
golden_ratio_approximation(47) = 4807526976 / 2971215073.
golden_ratio_approximation(47) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(47) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(48) = fibonacci(48) / fibonacci(47).
golden_ratio_approximation(48) = 7778742049 / 4807526976.
golden_ratio_approximation(48) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(48) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(49) = fibonacci(49) / fibonacci(48).
golden_ratio_approximation(49) = 12586269025 / 7778742049.
golden_ratio_approximation(49) =
1.618033988749894848207210029666924810953787527978420257568359375.

G = golden_ratio_approximation(49) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(50) = fibonacci(50) / fibonacci(49).
golden_ratio_approximation(50) = 20365011074 / 12586269025.
golden_ratio_approximation(50) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(50) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(51) = fibonacci(51) / fibonacci(50).
golden_ratio_approximation(51) = 32951280099 / 20365011074.
golden_ratio_approximation(51) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(51) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(52) = fibonacci(52) / fibonacci(51).
golden_ratio_approximation(52) = 53316291173 / 32951280099.
golden_ratio_approximation(52) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(52) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(53) = fibonacci(53) / fibonacci(52).
golden_ratio_approximation(53) = 86267571272 / 53316291173.
golden_ratio_approximation(53) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(53) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(54) = fibonacci(54) / fibonacci(53).
golden_ratio_approximation(54) = 139583862445 / 86267571272.
golden_ratio_approximation(54) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(54) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(55) = fibonacci(55) / fibonacci(54).
golden_ratio_approximation(55) = 225851433717 / 139583862445.
golden_ratio_approximation(55) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(55) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(56) = fibonacci(56) / fibonacci(55).
golden_ratio_approximation(56) = 365435296162 / 225851433717.
golden_ratio_approximation(56) =
1.6180339887498948482072100296669248109537875279784202575683593 75.
G = golden_ratio_approximation(56) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(57) = fibonacci(57) / fibonacci(56).
golden_ratio_approximation(57) = 591286729879 / 365435296162.
golden_ratio_approximation(57) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(57) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(58) = fibonacci(58) / fibonacci(57).
golden_ratio_approximation(58) = 956722026041 / 591286729879.
golden_ratio_approximation(58) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(58) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(59) = fibonacci(59) / fibonacci(58).
golden_ratio_approximation(59) = 1548008755920 / 956722026041.
golden_ratio_approximation(59) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(59) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(60) = fibonacci(60) / fibonacci(59).
golden_ratio_approximation(60) = 2504730781961 / 1548008755920.
golden_ratio_approximation(60) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(60) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(61) = fibonacci(61) / fibonacci(60).
golden_ratio_approximation(61) = 4052739537881 / 2504730781961.
golden_ratio_approximation(61) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(61) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(62) = fibonacci(62) / fibonacci(61).

golden_ratio_approximation(62) = 6557470319842 / 4052739537881.
golden_ratio_approximation(62) =
1.6180339887498948482072100296669248109537875279784202575683593 75.
G = golden_ratio_approximation(62) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(63) = fibonacci(63) / fibonacci(62).
golden_ratio_approximation(63) = 10610209857723 / 6557470319842.
golden_ratio_approximation(63) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(63) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(64) = fibonacci(64) / fibonacci(63).
golden_ratio_approximation(64) = 17167680177565 / 10610209857723.
golden_ratio_approximation(64) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(64) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(65) = fibonacci(65) / fibonacci(64).
golden_ratio_approximation(65) = 27777890035288 / 17167680177565.
golden_ratio_approximation(65) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(65) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(66) = fibonacci(66) / fibonacci(65).
golden_ratio_approximation(66) = 44945570212853 / 27777890035288.
golden_ratio_approximation(66) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(66) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(67) = fibonacci(67) / fibonacci(66).
golden_ratio_approximation(67) = 72723460248141 / 44945570212853.
golden_ratio_approximation(67) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(67) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(68) = fibonacci(68) / fibonacci(67).
golden_ratio_approximation(68) = 117669030460994 / 72723460248141.

golden_ratio_approximation(68) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(68) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(69) = fibonacci(69) / fibonacci(68).
golden_ratio_approximation(69) = 190392490709135 / 117669030460994.
golden_ratio_approximation(69) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(69) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(70) = fibonacci(70) / fibonacci(69).
golden_ratio_approximation(70) = 308061521170129 / 190392490709135.
golden_ratio_approximation(70) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(70) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(71) = fibonacci(71) / fibonacci(70).
golden_ratio_approximation(71) = 498454011879264 / 308061521170129.
golden_ratio_approximation(71) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(71) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(72) = fibonacci(72) / fibonacci(71).
golden_ratio_approximation(72) = 806515533049393 / 498454011879264.
golden_ratio_approximation(72) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(72) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(73) = fibonacci(73) / fibonacci(72).
golden_ratio_approximation(73) = 1304969544928657 / 806515533049393.
golden_ratio_approximation(73) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(73) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(74) = fibonacci(74) / fibonacci(73).
golden_ratio_approximation(74) = 2111485077978050 / 1304969544928657.
golden_ratio_approximation(74) =
1.618033988749894848207210029666924810953787527978420257568359375.

G = golden_ratio_approximation(74) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(75) = fibonacci(75) / fibonacci(74).
golden_ratio_approximation(75) = 3416454622906707 / 2111485077978050.
golden_ratio_approximation(75) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(75) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(76) = fibonacci(76) / fibonacci(75).
golden_ratio_approximation(76) = 5527939700884757 / 3416454622906707.
golden_ratio_approximation(76) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(76) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(77) = fibonacci(77) / fibonacci(76).
golden_ratio_approximation(77) = 8944394323791464 / 5527939700884757.
golden_ratio_approximation(77) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(77) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(78) = fibonacci(78) / fibonacci(77).
golden_ratio_approximation(78) = 14472334024676221 / 8944394323791464.
golden_ratio_approximation(78) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(78) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(79) = fibonacci(79) / fibonacci(78).
golden_ratio_approximation(79) = 23416728348467685 / 14472334024676221.
golden_ratio_approximation(79) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(79) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(80) = fibonacci(80) / fibonacci(79).
golden_ratio_approximation(80) = 37889062373143906 / 23416728348467685.
golden_ratio_approximation(80) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(80) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(81) = fibonacci(81) / fibonacci(80).
golden_ratio_approximation(81) = 61305790721611591 / 37889062373143906.
golden_ratio_approximation(81) =
1.6180339887498948482072100296669248109537875279784202575683593750.
G = golden_ratio_approximation(81) =
1.6180339887498948482072100296669248109537875279784202575683593750.

golden_ratio_approximation(82) = fibonacci(82) / fibonacci(81).
golden_ratio_approximation(82) = 99194853094755497 / 61305790721611591.
golden_ratio_approximation(82) =
1.6180339887498948482072100296669248109537875279784202575683593750.
G = golden_ratio_approximation(82) =
1.6180339887498948482072100296669248109537875279784202575683593750.

golden_ratio_approximation(83) = fibonacci(83) / fibonacci(82).
golden_ratio_approximation(83) = 160500643816367088 / 99194853094755497.
golden_ratio_approximation(83) =
1.6180339887498948482072100296669248109537875279784202575683593750.
G = golden_ratio_approximation(83) =
1.6180339887498948482072100296669248109537875279784202575683593750.

golden_ratio_approximation(84) = fibonacci(84) / fibonacci(83).
golden_ratio_approximation(84) = 259695496911122585 / 160500643816367088.
golden_ratio_approximation(84) =
1.6180339887498948482072100296669248109537875279784202575683593750.
G = golden_ratio_approximation(84) =
1.6180339887498948482072100296669248109537875279784202575683593750.

golden_ratio_approximation(85) = fibonacci(85) / fibonacci(84).
golden_ratio_approximation(85) = 420196140727489673 / 259695496911122585.
golden_ratio_approximation(85) =
1.6180339887498948482072100296669248109537875279784202575683593750.
G = golden_ratio_approximation(85) =
1.6180339887498948482072100296669248109537875279784202575683593750.

golden_ratio_approximation(86) = fibonacci(86) / fibonacci(85).
golden_ratio_approximation(86) = 679891637638612258 / 420196140727489673.
golden_ratio_approximation(86) =
1.6180339887498948482072100296669248109537875279784202575683593750.
G = golden_ratio_approximation(86) =
1.6180339887498948482072100296669248109537875279784202575683593750.

golden_ratio_approximation(87) = fibonacci(87) / fibonacci(86).

golden_ratio_approximation(87) = 1100087778366101931 / 679891637638612258.
golden_ratio_approximation(87) =
1.6180339887498948482072100296669248109537875279784202575683593 75.
G = golden_ratio_approximation(87) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(88) = fibonacci(88) / fibonacci(87).
golden_ratio_approximation(88) = 1779979416004714189 / 1100087778366101931.
golden_ratio_approximation(88) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(88) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(89) = fibonacci(89) / fibonacci(88).
golden_ratio_approximation(89) = 2880067194370816120 / 1779979416004714189.
golden_ratio_approximation(89) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(89) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(90) = fibonacci(90) / fibonacci(89).
golden_ratio_approximation(90) = 4660046610375530309 / 2880067194370816120.
golden_ratio_approximation(90) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(90) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(91) = fibonacci(91) / fibonacci(90).
golden_ratio_approximation(91) = 7540113804746346429 / 4660046610375530309.
golden_ratio_approximation(91) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(91) =
1.618033988749894848207210029666924810953787527978420257568359375.

golden_ratio_approximation(92) = fibonacci(92) / fibonacci(91).
golden_ratio_approximation(92) = 12200160415121876738 / 7540113804746346429.
golden_ratio_approximation(92) =
1.618033988749894848207210029666924810953787527978420257568359375.
G = golden_ratio_approximation(92) =
1.618033988749894848207210029666924810953787527978420257568359375.

End Of Program

[End of abridged plain-text content from GOLDEN_RATIO_APPROXIMATION]

NUMBERS

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/number_sets_diagram.png

---

The following terms and their respective definitions describe all types of numbers. A number is a piece of information which non-ambiguously represents one of limitlessly many unique abstract patterns.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

ONE: the smallest natural number; the length of the line segment whose endpoints are adjacent integers on some dimensional axis of a Cartesian grid.

NUMBER: a piece of information which represents exactly one finite quantity; a piece of information which can be approximated to some arbitrary degree of precision or else represented exactly by a particular corresponding finite sequence binary digits (and a binary digit is the smallest unit of information verbatim transmissible information).

INFINITY: the (hypothetical or actual) instantiation of limitlessly many copies of exactly one pattern or the (hypothetical or actual) instantiation of limitlessly many unique patterns.

NATURAL_NUMBER: an element of the indefinitely large set (and hypothetically infinitely large set) whose elements consist exclusively of one and of every unique sum comprised of one being added to itself for some finite number of additions.

---

length("") = 0. // zero (i.e. the quantity which symbolically represents the detection of some noumenon)
length("X") = 1. // smallest natural number (i.e. the quantity which symbolically represents the detection of some phenomenon)
length("XX") = 2 = (1 + 1). // second smallest natural number
length("XXX") = 3 = (2 + 1) = (1 + 2) = ((1 + 1) + 1) = (1 + (1 + 1)). // third smallest natural number

---

INTEGER: an element of the indefinitely large set (and hypothetically infinitely large set) whose elements consist exclusively of each natural number, each natural number multiplied by negative one, and zero.

---

array_of_integers := [-5, 3, 0, 2].

RATIONAL_NUMBER: an element of the indefinitely large set (and hypothetically infinitely large set) whose elements consist exclusively of each integer and each ratio (A/B) such that A represents any integer while B represents any integer other than zero.

---

is_rational_number(1/3) = true.
is_rational_number(1/1) = true.
is_rational_number(square_root(2)) = false.
is_rational_number(square_root(1)) = true. // square_root(1) = 1.
is_rational_number(square_root(0)) = true. // square_root(0) = 0.
is_rational_number(square_root(-1)) = false. // i := square_root(-1). // i is an imaginary number.
Each rational number is a real number.
is_rational_number(0/1) = true. // (0/1) = 0.
is_rational_number(0/0) = false. // Infinity is not a number.
is_rational_number(1/0) = false. // Infinity is not a number.

IRRATIONAL_NUMBER: an element of the indefinitely large set (and hypothetically infinitely large set) whose elements consist exclusively of real numbers which are not rational numbers.

An example of an irrational number is Pi.

REAL_NUMBER an element of the indefinitely large set (and hypothetically infinitely large set) whose elements consist exclusively of numbers which each represents a specific point along some dimensional axis of a Cartesian grid.

IMAGINARY_NUMBER an element of the indefinitely large set (and hypothetically infinitely large set) whose elements consist exclusively of numbers which are each the product of the square root of negative one multiplied by some real number.

---

i := square_root(-1). // imaginary number
(i * i) = -1. // real number
((i * i) * i) := ((-1) * i). // imaginary number

COMPLEX_NUMBER: the sum of a real number and an imaginary number.

(2 * i) + 3. // complex number
(2 * i). // imaginary number
(1 * i). // imaginary number
(0 * i) = 0. // imaginary number to the left of the equal sign and real number to the right of the equal sign
(0 * i) - 8 = -8. // complex number to the left of the equal sign and real number to the right of the equal sign

---

ALGEBRAIC_REAL_NUMBER: a number which is the root of a non-zero polynomial equation such that the coefficients of that polynomial equation are rational numbers.

An example of an algebraic number is the Golden Ratio.

---

golden_ratio := (1 + (5 ^ (1/2))) / 2.

some_polynomial := y = (x ^ 2) − x − 1.

proof that golden_ratio is a root of some_polynomial:

0 = (((1 + (5 ^ (1/2))) / 2)) ^ 2) − ((1 + (5 ^ (1/2))) / 2) − 1.

TRANSCENDENTAL_NUMBER: a real or complex number which is not an algebraic number.

An example of a transcendental number is Euler’s Number.

This web page was last updated on 18_OCTOBER_2023. The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

[End of abridged plain-text content from NUMBERS]

SQUARE_ROOT_APPROXIMATION

The C++ program featured in this tutorial web page computes the approximate square root of a real number using an iterative algorithm.

Note that, even though the program accepts negative real numbers as input values, the square root approximation function returns negative real number values for negative real number input values. Technically, if a negative real number is that function’s input, the value returned by that function should be a positive real number multiplied by the square root of negative one (and such a return value is an imaginary number instead of a real number).

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

Y := square_root(X). // Y = X ^ (1/2).
(Y * Y) = X. // X = Y ^ 2.

SOFTWARE_APPLICATION_COMPONENTS

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/square_root_approximation.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/square_root_approximation_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ source code into a new text editor document and save that document as the following file name:

square_root_approximation.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ square_root_approximation.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: After running the g++ command, run the executable file using the following command:

./app

STEP_5: Once the application is running, the following prompt will appear:

Enter a real number (represented using only base-ten digits with an optional radix and with an optional negative sign), x, which is no larger than 100:

STEP_6: Enter a value for x using the keyboard.

STEP_7: Statements showing program throughput and the value returned by the square root function which computes the approximate value of x raised to the power of 0.5 will be printed to the command line terminal and to the file output stream and then the following prompt will appear:

Would you like to continue inputting program values? (Enter 1 if YES. Enter 0 if NO):

STEP_8: Enter a value according to your preference until you decide to close the program (and save your program data to the output text file).

PROGRAM_SOURCE_CODE

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/square_root_approximation.cpp

---

```
/**
 * file: square_root_approximation.cpp
 * type: C++ (source file)
 * date: 19_JUNE_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#include  < iostream > // standard input (std::cin), standard output (std::cout)
#include  < fstream > // file input, file output
```

```
#define MAXIMUM_X 100 // constant which represents maximum absolute value of the program
input value
#define E 0.00000001 // constant which represents the degree of accuracy of the square root
approximation

/* function prototypes */
float absolute_value(float x);
long double compute_square_root_of_real_number(float x, std::ostream & output);

/**
 * Return the absolute value of a real number input, x.
 */
float absolute_value(float x)
{
	if (x < 0) return -1 * x;
	return x;
}

/**
 * Compute the approximate square root of a real number, x, using an iterative method.
 *
 * The square root of x is x raised to the power of 0.5 (i.e. x ^ (1/2)).
 *
 * Assume that x is a float type value and that output is an output stream object.
 *
 * This function returns a value whose data type is long double (which is a floating-point
number).
 */
long double compute_square_root_of_real_number(float x, std::ostream & output)
{
	int i = 0;
	float original_x = x, absolute_value_of_original_x = 0.0;
	long double S = 0.0, Y = 1.0;
	x = absolute_value(x);
	absolute_value_of_original_x = x;
	x = (x > MAXIMUM_X) ? 0 : x; // If x is out of range, then set x to 0.
	S = x;
	output << "\n\nx = " << x << ". // real number to take the square root of";
	output << "\nS = " << S << ". // variable for storing the approximate square root of x";
	output << "\nY = " << Y < E)
	{
	S = (S + Y) / 2;
	Y = absolute_value_of_original_x / S;
	output << "\n\ni := " << i << ".";
```

```
	output << "\nS := ((S + Y) / 2) = " << S << ".";
	output << "\nY := (absolute_value_of_original_x / S) = " << Y << ".";
	i += 1;
	}
	if (original_x < 0) return -1 * S;
	return S;
}

/* program entry point */
int main()
{
	// Declare a float type variable and set its initial value to zero.
	float x = 0.0;

	// Declare a double type variable and set its initial value to zero.
	long double A = 0.0;

	// Declare a variable for storing the program user's answer of whether or not to continue
inputting values.
	int input_additional_values = 1;

	// Declare a file output stream object.
	std::ofstream file;

	// Set the number of digits of floating-point numbers which are printed to the command
line terminal to 100 digits.
	std::cout.precision(100);

	// Set the number of digits of floating-point numbers which are printed to the file output
stream to 100 digits.
	file.precision(100);

	/**
	* If square_root_approximation_output.txt does not already exist in the same directory as
square_root_approximation.cpp,
	* create a new file named square_root_approximation_output.txt.
	*
	* Open the plain-text file named square_root_approximation_output.txt
	* and set that file to be overwritten with program data.
	*/
	file.open("square_root_approximation_output.txt");

	// Print an opening message to the command line terminal.
	std::cout << "\n\n--------------------------------";
```

```
        std::cout << "\nStart Of Program";
        std::cout << "\n-------------------------------";

        // Print an opening message to the file output stream.
        file << "-------------------------------";
        file << "\nStart Of Program";
        file << "\n-------------------------------";

        // Prompt the user to enter an x value as many times as the user chooses to.
        while (input_additional_values != 0)
        {
        // Print "Enter a real number (represented using only base-ten digits with an optional
radix and with an optional negative sign), x, which is no larger than {MAXIMUM_X}: " to the
command line terminal.
        std::cout << "\n\nEnter a real number (represented using only base-ten digits with an
optional radix and with an optional negative sign), x, which is no larger than " << MAXIMUM_X
<> x;

        // Print "The value which was entered for x is {x}." to the command line terminal.
        std::cout << "\nThe value which was entered for x is " << x << ".";

        // Print "The value which was entered for x is {x}." to the file output stream.
        file << "\n\nThe value which was entered for x is " << x << ".";

        // Print a horizontal line to the command line terminal.
        std::cout << "\n\n-------------------------------";

        // Print a horizontal line to the command line terminal.
        file << "\n\n-------------------------------";

        // Print "Computing the approximate square root of x:" to the command line terminal.
        std::cout << "\n\nComputing the approximate square root of x:";

        // Print "Computing the approximate square root of x:" to the file output stream.
        file << "\n\nComputing the approximate square root of x:";

        // Compute the approximate square root of x using Heron's Method, print the
computational steps to the command line terminal, and store the function result in A.
        A = compute_square_root_of_real_number(x, std::cout);

        // Compute the approximate square root of x using Heron's Method and print the
computational steps to the file output stream.
        compute_square_root_of_real_number(x, file);
```

```
	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print "A = approximate_square_root({x}) = {A}." to the command line terminal.
	std::cout << "\n\nA = approximate_square_root(" << x << ") = " << A << ".";

	// Print "A = approximate_square_root({x}) = {A}." to the file output stream.
	file << "\n\nA = approximate_square_root(" << x << ") = " << A << ".";

	// Print "(A * A) = " << {(A * A)} << ". // the approximate value of x" to the command line
terminal.
	std::cout << "\n\n(A * A) = " << (A * A) << ". // the approximate absolute value of x";

	// Print "(A * A) = " << {(A * A)} << ". // the approximate value of x" to the file output
stream.
	file << "\n\n(A * A) = " << (A * A) << ". // the approximate absolute value of x";

	// Ask the user whether or not to continue inputing values.
	std::cout <> input_additional_values;

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";
	}

	// Print a closing message to the command line terminal.
	std::cout << "\nEnd Of Program";
	std::cout << "\n--------------------------------\n\n";

	// Print a closing message to the file output stream.
	file << "\nEnd Of Program";
	file << "\n--------------------------------";

	// Close the file output stream.
	file.close();

	// Exit the program.
	return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/square_root_approximation_output.txt

---

```
--------------------------------
Start Of Program
--------------------------------

The value which was entered for x is -25.

--------------------------------

Computing the approximate square root of x:

x = 25. // real number to take the square root of
S = 25. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

i := 0.
S := ((S + Y) / 2) = 13.
Y := (absolute_value_of_original_x / S) =
1.923076923076923076881376839519788290999713353812694549560546875.

i := 1.
S := ((S + Y) / 2) =
7.4615384615384615384948985283841693672002293169498443603515625.
Y := (absolute_value_of_original_x / S) =
3.35051546391752577322940831461295374538167379796504974365234375.

i := 2.
S := ((S + Y) / 2) =
5.406026962727993655753733204250011112890206277370452880859375.
Y := (absolute_value_of_original_x / S) =
4.624468241161801379014717472415441079647280275821685791015625.
```

i := 3.
S := ((S + Y) / 2) = 5.01524760194489751738422533833272609626874327659606933593750.
Y := (absolute_value_of_original_x / S) =
4.98479875456300049445940159031920302368234843015670776367187 5.

i := 4.
S := ((S + Y) / 2) =
5.000023178253949005921813464325964559975545853376388549804687 5.
Y := (absolute_value_of_original_x / S) =
4.99997682185349678722630084592992716352455317974090576171875.

i := 5.
S := ((S + Y) / 2) =
5.00000000005372289679089758962504674855154007673263549804687 5.
Y := (absolute_value_of_original_x / S) =
4.999999999462771032091024103749532514484599232673645019531 25.

A = approximate_square_root(-25) =
-5.00000000005372289679089758962504674855154007673263549804687 5.

(A * A) = 25.00000000005372289679089758962504674855154007673263549804687 5. // the approximate absolute value of x

----

The value which was entered for x is 100.

Computing the approximate square root of x:

x = 100. // real number to take the square root of
S = 100. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

i := 0.
S := ((S + Y) / 2) = 50.5.
Y := (absolute_value_of_original_x / S) =
1.98019801980198019799618569525279099252657033503055572509765625.

i := 1.

S := ((S + Y) / 2) =
26.24009900990099009888967263037784505286253988742828369140625.
Y := (absolute_value_of_original_x / S) =
3.81096123007263465711814964809178718496696092188358306884765625.

i := 2.
S := ((S + Y) / 2) =
15.025530119986812377895490921986265675514005124568939208984375.
Y := (absolute_value_of_original_x / S) =
6.6553392260670379662786111385486265135114081203937530517578125.

i := 3.
S := ((S + Y) / 2) =
10.84043467302692517230389146476454698131419718265533447265625.
Y := (absolute_value_of_original_x / S) =
9.22472234889428614745821022324889781884849071502685546875.

i := 4.
S := ((S + Y) / 2) =
10.032578510960605659881050844006722400081343948841094970703125.
Y := (absolute_value_of_original_x / S) =
9.96752728032478032237084786260084001696668565273284912109375.

i := 5.
S := ((S + Y) / 2) =
10.00005289564269299155963022229798298212699592113494873046875.
Y := (absolute_value_of_original_x / S) =
9.999947104637100430378493509664394878200255334377288818359375.

i := 6.
S := ((S + Y) / 2) =
10.00000000013989671053538099698698715656064450740814208984375.
Y := (absolute_value_of_original_x / S) =
9.99999999986010328946461900301301284343935549259185791015625.

A = approximate_square_root(100) =
10.00000000013989671053538099698698715656064450740814208984375.

(A * A) = 100.000000002797934210707619939739743131212890148162841796875. // the approximate absolute value of x

----

The value which was entered for x is 3.1400001049041748046875.

Computing the approximate square root of x:

x = 3.1400001049041748046875. // real number to take the square root of
S = 3.1400001049041748046875. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

i := 0.
S := ((S + Y) / 2) = 2.07000005245208740234375.
Y := (absolute_value_of_original_x / S) =
1.51690822480153237488721684744774620412499643862247467041015625.

i := 1.
S := ((S + Y) / 2) =
1.793454138626809888615483423723873102062498219311237335205078125.
Y := (absolute_value_of_original_x / S) =
1.750811485655480336811294639343117296448326669633388519287109375.

i := 2.
S := ((S + Y) / 2) =
1.772132812141145112713389031533495199255412444472312927246609375.
Y := (absolute_value_of_original_x / S) =
1.771876285677669133245167032431055531560559757053852081298828125.

i := 3.
S := ((S + Y) / 2) = 1.772004548909407123033488140606550587108358740806579589843 75.
Y := (absolute_value_of_original_x / S) =
1.77200453962531322702270153435222255211556330323219299316 40625.

A = approximate_square_root(3.1400001049041748046875) =
1.772004548909407123033488140606550587108358740806579589843 75.

(A * A) = 3.14000012135563142073695075406902788017760030925273895263671875. // the approximate absolute value of x

----

The value which was entered for x is -16.

Computing the approximate square root of x:

x = 16. // real number to take the square root of
S = 16. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

i := 0.
S := ((S + Y) / 2) = 8.5.
Y := (absolute_value_of_original_x / S) =
1.882352941176470588241671777485208849611808545887470245361328125.

i := 1.
S := ((S + Y) / 2) =
5.191176470588235294066625780118329203105531632900238037109375.
Y := (absolute_value_of_original_x / S) =
3.082152974504249291765045626334540429525077342987060546875.

i := 2.
S := ((S + Y) / 2) =
4.1366647225462422929158357032264348163153044879436492919921875.
Y := (absolute_value_of_original_x / S) =
3.867850327050802394790451899098115973174571990966796875.

i := 3.
S := ((S + Y) / 2) = 4.00225752479852234406998423565937628154642879962921142578125.
Y := (absolute_value_of_original_x / S) =
3.99774374858735659192532363448435717145912349224090576171875.

i := 4.
S := ((S + Y) / 2) = 4.00000063669293946799765393507186672650277614593505859375.
Y := (absolute_value_of_original_x / S) =
3.99999936330716187649937654047249679933884181082248687744140625.

i := 5.
S := ((S + Y) / 2) =
4.000000000000050672140095020523631319520063698291778564453125.
Y := (absolute_value_of_original_x / S) =
3.999999999999949327859904979476368680479936301708221435546875.

A = approximate_square_root(-16) =
-4.000000000000050672140095020523631319520063698291778564453125.

(A * A) = 16.0000000000004053771207601641890505561605095863342285 15625. // the approximate absolute value of x

----

The value which was entered for x is 0.

Computing the approximate square root of x:

x = 0. // real number to take the square root of
S = 0. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

----

A = approximate_square_root(0) = 0.

(A * A) = 0. // the approximate absolute value of x

The value which was entered for x is -1.

Computing the approximate square root of x:

x = 1. // real number to take the square root of
S = 1. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

----

A = approximate_square_root(-1) = -1.

(A * A) = 1. // the approximate absolute value of x

The value which was entered for x is 1.

Computing the approximate square root of x:

x = 1. // real number to take the square root of
S = 1. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

----

A = approximate_square_root(1) = 1.

(A * A) = 1. // the approximate absolute value of x

The value which was entered for x is -100.

Computing the approximate square root of x:

x = 100. // real number to take the square root of
S = 100. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

i := 0.
S := ((S + Y) / 2) = 50.5.
Y := (absolute_value_of_original_x / S) =
1.980198019801980197996185695252790992526570335030555725097656 25.

i := 1.
S := ((S + Y) / 2) =
26.240099009900990098889672630377845052862539887428283691406 25.
Y := (absolute_value_of_original_x / S) =
3.810961230072634657118149648091787184966960921883583068847656 25.

i := 2.
S := ((S + Y) / 2) =
15.025530119986812377895490921986265675514005124568939208984375.
Y := (absolute_value_of_original_x / S) =
6.655339226067037966278611138548626513511408120393753051757812 5.

i := 3.
S := ((S + Y) / 2) =
10.84043467302692517230389146476454698131419718265533447265625.
Y := (absolute_value_of_original_x / S) =
9.22472234889428614745821022324889781884849071502685546875.

i := 4.
S := ((S + Y) / 2) =
10.032578510960605659881050844006722400081343948841094970703125.
Y := (absolute_value_of_original_x / S) =
9.96752728032478032237084786260084001696668565273284912109375.

i := 5.
S := ((S + Y) / 2) =
10.00005289564269299155963022229798298212699592113494873046875.
Y := (absolute_value_of_original_x / S) =
9.999947104637100430378493509664394878200255334377288818359375.

i := 6.
S := ((S + Y) / 2) =
10.00000000013989671053538099698698715656064450740814208984375.
Y := (absolute_value_of_original_x / S) =
9.99999999986010328946461900301301284343935549259185791015625.

A = approximate_square_root(-100) =
-10.00000000013989671053538099698698715656064450740814208984375.

(A * A) = 100.000000002797934210707619939739743131212890148162841796875. // the approximate absolute value of x

----

The value which was entered for x is 0.5.

Computing the approximate square root of x:

x = 0.5. // real number to take the square root of
S = 0.5. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

----

A = approximate_square_root(0.5) = 0.5.

(A * A) = 0.25. // the approximate absolute value of x

The value which was entered for x is 0.3330000042915344238281\25.

Computing the approximate square root of x:

x = 0.33300000429153442382812\5. // real number to take the square root of
S = 0.33300000429153442382812\5. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

----

A = approximate_square_root(0.333000004291534423828125) =
0.333000004291534423828125.

(A * A) = 0.110889002858161944686798960901796817779541015625. // the approximate absolute value of x

----

The value which was entered for x is 64.

Computing the approximate square root of x:

x = 64. // real number to take the square root of
S = 64. // variable for storing the approximate square root of x
Y = 1. // number to add to S before dividing S by 2 for each while loop iteration, i

i := 0.
S := ((S + Y) / 2) = 32.5.
Y := (absolute_value_of_original_x / S) =
1.969230769230769230795918822707335493760183453559875488281\25.

i := 1.
S := ((S + Y) / 2) = 17.234615384615384614530597673365264199674129486083984375.
Y := (absolute_value_of_original_x / S) =
3.713456817674626199606013887688504837569780647754669189453125.

i := 2.
S := ((S + Y) / 2) =
10.47403610114500540663462491153268274501897394657135009765625.
Y := (absolute_value_of_original_x / S) =
6.1103474708287113035913573622082139991107396781444549560546875.

i := 3.
S := ((S + Y) / 2) =
8.292191785986858355329831571367549258866347372531890869140625.
Y := (absolute_value_of_original_x / S) =
7.7181041697750993844928668607963118120096623897552490234375.

i := 4.
S := ((S + Y) / 2) = 8.00514797788097886947766834708772876183502376079559326171875.
Y := (absolute_value_of_original_x / S) =
7.994855332698205459783513671112586962408386170864105224609375.

i := 5.
S := ((S + Y) / 2) = 8.00000165528959216419691014010595608851872384548187255859375.
Y := (absolute_value_of_original_x / S) =
7.999998344710750333534654554767939771409146487712860107421875.

i := 6.
S := ((S + Y) / 2) = 8.0000000000001712484321014784427461563609540462493896484375.
Y := (absolute_value_of_original_x / S) =
7.9999999999998287515678985215572538436390459537506103515625.

A = approximate_square_root(64) =
8.0000000000001712484321014784427461563609540462493896484375.

(A * A) = 64.000000000002739974913623655083938501775264739990234375. // the approximate absolute value of x

----
End Of Program

This web page was last updated on 17_OCTOBER_2023. The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

[End of abridged plain-text content from SQUARE_ROOT_APPROXIMATION]

CUBE_ROOT_APPROXIMATION

The C++ program featured in this tutorial web page computes the approximate cube root of a real number using an iterative algorithm.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

Y := cube_root(X). // Y = X ^ (1/3).
(Y * Y * Y) = X. // X = Y ^ 3.

SOFTWARE_APPLICATION_COMPONENTS

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/cube_root_approximation.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/cube_root_approximation_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ source code into a new text editor document and save that document as the following file name:

cube_root_approximation.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ cube_root_approximation.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: After running the g++ command, run the executable file using the following command:

./app

STEP_5: Once the application is running, the following prompt will appear:

Enter a real number (represented using only base-ten digits with an optional radix and with an optional negative sign), x, which is no larger than 100:

STEP_6: Enter a value for N using the keyboard.

STEP_7: Statements showing program throughput and the value returned by the cube root function which computes the approximate value of x raised to the power of (1/3) will be printed to the command line terminal and to the file output stream and then the following prompt will appear:

Would you like to continue inputting program values? (Enter 1 if YES. Enter 0 if NO):

STEP_8: Enter a value according to your preference until you decide to close the program (and save your program data to the output file).

PROGRAM_SOURCE_CODE

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/cube_root_approximation.cpp

---

```
/**
 * file: cube_root_approximation.cpp
 * type: C++ (source file)
 * date: 19_JUNE_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#include < iostream > // standard input (std::cin), standard output (std::cout)
#include < fstream > // file input, file output
#define MAXIMUM_X 1000 // constant which represents maximum absolute value of the program input value
#define E 0.00000001 // constant which represents the degree of accuracy of the square root approximation

/* function prototypes */
float absolute_value(float x);
long double difference(long double n, long double b);
long double compute_cube_root_of_real_number(float x, std::ostream & output);

/**
 * Return the absolute value of a real number input, x.
 */
float absolute_value(float x)
{
	if (x  (b * b * b)) return (n - (b * b * b));
	return ((b * b * b) - n);
}

/**
 * Return the absolute value of (n - (b *  b * b)).
 */
long double difference(long double n, long double b)
{
	if (n > (b * b * b)) return (n - (b * b * b));
	return ((b * b * b) - n);
}
```

```
/**
 * Compute the approximate cube root of a real number, x, using an iterative method.
 *
 * The cube root of x is x raised to the power of (1/3).
 *
 * Assume that x is a float type value and that output is an output stream object.
 *
 * This function returns a value whose data type is long double (which is a floating-point
number).
 */
long double compute_cube_root_of_real_number(float x, std::ostream & output)
{
        int i = 0;
        float original_x = x;
        long double A = 0.0, B = 0.0, C = 0.0, epsilon = 0.0;
        x = absolute_value(x);
        x = ((x > MAXIMUM_X) || (x < 1)) ? 0 : x; // If x is out of range, then set x to 0. Also, to
avoid an infinite loop as a result of the absolute value of x being too small, set x to 0 if the
absolute value of x is smaller than 1.
        C = x;
        output << "\n\nC = " << C << ". // real number to take the cube root of";
        output << "\nB = " << B << ". // variable for storing the approximate cube root of x";
        output << "\nA = " << A << ". // number to add to C before dividing the sum of A and C by
2 for each while loop iteration, i";
        output << "\nepsilon = " << epsilon << ". // variable for storing the difference between the
input value and B raised to the power of 3";
        while (true)
        {
        output << "\n\ni := " << i << ".";
        output << "\nC := " << C << ".";
        output << "\nA := " << A << ".";
        B = (A + C) / 2;
        epsilon = difference(x, B);
        output << "\nB := (A + C) / 2 = " << B << ".";
        output << "\nepsilon = difference(x , B) = " << epsilon << ".";
        if (epsilon <= E)
        {
        if (original_x  x) C = B;
        else A = B;
        i += 1;
        }
}
```

```
/* program entry point */
int main()
{
	// Declare a float type variable and set its initial value to zero.
	float x = 0.0;

	// Declare a double type variable and set its initial value to zero.
	long double S = 0.0;

	// Declare a variable for storing the program user's answer of whether or not to continue 
inputting values.
	int input_additional_values = 1;

	// Declare a file output stream object.
	std::ofstream file;

	// Set the number of digits of floating-point numbers which are printed to the command 
line terminal to 100 digits.
	std::cout.precision(100);

	// Set the number of digits of floating-point numbers which are printed to the file output 
stream to 100 digits.
	file.precision(100);

	/**
	* If cube_root_approximation_output.txt does not already exist in the same directory as 
cube_root_approximation.cpp,
	* create a new file named cube_root_approximation_output.txt.
	*
	* Open the plain-text file named cube_root_approximation_output.txt
	* and set that file to be overwritten with program data.
	*/
	file.open("cube_root_approximation_output.txt");

	// Print an opening message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nStart Of Program";
	std::cout << "\n--------------------------------";

	// Print an opening message to the file output stream.
	file << "--------------------------------";
	file << "\nStart Of Program";
	file << "\n--------------------------------";
```

```
	// Prompt the user to enter an x value as many times as the user chooses to.
	while (input_additional_values != 0)
	{
	// Print "Enter a real number (represented using only base-ten digits with an optional radix and with an optional negative sign), x, which is no larger than {MAXIMUM_X}: " to the command line terminal.
	std::cout << "\n\nEnter a real number (represented using only base-ten digits with an optional radix and with an optional negative sign), x, which is no larger than " << MAXIMUM_X <> x;

	// Print "The value which was entered for x is {x}." to the command line terminal.
	std::cout << "\nThe value which was entered for x is " << x << ".";

	// Print "The value which was entered for x is {x}." to the file output stream.
	file << "\n\nThe value which was entered for x is " << x << ".";

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print "Computing the approximate cube root of x:" to the command line terminal.
	std::cout << "\n\nComputing the approximate cube root of x:";

	// Print "Computing the approximate cube root of x:" to the file output stream.
	file << "\n\nComputing the approximate cube root of x:";

	// Compute the approximate cube root of x using the Bijection Method, print the computational steps to the command line terminal, and store the function result in S.
	S = compute_cube_root_of_real_number(x, std::cout);

	// Compute the approximate square root of x using Heron's Method and print the computational steps to the file output stream.
	compute_cube_root_of_real_number(x, file);

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print "S = approximate_cube_root({x}) = {S}." to the command line terminal.
	std::cout << "\n\nS = approximate_cube_root(" << x << ") = " << S << ".";
```

```
	// Print "S = approximate_cube_root({x}) = {S}." to the file output stream.
	file << "\n\nS = approximate_cube_root(" << x << ") = " << S << ".";

	// Print "(S * S * S) = " << {(S * S * S)} << ". // the approximate value of x" to the command line terminal.
	std::cout << "\n\n(S * S * S) = " << (S * S * S) << ". // the approximate absolute value of x";

	// Print "(S * S * S) = " << {(S * S * S)} << ". // the approximate value of x" to the command line terminal.
	std::cout << "\n\n(S * S * S) = " << (S * S * S) << ". // the approximate absolute value of x";

	// Ask the user whether or not to continue inputing values.
	std::cout <> input_additional_values;

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";
	}

	// Print a closing message to the command line terminal.
	std::cout << "\nEnd Of Program";
	std::cout << "\n--------------------------------\n\n";

	// Print a closing message to the file output stream.
	file << "\nEnd Of Program";
	file << "\n--------------------------------";

	// Close the file output stream.
	file.close();

	// Exit the program.
	return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/cube_root_approximation_output.txt

---

Start Of Program

The value which was entered for x is 8.

Computing the approximate cube root of x:

C = 8. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 8.
A := 0.
B := (A + C) / 2 = 4.
epsilon = difference(x , B) = 56.

i := 1.
C := 4.
A := 0.
B := (A + C) / 2 = 2.
epsilon = difference(x , B) = 0.

S = approximate_cube_root(8) = 2.

The value which was entered for x is 125.

Computing the approximate cube root of x:

C = 125. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 125.
A := 0.
B := (A + C) / 2 = 62.5.
epsilon = difference(x , B) = 244015.625.

i := 1.
C := 62.5.
A := 0.
B := (A + C) / 2 = 31.25.
epsilon = difference(x , B) = 30392.578125.

i := 2.
C := 31.25.
A := 0.
B := (A + C) / 2 = 15.625.
epsilon = difference(x , B) = 3689.697265625.

i := 3.
C := 15.625.
A := 0.
B := (A + C) / 2 = 7.8125.
epsilon = difference(x , B) = 351.837158203125.

i := 4.
C := 7.8125.
A := 0.
B := (A + C) / 2 = 3.90625.
epsilon = difference(x , B) = 65.395355224609375.

i := 5.

C := 7.8125.
A := 3.90625.
B := (A + C) / 2 = 5.859375.
epsilon = difference(x , B) = 76.165676116943359375.

i := 6.
C := 5.859375.
A := 3.90625.
B := (A + C) / 2 = 4.8828125.
epsilon = difference(x , B) = 8.584678173065185546875.

i := 7.
C := 5.859375.
A := 4.8828125.
B := (A + C) / 2 = 5.37109375.
epsilon = difference(x , B) = 29.948793351650238037109375.

i := 8.
C := 5.37109375.
A := 4.8828125.
B := (A + C) / 2 = 5.126953125.
epsilon = difference(x , B) = 9.765286929905414581298828125.

i := 9.
C := 5.126953125.
A := 4.8828125.
B := (A + C) / 2 = 5.0048828125.
epsilon = difference(x , B) = 0.366568681783974170684814453125.

i := 10.
C := 5.0048828125.
A := 4.8828125.
B := (A + C) / 2 = 4.94384765625.
epsilon = difference(x , B) = 4.164306548773311078548431396484375.

i := 11.
C := 5.0048828125.
A := 4.94384765625.
B := (A + C) / 2 = 4.974365234375.
epsilon = difference(x , B) = 1.912767149406136013567447662353515625.

i := 12.
C := 5.0048828125.
A := 4.974365234375.

B := (A + C) / 2 = 4.9896240234375.
epsilon = difference(x , B) = 0.7765844459299842128530144691467285156 25.

i := 13.
C := 5.0048828125.
A := 4.9896240234375.
B := (A + C) / 2 = 4.99725341796875.
epsilon = difference(x , B) = 0.2058805173703603941248729825019836425781 25.

i := 14.
C := 5.0048828125.
A := 4.99725341796875.
B := (A + C) / 2 = 5.001068115234375.
epsilon = difference(x , B) = 0.0801257568490143512462964281439781188964 84375.

i := 15.
C := 5.001068115234375.
A := 4.99725341796875.
B := (A + C) / 2 = 4.9991607666015625.
epsilon = difference(x , B) = 0.0629319407834394439760217210277915000915 52734375.

i := 16.
C := 5.001068115234375.
A := 4.9991607666015625.
B := (A + C) / 2 = 5.00011444091796875.
epsilon = difference(x , B) = 0.0085832653000106340357433509780094027519 22607421875.

i := 17.
C := 5.00011444091796875.
A := 4.9991607666015625.
B := (A + C) / 2 = 4.999637603759765625.
epsilon = difference(x , B) =
0.0271777480996479581243363554676761850714 68353271484375.

i := 18.
C := 5.00011444091796875.
A := 4.999637603759765625.
B := (A + C) / 2 = 4.9998760223388671875.
epsilon = difference(x , B) =
0.0092980940299596318010522111308091552928 09009552001953125.

i := 19.
C := 5.00011444091796875.
A := 4.9998760223388671875.

B := (A + C) / 2 = 4.99999523162841796875.
epsilon = difference(x , B) =
0.00035762752759194160745437329751439392566680908203125.

i := 20.
C := 5.00011444091796875.
A := 4.99999523162841796875.
B := (A + C) / 2 = 5.000054836273193359375.
epsilon = difference(x , B) =
0.0041127655949197150508922504741349257528781890869140625.

i := 21.
C := 5.000054836273193359375.
A := 4.99999523162841796875.
B := (A + C) / 2 = 5.0000250339508056640625.
epsilon = difference(x , B) =
0.001877555710920887632742193318335921503603458404541015625.

i := 22.
C := 5.0000250339508056640625.
A := 4.99999523162841796875.
B := (A + C) / 2 = 5.00001013278961181640625.
epsilon = difference(x , B) =
0.00075996076098865106285273895991849713027477264404296875.

i := 23.
C := 5.00001013278961181640625.
A := 4.99999523162841796875.
B := (A + C) / 2 = 5.000002682209014892578125.
epsilon = difference(x , B) =
0.000201165784030642169621927450862131081521511077880859375.

i := 24.
C := 5.000002682209014892578125.
A := 4.99999523162841796875.
B := (A + C) / 2 = 4.9999989569187164306640625.
epsilon = difference(x , B) = 7.823107994742173332269885577261447906494140625e-05.

i := 25.
C := 5.000002682209014892578125.
A := 4.9999989569187164306640625.
B := (A + C) / 2 = 5.00000081956386566162109375.
epsilon = difference(x , B) = 6.14672999998955305045456043444573879241943359375e-05.

i := 26.
C := 5.000000819563865661621093750.
A := 4.99999895691871643066406250.
B := (A + C) / 2 = 4.99999988824129104614257812 5.
epsilon = difference(x , B) = 8.381902984189171235129833803512156009674072265625e-06.

i := 27.
C := 5.000000819563865661621093750.
A := 4.99999988824129104614257812 5.
B := (A + C) / 2 = 5.00000035390257835388183593 75.
epsilon = difference(x , B) =
2.6542695255246662178194583248114213347434997558593 75e-05.

i := 28.
C := 5.00000035390257835388183593 75.
A := 4.99999988824129104614257812 5.
B := (A + C) / 2 = 5.00000012107193470001220703125.
epsilon = difference(x , B) = 9.080395322380585554356002830900251865386962890625e-06.

i := 29.
C := 5.00000012107193470001220703125.
A := 4.99999988824129104614257812 5.
B := (A + C) / 2 = 5.00000000465661287307739257812 5.
epsilon = difference(x , B) =
3.492459657999356293910750537179410457611083984375e-07.

i := 30.
C := 5.00000000465661287307739257812 5.
A := 4.99999988824129104614257812 5.
B := (A + C) / 2 = 4.99999994644895195960998535156 25.
epsilon = difference(x , B) =
4.016328560015047788311903786961920559406280517578125e-06.

i := 31.
C := 5.00000000465661287307739257812 5.
A := 4.99999994644895195960998535156 25.
B := (A + C) / 2 = 4.99999997555278241634368896484375.
epsilon = difference(x , B) =
1.833541309802233509884672457701526582241058349609375e-06.

i := 32.
C := 5.00000000465661287307739257812 5.
A := 4.99999997555278241634368896484375.
B := (A + C) / 2 = 4.99999999010469764471054077148437 5.

epsilon = difference(x , B) =
7.421476751826028284142466873163357377052307128906250e-07.

i := 33.
C := 5.00000000465661287307739257812 5.
A := 4.99999999010469764471054077148 4375.
B := (A + C) / 2 = 4.99999999738065525889396667480 46875.
epsilon = difference(x , B) =
1.9645085547886909083104001183528453111648559570312 5e-07.

i := 34.
C := 5.000000004656612873077392578125.
A := 4.9999999973806552588939666748046875.
B := (A + C) / 2 = 5.0000000010186340659856796264648 4375.
epsilon = difference(x , B) =
7.639755496974265369658496638294309377670288085937 5e-08.

i := 35.
C := 5.00000000101863406598567962646484375.
A := 4.9999999973806552588939666748046875.
B := (A + C) / 2 = 4.999999999199644662439823150634765625.
epsilon = difference(x , B) =
6.002665031007436979848534974735230207443237304687 5e-08.

i := 36.
C := 5.00000000101863406598567962646484375.
A := 4.999999999199644662439823150634765625.
B := (A + C) / 2 = 5.0000000001091393642127513885498046875.
epsilon = difference(x , B) = 8.18545231595635414123535156 25e-09.

S = approximate_cube_root(125) = 5.0000000001091393642127513885498046875.

The value which was entered for x is -8.

Computing the approximate cube root of x:

C = 8. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x

A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 8.
A := 0.
B := (A + C) / 2 = 4.
epsilon = difference(x , B) = 56.

i := 1.
C := 4.
A := 0.
B := (A + C) / 2 = 2.
epsilon = difference(x , B) = 0.

S = approximate_cube_root(-8) = -2.

The value which was entered for x is 1000.

Computing the approximate cube root of x:

C = 1000. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 1000.
A := 0.
B := (A + C) / 2 = 500.
epsilon = difference(x , B) = 124999000.

i := 1.
C := 500.

A := 0.
B := (A + C) / 2 = 250.
epsilon = difference(x , B) = 15624000.

i := 2.
C := 250.
A := 0.
B := (A + C) / 2 = 125.
epsilon = difference(x , B) = 1952125.

i := 3.
C := 125.
A := 0.
B := (A + C) / 2 = 62.5.
epsilon = difference(x , B) = 243140.625.

i := 4.
C := 62.5.
A := 0.
B := (A + C) / 2 = 31.25.
epsilon = difference(x , B) = 29517.578125.

i := 5.
C := 31.25.
A := 0.
B := (A + C) / 2 = 15.625.
epsilon = difference(x , B) = 2814.697265625.

i := 6.
C := 15.625.
A := 0.
B := (A + C) / 2 = 7.8125.
epsilon = difference(x , B) = 523.162841796875.

i := 7.
C := 15.625.
A := 7.8125.
B := (A + C) / 2 = 11.71875.
epsilon = difference(x , B) = 609.325408935546875.

i := 8.
C := 11.71875.
A := 7.8125.
B := (A + C) / 2 = 9.765625.

epsilon = difference(x , B) = 68.677425384521484375.

i := 9.
C := 11.71875.
A := 9.765625.
B := (A + C) / 2 = 10.7421875.
epsilon = difference(x , B) = 239.590346813201904296875.

i := 10.
C := 10.7421875.
A := 9.765625.
B := (A + C) / 2 = 10.25390625.
epsilon = difference(x , B) = 78.12229543924331665039062 5.

i := 11.
C := 10.25390625.
A := 9.765625.
B := (A + C) / 2 = 10.009765625.
epsilon = difference(x , B) = 2.932549454271793365478515625.

i := 12.
C := 10.009765625.
A := 9.765625.
B := (A + C) / 2 = 9.8876953125.
epsilon = difference(x , B) = 33.314452390186488628387451171875.

i := 13.
C := 10.009765625.
A := 9.8876953125.
B := (A + C) / 2 = 9.94873046875.
epsilon = difference(x , B) = 15.3021371952490881085395812988281 25.

i := 14.
C := 10.009765625.
A := 9.94873046875.
B := (A + C) / 2 = 9.979248046875.
epsilon = difference(x , B) = 6.21267556743987370282411575317382 8125.

i := 15.
C := 10.009765625.
A := 9.979248046875.
B := (A + C) / 2 = 9.9945068359375.
epsilon = difference(x , B) = 1.647044138962883152998983860015869140625.

i := 16.
C := 10.009765625.
A := 9.9945068359375.
B := (A + C) / 2 = 10.00213623046875.
epsilon = difference(x , B) = 0.64100605479211480997037142515182495117 1875.

i := 17.
C := 10.00213623046875.
A := 9.9945068359375.
B := (A + C) / 2 = 9.998321533203125.
epsilon = difference(x , B) = 0.503455526267515551808173768222332000732421875.

i := 18.
C := 10.00213623046875.
A := 9.998321533203125.
B := (A + C) / 2 = 10.0002288818359375.
epsilon = difference(x , B) = 0.068666122400085072285946807824075222015380859375.

i := 19.
C := 10.0002288818359375.
A := 9.998321533203125.
B := (A + C) / 2 = 9.99927520751953125.
epsilon = difference(x , B) = 0.217421984797183664994690843741409480571746826171875.

i := 20.
C := 10.0002288818359375.
A := 9.99927520751953125.
B := (A + C) / 2 = 9.999752044677734375.
epsilon = difference(x , B) =
0.0743847522396770544084176890464732423424720764160 15625.

i := 21.
C := 10.0002288818359375.
A := 9.999752044677734375.
B := (A + C) / 2 = 9.9999904632568359375.
epsilon = difference(x , B) = 0.00286102022073553285963498638011515140533447265625.

i := 22.
C := 10.0002288818359375.
A := 9.9999904632568359375.
B := (A + C) / 2 = 10.00010967254638671875.
epsilon = difference(x , B) = 0.032902124759357720407138003793079406023025512695 3125.

i := 23.

C := 10.0001096725463867187 5.

A := 9.9999904632568359375.
B := (A + C) / 2 = 10.000050067901611328125.
epsilon = difference(x , B) =
0.015020445687367101061937546546687372028827667236328125.

i := 24.
C := 10.000050067901611328125.
A := 9.9999904632568359375.
B := (A + C) / 2 = 10.0000202655792236328125.
epsilon = difference(x , B) =
0.0060796860879092085028219116793479770421981811523437 5.

i := 25.
C := 10.0000202655792236328125.
A := 9.9999904632568359375.
B := (A + C) / 2 = 10.00000536441802978515625.
epsilon = difference(x , B) =
0.00160932627224513735697541960689704865217208862304687 5.

i := 26.
C := 10.00000536441802978515625.
A := 9.9999904632568359375.
B := (A + C) / 2 = 9.999997913837432861328125.
epsilon = difference(x , B) = 0.0006258486395793738665815908461809158325195312 5.

i := 27.
C := 10.00000536441802978515625.
A := 9.999997913837432861328125.
B := (A + C) / 2 = 10.0000016391277313232421875.
epsilon = difference(x , B) = 0.000491738399991642440363648347556591033935546875.

i := 28.
C := 10.0000016391277313232421875.
A := 9.999997913837432861328125.
B := (A + C) / 2 = 9.99999977648258209228515625.
epsilon = difference(x , B) = 6.7055223873513369881038670428097248077392578125e-05.

i := 29.
C := 10.0000016391277313232421875.
A := 9.99999977648258209228515625.
B := (A + C) / 2 = 10.000000707805156707763671875.
epsilon = difference(x , B) =
0.000212341562041973297425556659849137067794799804687 5.

i := 30.
C := 10.000000707805156707763671875.
A := 9.999999776482582092285156250.
B := (A + C) / 2 = 10.0000002421438694000244140625.
epsilon = difference(x , B) = 7.2643162579044684434848022647202014923095703125e-05.

i := 31.
C := 10.0000002421438694000244140625.
A := 9.999999776482582092285156250.
B := (A + C) / 2 = 10.00000000931322574615478515625.
epsilon = difference(x , B) = 2.793967726399948503512860042974352836608886718750e-06.

i := 32.
C := 10.00000000931322574615478515625.
A := 9.999999776482582092285156250.
B := (A + C) / 2 = 9.999999892897903919219970703125.
epsilon = difference(x , B) =
3.21306284801203823064952302956953644752502441406250e-05.

i := 33.
C := 10.00000000931322574615478515625.
A := 9.999999892897903919219970703125.
B := (A + C) / 2 = 9.9999999511055648326873779296875.
epsilon = difference(x , B) =
1.46683304784178680790773796616122126579284667968750e-05.

i := 34.
C := 10.00000000931322574615478515625.
A := 9.9999999511055648326873779296875.
B := (A + C) / 2 = 9.99999998020939528942108154296875.
epsilon = difference(x , B) = 5.9371814014608226273139734985306859016418457031250e-06.

i := 35.
C := 10.00000000931322574615478515625.
A := 9.99999998020939528942108154296875.
B := (A + C) / 2 = 9.999999994761310517787933349609375.
epsilon = difference(x , B) = 1.57160684383095272664832009468227624893188476562500e-06.

i := 36.
C := 10.00000000931322574615478515625.
A := 9.999999994761310517787933349609375.
B := (A + C) / 2 = 10.0000000020372681319713592529296875.
epsilon = difference(x , B) = 6.1118043975794122957267973106354475021362304687500e-07.

i := 37.
C := 10.0000000020372681319713592529296875.
A := 9.99999999476131051778793334960937 5.
B := (A + C) / 2 = 9.99999999839928932487964630126953125.
epsilon = difference(x , B) = 4.802132024805949583878827979788184165945898 4375e-07.

i := 38.
C := 10.0000000020372681319713592529296875.
A := 9.99999999839928932487964630126953125.
B := (A + C) / 2 = 10.000000000218278728425502777099609375.
epsilon = difference(x , B) = 6.5483618527650833129882 8125e-08.

i := 39.
C := 10.000000000218278728425502777099609375.
A := 9.99999999839928932487964630126953125.
B := (A + C) / 2 = 9.9999999993087840266525745391845703125.
epsilon = difference(x , B) = 2.0736479200422763824462890625e-07.

i := 40.
C := 10.000000000218278728425502777099609375.
A := 9.9999999993087840266525745391845703125.
B := (A + C) / 2 = 9.99999999976353137753903865814208984375.
epsilon = difference(x , B) = 7.0940586738288402557373046875e-08.

i := 41.
C := 10.000000000218278728425502777099609375.
A := 9.99999999976353137753903865814208984375.
B := (A + C) / 2 = 9.999999999990905052982270717620849609375.
epsilon = difference(x , B) = 2.7284841053187847137451171875e-09.

S = approximate_cube_root(1000) = 9.999999999990905052982270717620849609375.

The value which was entered for x is 3.1400001049041748046875.

Computing the approximate cube root of x:

C = 3.1400001049041748046875. // real number to take the cube root of

B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 3.1400001049041748046875.
A := 0.
B := (A + C) / 2 = 1.57000005245208740234375.
epsilon = difference(x , B) =
0.7298932829632888677397900556442777997290249913930892944335 9375.

i := 1.
C := 1.57000005245208740234375.
A := 0.
B := (A + C) / 2 = 0.785000026226043701171875.
epsilon = difference(x , B) =
2.656263431420741845606983688732327664183685556054115295410156 25.

i := 2.
C := 1.57000005245208740234375.
A := 0.785000026226043701171875.
B := (A + C) / 2 = 1.1775000393390655517578125.
epsilon = difference(x , B) =
1.507388831897588567926282721032293920870870351791381835937 5.

i := 3.
C := 1.57000005245208740234375.
A := 1.1775000393390655517578125.
B := (A + C) / 2 = 1.37375004589557647705078125.
epsilon = difference(x , B) =
0.54747387045390128986269151489807427424238994717597961425781 25.

i := 4.
C := 1.57000005245208740234375.
A := 1.37375004589557647705078125.
B := (A + C) / 2 = 1.471875049173831939697265625.
epsilon = difference(x , B) =
0.048693787686813938932542389359525714098708704113960266113281 25.

i := 5.
C := 1.471875049173831939697265625.
A := 1.37375004589557647705078125.

B := (A + C) / 2 = 1.42281254753470420837402343750.
epsilon = difference(x , B) =
0.259664721704114639022273336443902280734619125723838806152343750.

i := 6.
C := 1.47187504917383193969726562500.
A := 1.42281254753470420837402343750.
B := (A + C) / 2 = 1.44734379835426807403564453125.
epsilon = difference(x , B) =
0.108098424503967965914011384320758679677965119481086730957031250.

i := 7.
C := 1.47187504917383193969726562500.
A := 1.44734379835426807403564453125.
B := (A + C) / 2 = 1.45960942376405000686645507812500.
epsilon = difference(x , B) =
0.030361093709303276725469666885715014359448105096817016601562500.

i := 8.
C := 1.47187504917383193969726562500.
A := 1.45960942376405000686645507812500.
B := (A + C) / 2 = 1.46574223646894097328186035156250.
epsilon = difference(x , B) =
0.009000961172711657940631591046098947117570787668228149414062500.

i := 9.
C := 1.46574223646894097328186035156250.
A := 1.45960942376405000686645507812500.
B := (A + C) / 2 = 1.46267583011649549007415771484375.
epsilon = difference(x , B) =
0.010721326223448964399392879798256217327434569597244262695312500.

i := 10.
C := 1.46574223646894097328186035156250.
A := 1.46267583011649549007415771484375.
B := (A + C) / 2 = 1.46420903329271823167800903320312500.
epsilon = difference(x , B) =
0.000870508326514162367876226156226948660332709550857543945312500.

i := 11.
C := 1.46574223646894097328186035156250.
A := 1.46420903329271823167800903320312500.
B := (A + C) / 2 = 1.46497563488082960247993469238281250.

epsilon = difference(x , B) =
0.004062643621267717723016810182912195159588009119033813476562 5.

i := 12.
C := 1.464975634880829602479934692382812 5.
A := 1.464209033292718231678009033203125.
B := (A + C) / 2 = 1.4645923340867739170789718627929687 5.
epsilon = difference(x , B) =
0.0015954221158621017997297775181664292176719754934310913085937 5.

i := 13.
C := 1.46459233408677391707897186279296875.
A := 1.464209033292718231678009033203125.
B := (A + C) / 2 = 1.464400683689746074378490447998046875.
epsilon = difference(x , B) =
0.000362295532913186127399463032361381920054554939270019531 25.

i := 14.
C := 1.464400683689746074378490447998046875.
A := 1.464209033292718231678009033203125.
B := (A + C) / 2 = 1.4643048584912321530282497406005859375.
epsilon = difference(x , B) =
0.000254146734600948663237995894803589180810377001762390136718 75.

i := 15.
C := 1.464400683689746074378490447998046875.
A := 1.4643048584912321530282497406005859375.
B := (A + C) / 2 = 1.46435277109048911370337009429931640625.
epsilon = difference(x , B) =
5.4064314376036856635288074812706327065825462341308593 75e-05.

i := 16.
C := 1.46435277109048911370337009429931640625.
A := 1.4643048584912321530282497406005859375.
B := (A + C) / 2 = 1.464328814790860633365809917449951171875.
epsilon = difference(x , B) =
0.0001000437312662306923310984174690929648932069540023803710937 5.

i := 17.
C := 1.46435277109048911370337009429931640625.
A := 1.464328814790860633365809917449951171875.
B := (A + C) / 2 = 1.4643407929406748735345900058746337890625.
epsilon = difference(x , B) =
2.299033873869645722143317811969609465450048446655273437 5e-05.

i := 18.
C := 1.464352771090489113703370094299316406250.
A := 1.4643407929406748735345900058746337890625.
B := (A + C) / 2 = 1.46434678201558199361898005008697509765625.
epsilon = difference(x , B) =
1.5536830244626109037708872051553044002503156661987304 6875e-05.

i := 19.
C := 1.46434678201558199361898005008697509765625.
A := 1.4643407929406748735345900058746337890625.
B := (A + C) / 2 = 1.464343787478128433576785027980804443359375.
epsilon = difference(x , B) =
3.726793640465532117533875577919388888403773307800292968 75e-06.

i := 20.
C := 1.46434678201558199361898005008697509765625.
A := 1.464343787478128433576785027980804443359375.
B := (A + C) / 2 = 1.4643452847468552135978825390338897705078125.
epsilon = difference(x , B) =
5.905008453712399033030955308731790864840149879455566406 25e-06.

i := 21.
C := 1.4643452847468552135978825390338897705078125.
A := 1.464343787478128433576785027980804443359375.
B := (A + C) / 2 = 1.46434453611249182358733378350734710693359375.
epsilon = difference(x , B) =
1.089104944532572408211201775429799454286694526672363281 25e-06.

i := 22.
C := 1.46434453611249182358733378350734710693359375.
A := 1.464343787478128433576785027980804443359375.
B := (A + C) / 2 = 1.464344161795310128582059405744075775146484375.
epsilon = difference(x , B) =
1.318844963489086696828422873295494355261325836181640625e-06.

i := 23.
C := 1.46434453611249182358733378350734710693359375.
A := 1.464344161795310128582059405744075775146484375.
B := (A + C) / 2 = 1.4643443489539009760846965946257114410400390625.
epsilon = difference(x , B) =
1.148701633587462245245092162804212421178817749023437 5e-07.

i := 24.

C := 1.46434453611249182358733378350734710693359375.
A := 1.4643443489539009760846965946257114410400390625.
B := (A + C) / 2 = 1.46434444253319639983601518906652927398681640625.
epsilon = difference(x , B) =
4.8711735211692634706093230079204658977687358856201171875e-07.

i := 25.
C := 1.46434444253319639983601518906652927398681640625.
A := 1.4643443489539009760846965946257114410400390625.
B := (A + C) / 2 = 1.464344395743548687960355891846120357513427734375.
epsilon = difference(x , B) =
1.8612358476167469023554446039270260371267795562744140625e-07.

i := 26.
C := 1.464344395743548687960355891846120357513427734375.
A := 1.4643443489539009760846965946257114410400390625.
B := (A + C) / 2 = 1.4643443723487248320225262432359158992767333984375.
epsilon = difference(x , B) =
3.562670829702907493441443875781260430812835693359375e-08.

i := 27.
C := 1.4643443723487248320225262432359158992767333984375.
A := 1.4643443489539009760846965946257114410400390625.
B := (A + C) / 2 = 1.46434436065131290405361141893081367015838623046875.
epsilon = difference(x , B) =
3.9621728131940259221011046975036151707172393798828125e-08.

i := 28.
C := 1.4643443723487248320225262432359158992767333984375.
A := 1.46434436065131290405361141893081367015838623046875.
B := (A + C) / 2 = 1.464344366500018868038068831083364784717559814453125.
epsilon = difference(x , B) =
1.997510067942853684286319548846222460269927978515625e-09.

S = approximate_cube_root(3.1400001049041748046875) =
1.464344366500018868038068831083364784717559814453125.

The value which was entered for x is 2.

Computing the approximate cube root of x:

C = 2. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 2.
A := 0.
B := (A + C) / 2 = 1.
epsilon = difference(x , B) = 1.

i := 1.
C := 2.
A := 1.
B := (A + C) / 2 = 1.5.
epsilon = difference(x , B) = 1.375.

i := 2.
C := 1.5.
A := 1.
B := (A + C) / 2 = 1.25.
epsilon = difference(x , B) = 0.046875.

i := 3.
C := 1.5.
A := 1.25.
B := (A + C) / 2 = 1.375.
epsilon = difference(x , B) = 0.599609375.

i := 4.
C := 1.375.
A := 1.25.
B := (A + C) / 2 = 1.3125.
epsilon = difference(x , B) = 0.260986328125.

i := 5.
C := 1.3125.
A := 1.25.
B := (A + C) / 2 = 1.28125.

epsilon = difference(x , B) = 0.103302001953125.

i := 6.
C := 1.28125.
A := 1.25.
B := (A + C) / 2 = 1.265625.
epsilon = difference(x , B) = 0.027286529541015625.

i := 7.
C := 1.265625.
A := 1.25.
B := (A + C) / 2 = 1.2578125.
epsilon = difference(x , B) = 0.010024547576904296875.

i := 8.
C := 1.265625.
A := 1.2578125.
B := (A + C) / 2 = 1.26171875.
epsilon = difference(x , B) = 0.008573234081268310546875.

i := 9.
C := 1.26171875.
A := 1.2578125.
B := (A + C) / 2 = 1.259765625.
epsilon = difference(x , B) = 0.000740073621273040771484375.

i := 10.
C := 1.26171875.
A := 1.259765625.
B := (A + C) / 2 = 1.2607421875.
epsilon = difference(x , B) = 0.003912973217666149139404296875.

i := 11.
C := 1.2607421875.
A := 1.259765625.
B := (A + C) / 2 = 1.26025390625.
epsilon = difference(x , B) = 0.001585548394359648227691650390625.

i := 12.
C := 1.26025390625.
A := 1.259765625.
B := (A + C) / 2 = 1.260009765625.
epsilon = difference(x , B) = 0.000422512079239822924137115478515625.

i := 13.
C := 1.260009765625.
A := 1.259765625.
B := (A + C) / 2 = 1.2598876953125.
epsilon = difference(x , B) = 0.000158837092385510914027690887451171875.

i := 14.
C := 1.260009765625.
A := 1.2598876953125.
B := (A + C) / 2 = 1.25994873046875.
epsilon = difference(x , B) = 0.000131823412402809481136500835418701171875.

i := 15.
C := 1.25994873046875.
A := 1.2598876953125.
B := (A + C) / 2 = 1.259918212890625.
epsilon = difference(x , B) = 1.35103601621722191339358687400817871093750e-05.

i := 16.
C := 1.25994873046875.
A := 1.259918212890625.
B := (A + C) / 2 = 1.2599334716796875.
epsilon = difference(x , B) = 5.91556460669551142927957698702812194824218750e-05.

i := 17.
C := 1.2599334716796875.
A := 1.259918212890625.
B := (A + C) / 2 = 1.25992584228515625.
epsilon = difference(x , B) = 2.2822422940382836031858460046350955963134765625e-05.

i := 18.
C := 1.25992584228515625.
A := 1.259918212890625.
B := (A + C) / 2 = 1.259922027587890625.
epsilon = difference(x , B) = 4.655976386269689015762196504510939121246337890625e-06.

i := 19.
C := 1.259922027587890625.
A := 1.259918212890625.
B := (A + C) / 2 = 1.2599201202392578125.
epsilon = difference(x , B) =
4.4272056386393532356748892198083922266960144042968750e-06.

i := 20.

C := 1.259922027587890625.
A := 1.2599201202392578125.
B := (A + C) / 2 = 1.25992107391357421875.
epsilon = difference(x , B) =
1.143819361405437606826751562039135023951530456542968 75e-07.

i := 21.
C := 1.25992107391357421875.
A := 1.2599201202392578125.
B := (A + C) / 2 = 1.259920597076416015625.
epsilon = difference(x , B) =
2.156412710667735509184606002008877112530171871185302734375e-06.

i := 22.
C := 1.25992107391357421875.
A := 1.259920597076416015625.
B := (A + C) / 2 = 1.2599208354949951171875.
epsilon = difference(x , B) =
1.021015602118259882336026578286691801622509956359863 28125e-06.

i := 23.
C := 1.25992107391357421875.
A := 1.2599208354949951171875.
B := (A + C) / 2 = 1.25992095470428466796875.
epsilon = difference(x , B) =
4.533168867025105103207849310820165555924177169799804 6875e-07.

i := 24.
C := 1.25992107391357421875.
A := 1.25992095470428466796875.
B := (A + C) / 2 = 1.259921014308929443359375.
epsilon = difference(x , B) =
1.694674887093693821382700548383581917732954025268554 6875e-07.

i := 25.
C := 1.25992107391357421875.
A := 1.259921014308929443359375.
B := (A + C) / 2 = 1.2599210441112518310546875.
epsilon = difference(x , B) =
2.754277964153641761191337877789919730275869369506835 9375e-08.

i := 26.
C := 1.25992107391357421875.
A := 1.2599210441112518310546875.

B := (A + C) / 2 = 1.25992105901241302490234375.
epsilon = difference(x , B) =
4.341957741027697992297618156953831203281879425048828125e-08.

i := 27.
C := 1.25992105901241302490234375.
A := 1.2599210441112518310546875.
B := (A + C) / 2 = 1.259921051561832427978515625.
epsilon = difference(x , B) = 7.93839867452295067096201819367706775665283203125e-09.

S = approximate_cube_root(2) = 1.259921051561832427978515625.

The value which was entered for x is -10.

Computing the approximate cube root of x:

C = 10. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 10.
A := 0.
B := (A + C) / 2 = 5.
epsilon = difference(x , B) = 115.

i := 1.
C := 5.
A := 0.
B := (A + C) / 2 = 2.5.
epsilon = difference(x , B) = 5.625.

i := 2.
C := 2.5.
A := 0.

B := (A + C) / 2 = 1.25.
epsilon = difference(x , B) = 8.046875.

i := 3.
C := 2.5.
A := 1.25.
B := (A + C) / 2 = 1.875.
epsilon = difference(x , B) = 3.408203125.

i := 4.
C := 2.5.
A := 1.875.
B := (A + C) / 2 = 2.1875.
epsilon = difference(x , B) = 0.467529296875.

i := 5.
C := 2.1875.
A := 1.875.
B := (A + C) / 2 = 2.03125.
epsilon = difference(x , B) = 1.619110107421875.

i := 6.
C := 2.1875.
A := 2.03125.
B := (A + C) / 2 = 2.109375.
epsilon = difference(x , B) = 0.614414215087890625.

i := 7.
C := 2.1875.
A := 2.109375.
B := (A + C) / 2 = 2.1484375.
epsilon = difference(x , B) = 0.0832772254943847656 25.

i := 8.
C := 2.1875.
A := 2.1484375.
B := (A + C) / 2 = 2.16796875.
epsilon = difference(x , B) = 0.189644992351531982421875.

i := 9.
C := 2.16796875.
A := 2.1484375.
B := (A + C) / 2 = 2.158203125.
epsilon = difference(x , B) = 0.0525664165616035461425781 25.

i := 10.
C := 2.158203125.
A := 2.1484375.
B := (A + C) / 2 = 2.1533203125.
epsilon = difference(x , B) = 0.015509421937167644500732421875.

i := 11.
C := 2.158203125.
A := 2.1533203125.
B := (A + C) / 2 = 2.15576171875.
epsilon = difference(x , B) = 0.018489949288778007030487060546875.

i := 12.
C := 2.15576171875.
A := 2.1533203125.
B := (A + C) / 2 = 2.154541015625.
epsilon = difference(x , B) = 0.001480632126913405954837799072265625.

i := 13.
C := 2.154541015625.
A := 2.1533203125.
B := (A + C) / 2 = 2.1539306640625.
epsilon = difference(x , B) = 0.007016802110229036770761013031005859375.

i := 14.
C := 2.154541015625.
A := 2.1539306640625.
B := (A + C) / 2 = 2.15423583984375.
epsilon = difference(x , B) = 0.002768686878198423073627054691314697265625.

i := 15.
C := 2.154541015625.
A := 2.15423583984375.
B := (A + C) / 2 = 2.154388427734375.
epsilon = difference(x , B) = 0.000644177857935801512212492525577545166015625.

i := 16.
C := 2.154541015625.
A := 2.154388427734375.
B := (A + C) / 2 = 2.1544647216796875.
epsilon = difference(x , B) = 0.000418189512583211353557999245822429656982421875.

i := 17.

C := 2.1544647216796875.
A := 2.154388427734375.
B := (A + C) / 2 = 2.15442657470703125.
epsilon = difference(x , B) = 0.000113003577986159342572136665694415569305419921875.

i := 18.
C := 2.1544647216796875.
A := 2.15442657470703125.
B := (A + C) / 2 = 2.154445648193359375.
epsilon = difference(x , B) =
0.0001525906159502432579699870984768494963645935058 59375.

i := 19.
C := 2.154445648193359375.
A := 2.15442657470703125.
B := (A + C) / 2 = 2.1544361114501953125.
epsilon = difference(x , B) =
1.9792931147573356032154379136045463383197784423828125e-05.

i := 20.
C := 2.1544361114501953125.
A := 2.15442657470703125.
B := (A + C) / 2 = 2.15443134307861328125.
epsilon = difference(x , B) =
4.6605470377584883034938201262775692157447338104248046875e-05.

i := 21.
C := 2.1544361114501953125.
A := 2.15443134307861328125.
B := (A + C) / 2 = 2.154433727264404296875.
epsilon = difference(x , B) =
1.34063063546192851038796334250946529209613800048828125e-05.

i := 22.
C := 2.1544361114501953125.
A := 2.154433727264404296875.
B := (A + C) / 2 = 2.1544349193572998046875.
epsilon = difference(x , B) =
3.1933032115681256324357661924295825883746147155761 71875e-06.

i := 23.
C := 2.1544349193572998046875.
A := 2.154433727264404296875.
B := (A + C) / 2 = 2.15443432331085205078125.

epsilon = difference(x , B) =
5.106503867751722991474849777659983374178409576416015625e-06.

i := 24.
C := 2.1544349193572998046875.
A := 2.15443432331085205078125.
B := (A + C) / 2 = 2.154434621334075927734375.
epsilon = difference(x , B) =
9.56600902148226073240522282503661699593067169189453125e-07.

i := 25.
C := 2.1544349193572998046875.
A := 2.154434621334075927734375.
B := (A + C) / 2 = 2.1544347703456878662109375.
epsilon = difference(x , B) =
1.118351011196276612036371034264448098838329315185546875e-06.

i := 26.
C := 2.1544347703456878662109375.
A := 2.154434621334075927734375.
B := (A + C) / 2 = 2.15443469583988189697265625.
epsilon = difference(x , B) =
8.0875018644739615769623242158559150993824005126953125e-08.

i := 27.
C := 2.15443469583988189697265625.
A := 2.154434621334075927734375.
B := (A + C) / 2 = 2.154434658586978912353515625.
epsilon = difference(x , B) =
4.37862950721130961273530601829406805336475372314453125e-07.

i := 28.
C := 2.15443469583988189697265625.
A := 2.154434658586978912353515625.
B := (A + C) / 2 = 2.1544346772134304046630859375.
epsilon = difference(x , B) =
1.78493968280325765451976849362836219370365142822265625e-07.

i := 29.
C := 2.15443469583988189697265625.
A := 2.1544346772134304046630859375.
B := (A + C) / 2 = 2.15443468652665615081787109375.
epsilon = difference(x , B) = 4.88094753781087575816854950971901416778564453125e-08.

i := 30.
C := 2.15443469583988189697265625.
A := 2.15443468652665615081787109375.
B := (A + C) / 2 = 2.154434691183269023895263671875.
epsilon = difference(x , B) =
1.6032771493236508408841700656921602785587310791015625e-08.

i := 31.
C := 2.154434691183269023895263671875.
A := 2.15443468652665615081787109375.
B := (A + C) / 2 = 2.1544346888549625873565673828125.
epsilon = difference(x , B) =
1.63883519775642749749522408819757401943206787109375e-08.

i := 32.
C := 2.154434691183269023895263671875.
A := 2.1544346888549625873565673828125.
B := (A + C) / 2 = 2.15443469001911580562591552734375.
epsilon = difference(x , B) = 1.777902512711815319335073581896722316741943359375e-10.

S = approximate_cube_root(-10) = -2.15443469001911580562591552734375.

The value which was entered for x is -1.

Computing the approximate cube root of x:

C = 1. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 1.
A := 0.
B := (A + C) / 2 = 0.5.
epsilon = difference(x , B) = 0.875.

i := 1.
C := 1.
A := 0.5.
B := (A + C) / 2 = 0.75.
epsilon = difference(x , B) = 0.578125.

i := 2.
C := 1.
A := 0.75.
B := (A + C) / 2 = 0.875.
epsilon = difference(x , B) = 0.330078125.

i := 3.
C := 1.
A := 0.875.
B := (A + C) / 2 = 0.9375.
epsilon = difference(x , B) = 0.176025390625.

i := 4.
C := 1.
A := 0.9375.
B := (A + C) / 2 = 0.96875.
epsilon = difference(x , B) = 0.090850830078125.

i := 5.
C := 1.
A := 0.96875.
B := (A + C) / 2 = 0.984375.
epsilon = difference(x , B) = 0.046146392822265625.

i := 6.
C := 1.
A := 0.984375.
B := (A + C) / 2 = 0.9921875.
epsilon = difference(x , B) = 0.023254871368408203125.

i := 7.
C := 1.
A := 0.9921875.
B := (A + C) / 2 = 0.99609375.
epsilon = difference(x , B) = 0.011673033237457275390625.

i := 8.

C := 1.
A := 0.99609375.
B := (A + C) / 2 = 0.998046875.
epsilon = difference(x , B) = 0.005847938358783721923828125.

i := 9.
C := 1.
A := 0.998046875.
B := (A + C) / 2 = 0.9990234375.
epsilon = difference(x , B) = 0.0029268274083733558654785156 25.

i := 10.
C := 1.
A := 0.9990234375.
B := (A + C) / 2 = 0.99951171875.
epsilon = difference(x , B) = 0.00146412861067801713943481445 3125.

i := 11.
C := 1.
A := 0.99951171875.
B := (A + C) / 2 = 0.999755859375.
epsilon = difference(x , B) = 0.000732243075617589056491851806640625.

i := 12.
C := 1.
A := 0.999755859375.
B := (A + C) / 2 = 0.9998779296875.
epsilon = difference(x , B) = 0.0003661662358354078605771064758300 78125.

i := 13.
C := 1.
A := 0.9998779296875.
B := (A + C) / 2 = 0.99993896484375.
epsilon = difference(x , B) = 0.000183094293106478289701044559478759765625.

i := 14.
C := 1.
A := 0.99993896484375.
B := (A + C) / 2 = 0.999969482421875.
epsilon = difference(x , B) = 9.154994043569786299485713243484497070 3125e-05.

i := 15.
C := 1.
A := 0.999969482421875.

B := (A + C) / 2 = 0.9999847412109375.
epsilon = difference(x , B) = 4.57756686991217520699137821793556213378906250e-05.

i := 16.
C := 1.
A := 0.9999847412109375.
B := (A + C) / 2 = 0.99999237060546875.
epsilon = difference(x , B) = 2.2888008971211348807628382928669452667236328125e-05.

i := 17.
C := 1.
A := 0.99999237060546875.
B := (A + C) / 2 = 0.999996185302734375.
epsilon = difference(x , B) =
1.1444048141184826050675837905146181583404541015625e-05.

i := 18.
C := 1.
A := 0.999996185302734375.
B := (A + C) / 2 = 0.9999980926513671875.
epsilon = difference(x , B) =
5.722034984508017618765052247908897697925567626953125e-06.

i := 19.
C := 1.
A := 0.9999980926513671875.
B := (A + C) / 2 = 0.99999904632568359375.
epsilon = difference(x , B) =
2.8610202220735512042953274658430018462240695953369140625e-06.

i := 20.
C := 1.
A := 0.99999904632568359375.
B := (A + C) / 2 = 0.999999523162841796875.
epsilon = difference(x , B) =
1.430510792488457090521070114164103870280086994171142578125e-06.

i := 21.
C := 1.
A := 0.999999523162841796875.
B := (A + C) / 2 = 0.9999997615814208984375.
epsilon = difference(x , B) = 7.152556677443091757595539093017578125e-07.

i := 22.

C := 1.
A := 0.99999976158142089843 75.
B := (A + C) / 2 = 0.999999880790710449218 75.
epsilon = difference(x , B) = 3.576278260197796043939888477325439453125e-07.

i := 23.
C := 1.
A := 0.99999988079071044921875.
B := (A + C) / 2 = 0.999999940395355224609375.
epsilon = difference(x , B) = 1.788139236680308385984972119331359863281 25e-07.

i := 24.
C := 1.
A := 0.999999940395355224609375.
B := (A + C) / 2 = 0.9999999701976776123046875.
epsilon = difference(x , B) = 8.9406964498550678399624302983283996582031 25e-08.

i := 25.
C := 1.
A := 0.9999999701976776123046875.
B := (A + C) / 2 = 0.99999998509883880615234375.
epsilon = difference(x , B) = 4.47034829154091539749060757458209991455078125e-08.

i := 26.
C := 1.
A := 0.99999998509883880615234375.
B := (A + C) / 2 = 0.999999992549419403076171875.
epsilon = difference(x , B) = 2.235174162423803068122651893645524978637695312 5e-08.

i := 27.
C := 1.
A := 0.999999992549419403076171875.
B := (A + C) / 2 = 0.9999999962747097015380859375.
epsilon = difference(x , B) = 1.117587085375237876405662973411381244659423828125e-08.

i := 28.
C := 1.
A := 0.9999999962747097015380859375.
B := (A + C) / 2 = 0.99999999813735485076904296875.
epsilon = difference(x , B) =
5.5879354372845302378891574335284531116485595703125e-09.

S = approximate_cube_root(-1) = -0.99999999813735485076904296875.

The value which was entered for x is 0.

Computing the approximate cube root of x:

C = 0. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 0.
A := 0.
B := (A + C) / 2 = 0.
epsilon = difference(x , B) = 0.

S = approximate_cube_root(0) = 0.

The value which was entered for x is 0.5.

Computing the approximate cube root of x:

C = 0. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 0.

A := 0.
B := (A + C) / 2 = 0.
epsilon = difference(x , B) = 0.

S = approximate_cube_root(0.5) = 0.

The value which was entered for x is -81.

Computing the approximate cube root of x:

C = 81. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 81.
A := 0.
B := (A + C) / 2 = 40.5.
epsilon = difference(x , B) = 66349.125.

i := 1.
C := 40.5.
A := 0.
B := (A + C) / 2 = 20.25.
epsilon = difference(x , B) = 8222.765625.

i := 2.
C := 20.25.
A := 0.
B := (A + C) / 2 = 10.125.
epsilon = difference(x , B) = 956.970703125.

i := 3.
C := 10.125.
A := 0.

B := (A + C) / 2 = 5.0625.
epsilon = difference(x , B) = 48.746337890625.

i := 4.
C := 5.0625.
A := 0.
B := (A + C) / 2 = 2.53125.
epsilon = difference(x , B) = 64.781707763671875.

i := 5.
C := 5.0625.
A := 2.53125.
B := (A + C) / 2 = 3.796875.
epsilon = difference(x , B) = 26.263263702392578125.

i := 6.
C := 5.0625.
A := 3.796875.
B := (A + C) / 2 = 4.4296875.
epsilon = difference(x , B) = 5.919909954071044921875.

i := 7.
C := 4.4296875.
A := 3.796875.
B := (A + C) / 2 = 4.11328125.
epsilon = difference(x , B) = 11.407054603099822998046875.

i := 8.
C := 4.4296875.
A := 4.11328125.
B := (A + C) / 2 = 4.271484375.
epsilon = difference(x , B) = 3.0642953887581825256347656 25.

i := 9.
C := 4.4296875.
A := 4.271484375.
B := (A + C) / 2 = 4.3505859375.
epsilon = difference(x , B) = 1.3461416875943541526794433 59375.

i := 10.
C := 4.3505859375.
A := 4.271484375.
B := (A + C) / 2 = 4.31103515625.
epsilon = difference(x , B) = 0.879307645722292363643646240234375.

i := 11.
C := 4.3505859375.
A := 4.31103515625.
B := (A + C) / 2 = 4.330810546875.
epsilon = difference(x , B) = 0.228336121697793714702129364013671875.

i := 12.
C := 4.330810546875.
A := 4.31103515625.
B := (A + C) / 2 = 4.3209228515625.
epsilon = difference(x , B) = 0.326753086765165789984166622161865234375.

i := 13.
C := 4.330810546875.
A := 4.3209228515625.
B := (A + C) / 2 = 4.32586669921875.
epsilon = difference(x , B) = 0.04952567622899550769943743944168090820312 5.

i := 14.
C := 4.330810546875.
A := 4.32586669921875.
B := (A + C) / 2 = 4.328338623046875.
epsilon = difference(x , B) = 0.0893258789971866917767329141497611999511718 75.

i := 15.
C := 4.328338623046875.
A := 4.32586669921875.
B := (A + C) / 2 = 4.3271026611328125.
epsilon = difference(x , B) = 0.01988027111396561963374551851302385330200195 3125.

i := 16.
C := 4.3271026611328125.
A := 4.32586669921875.
B := (A + C) / 2 = 4.32648468017578125.
epsilon = difference(x , B) = 0.01482765941702579581829013477545231580734252 9296875.

i := 17.
C := 4.3271026611328125.
A := 4.32648468017578125.
B := (A + C) / 2 = 4.326793670654296875.
epsilon = difference(x , B) =
0.002525066545089493796893975741113536059856414794921875.

i := 18.
C := 4.326793670654296875.
A := 4.32648468017578125.
B := (A + C) / 2 = 4.3266391754150390625.
epsilon = difference(x , B) =
0.006151606250750417392847424480351037345826625823974609375.

i := 19.
C := 4.326793670654296875.
A := 4.3266391754150390625.
B := (A + C) / 2 = 4.32671642303466796875.
epsilon = difference(x , B) =
0.0018133473079088857637941600842168554663658142089843 75.

i := 20.
C := 4.326793670654296875.
A := 4.32671642303466796875.
B := (A + C) / 2 = 4.326755046844482421875.
epsilon = difference(x , B) =
0.00035584025464784063697010196847259066998958587646484375.

i := 21.
C := 4.326755046844482421875.
A := 4.32671642303466796875.
B := (A + C) / 2 = 4.3267357349395751953125.
epsilon = difference(x , B) =
0.00072875836759452722324326146008388604968786239624023437 5.

i := 22.
C := 4.326755046844482421875.
A := 4.3267357349395751953125.
B := (A + C) / 2 = 4.32674539089202880859375.
epsilon = difference(x , B) =
0.000186460266717043687823007758197491057217121124267578125.

i := 23.
C := 4.326755046844482421875.
A := 4.32674539089202880859375.
B := (A + C) / 2 = 4.326750218868255615234375.
epsilon = difference(x , B) =
8.4689691404134237462386636252631433308124542236328125e-05.

i := 24.
C := 4.326750218868255615234375.

A := 4.326745390892028808593 75.
B := (A + C) / 2 = 4.32674780488014221191406 25.
epsilon = difference(x , B) =
5.08853632967265490094632696127519011497497558593 75e-05.

i := 25.
C := 4.326750218868255615234375.
A := 4.3267478048801422119140625.
B := (A + C) / 2 = 4.32674901187419891357421875.
epsilon = difference(x , B) =
1.69021451436324188222215525456704199314117431640625e-05.

i := 26.
C := 4.32674901187419891357421875.
A := 4.3267478048801422119140625.
B := (A + C) / 2 = 4.326748408377170562744140625.
epsilon = difference(x , B) =
1.699161380406405408294290282356087118387222290039062 5e-05.

i := 27.
C := 4.32674901187419891357421875.
A := 4.326748408377170562744140625.
B := (A + C) / 2 = 4.3267487101256847381591796875.
epsilon = difference(x , B) = 4.4735512100269048119116632733494043350219726562 5e-08.

i := 28.
C := 4.32674901187419891357421875.
A := 4.3267487101256847381591796875.
B := (A + C) / 2 = 4.32674886099994182586669921875.
epsilon = difference(x , B) =
8.42870452029409467087361917947418987751007080078125e-06.

i := 29.
C := 4.32674886099994182586669921875.
A := 4.3267487101256847381591796875.
B := (A + C) / 2 = 4.326748785562813282012939453125.
epsilon = difference(x , B) =
4.1919844302254483103808979649329558014869689941406 25e-06.

i := 30.
C := 4.326748785562813282012939453125.
A := 4.3267487101256847381591796875.
B := (A + C) / 2 = 4.3267487478442490100860595703125.
epsilon = difference(x , B) = 2.0736244406016623997857095673680305480957031 25e-06.

i := 31.
C := 4.3267487478442490100860595703125.
A := 4.326748710125684738159179 6875.
B := (A + C) / 2 = 4.32674872898496687412261962890625.
epsilon = difference(x , B) =
1.0144444596363322297349895961815491318702697753906 25e-06.

i := 32.
C := 4.32674872898496687412261962890625.
A := 4.3267487101256847381591796875.
B := (A + C) / 2 = 4.326748719555325806140899658203125.
epsilon = difference(x , B) =
4.848544726127057558073829568456858396530151367187 5e-07.

i := 33.
C := 4.326748719555325806140899658203125.
A := 4.3267487101256847381591796875.
B := (A + C) / 2 = 4.3267487148405052721500396728515625.
epsilon = difference(x , B) = 2.200594799717237037839367985725402832031 25e-07.

i := 34.
C := 4.3267487148405052721500396728515625.
A := 4.3267487101256847381591796875.
B := (A + C) / 2 = 4.32674871248309500515460968017578125.
epsilon = difference(x , B) = 8.7661983866338388793337799143046140670776367187 5e-08.

i := 35.
C := 4.32674871248309500515460968017578125.
A := 4.3267487101256847381591796875.
B := (A + C) / 2 = 4.326748711304389871656894683837890625.
epsilon = difference(x , B) =
2.146323586221798862538889807183295488357543945312 5e-08.

i := 36.
C := 4.326748711304389871656894683837890625.
A := 4.3267487101256847381591796875.
B := (A + C) / 2 = 4.3267487107150373049080371856689453125.
epsilon = difference(x , B) =
1.163613811555608279491025314200669527053833007812 5e-08.

i := 37.
C := 4.326748711304389871656894683837890625.
A := 4.3267487107150373049080371856689453125.

B := (A + C) / 2 = 4.326748711009713588282465934753417968750.
epsilon = difference(x , B) = 4.913548873330952915239322464913129806518554687 5e-09.

S = approximate_cube_root(-81) = -4.3267487110097135882824659347534179687 5.

The value which was entered for x is 27.

Computing the approximate cube root of x:

C = 27. // real number to take the cube root of
B = 0. // variable for storing the approximate cube root of x
A = 0. // number to add to C before dividing the sum of A and C by 2 for each while loop iteration, i
epsilon = 0. // variable for storing the difference between the input value and B raised to the power of 3

i := 0.
C := 27.
A := 0.
B := (A + C) / 2 = 13.5.
epsilon = difference(x , B) = 2433.375.

i := 1.
C := 13.5.
A := 0.
B := (A + C) / 2 = 6.75.
epsilon = difference(x , B) = 280.546875.

i := 2.
C := 6.75.
A := 0.
B := (A + C) / 2 = 3.375.
epsilon = difference(x , B) = 11.443359375.

i := 3.
C := 3.375.
A := 0.
B := (A + C) / 2 = 1.6875.

epsilon = difference(x , B) = 22.194580078125.

i := 4.
C := 3.375.
A := 1.6875.
B := (A + C) / 2 = 2.53125.
epsilon = difference(x , B) = 10.781707763671875.

i := 5.
C := 3.375.
A := 2.53125.
B := (A + C) / 2 = 2.953125.
epsilon = difference(x , B) = 1.245952606201171875.

i := 6.
C := 3.375.
A := 2.953125.
B := (A + C) / 2 = 3.1640625.
epsilon = difference(x , B) = 4.676352024078369140625.

i := 7.
C := 3.1640625.
A := 2.953125.
B := (A + C) / 2 = 3.05859375.
epsilon = difference(x , B) = 1.6131314635276794433359375.

i := 8.
C := 3.05859375.
A := 2.953125.
B := (A + C) / 2 = 3.005859375.
epsilon = difference(x , B) = 0.15851231664419174194335937 5.

i := 9.
C := 3.005859375.
A := 2.953125.
B := (A + C) / 2 = 2.9794921875.
epsilon = difference(x , B) = 0.549934429116547107696533203125.

i := 10.
C := 3.005859375.
A := 2.9794921875.
B := (A + C) / 2 = 2.99267578125.
epsilon = difference(x , B) = 0.197271501529030501842498779296875.

i := 11.
C := 3.005859375.
A := 2.99267578125.
B := (A + C) / 2 = 2.999267578125.
epsilon = difference(x , B) = 0.019770563041674904525279998779296875.

i := 12.
C := 3.005859375.
A := 2.999267578125.
B := (A + C) / 2 = 3.0025634765625.
epsilon = difference(x , B) = 0.069273026741939247585833072662353515625.

i := 13.
C := 3.0025634765625.
A := 2.999267578125.
B := (A + C) / 2 = 3.00091552734375.
epsilon = difference(x , B) = 0.0247267827614905399968847632408142089843 75.

i := 14.
C := 3.00091552734375.
A := 2.999267578125.
B := (A + C) / 2 = 3.000091552734375.
epsilon = difference(x , B) = 0.002471999266020930008380673825740814208984375.

i := 15.
C := 3.000091552734375.
A := 2.999267578125.
B := (A + C) / 2 = 2.9996795654296875.
epsilon = difference(x , B) = 0.008650809326514519170814310200512409210205078125.

i := 16.
C := 3.000091552734375.
A := 2.9996795654296875.
B := (A + C) / 2 = 2.99988555908203125.
epsilon = difference(x , B) = 0.0030897869161417013117443275405094027519226074218 75.

i := 17.
C := 3.000091552734375.
A := 2.99988555908203125.
B := (A + C) / 2 = 2.999988555908203125.
epsilon = difference(x , B) =
0.00030898929981199030336824762343894690275192260742 1875.

i := 18.

C := 3.000091552734375.
A := 2.999988555908203125.
B := (A + C) / 2 = 3.0000400543212890625.
epsilon = difference(x , B) =
0.001081481114006833943452789981165551580488681793212890625.

i := 19.
C := 3.0000400543212890625.
A := 2.999988555908203125.
B := (A + C) / 2 = 3.00001430511474609375.
epsilon = difference(x , B) =
0.00038623993987423055340713062832946889102458953857421875.

i := 20.
C := 3.00001430511474609375.
A := 2.999988555908203125.
B := (A + C) / 2 = 3.000001430511474609375.
epsilon = difference(x , B) =
3.86238282317243053487487713937298394739627838134765625e-05.

i := 21.
C := 3.000001430511474609375.
A := 2.999988555908203125.
B := (A + C) / 2 = 2.9999949932098388671875.
epsilon = difference(x , B) =
0.00013518310873918137904325931231142021715641021728515625.

i := 22.
C := 3.000001430511474609375.
A := 2.9999949932098388671875.
B := (A + C) / 2 = 2.99999821186065673828125.
epsilon = difference(x , B) =
4.827973349109081213637040264075039885938167572021484375e-05.

i := 23.
C := 3.000001430511474609375.
A := 2.99999821186065673828125.
B := (A + C) / 2 = 2.999999821186065673828125.
epsilon = difference(x , B) = 4.827975939036832642159424722194671630859375e-06.

i := 24.
C := 3.000001430511474609375.
A := 2.999999821186065673828125.
B := (A + C) / 2 = 3.0000006258487701416015625.

epsilon = difference(x , B) = 1.689792031900338997729704715311527252197265625e-05.

i := 25.
C := 3.0000006258487701416015625.
A := 2.999999821186065673828125.
B := (A + C) / 2 = 3.00000022351741790771484375.
epsilon = difference(x , B) = 6.0349707331486257544383988715708255767822265625e-06.

i := 26.
C := 3.00000022351741790771484375.
A := 2.999999821186065673828125.
B := (A + C) / 2 = 3.000000022351741790771484375.
epsilon = difference(x , B) = 6.03497032847233327856883988715708255767822265625e-07.

i := 27.
C := 3.000000022351741790771484375.
A := 2.999999821186065673828125.
B := (A + C) / 2 = 2.9999999217689037322998046875.
epsilon = difference(x , B) =
2.11223954414696546422192113823257386684417724609375e-06.

i := 28.
C := 3.000000022351741790771484375.
A := 2.9999999217689037322998046875.
B := (A + C) / 2 = 2.99999997206032276153564453125.
epsilon = difference(x , B) =
7.543712784129075199501812676317058503627777099609375e-07.

i := 29.
C := 3.000000022351741790771484375.
A := 2.99999997206032276153564453125.
B := (A + C) / 2 = 2.999999997206032276153564453125.
epsilon = difference(x , B) = 7.543712847446482072655271622352302074432373046875e-08.

i := 30.
C := 3.000000022351741790771484375.
A := 2.999999997206032276153564453125.
B := (A + C) / 2 = 3.0000000097788870334625244140625.
epsilon = difference(x , B) = 2.640299507639110032641838188283145427703857421875e-07.

i := 31.
C := 3.0000000097788870334625244140625.
A := 2.999999997206032276153564453125.
B := (A + C) / 2 = 3.00000000349245965480804443359375.

epsilon = difference(x , B) =
9.429641078910477869357009694795124232769012451171875e-08.

i := 32.
C := 3.000000000349245965480804443359375.
A := 2.9999999997206032276153564453125.
B := (A + C) / 2 = 3.000000000349245965480804443359375.
epsilon = difference(x , B) = 9.429641067981719970703125e-09.

S = approximate_cube_root(27) = 3.000000000349245965480804443359375.

End Of Program

This web page was last updated on 17_OCTOBER_2023. The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

[End of abridged plain-text content from CUBE_ROOT_APPROXIMATION]

EULERS_NUMBER_APPROXIMATION

The C++ program featured in this tutorial web page generates an approximation of the the mathematical constant named e (i.e. Euler's Number). The program user is prompted to enter a nonnegative integer to store in a variable named N. Then the approximate value of e is computed by adding N unique terms (and each of those N terms is 1 divided by factorial i (and i is a nonnegative integer which is smaller than or equal to (N – 1))). Note that the actual value of e is a limit which cannot be determined using a finite number of additions as described in the previous sentence. Instead, the actual value of e is defined as the sum of every unique instance of (1/(i!)) such that i is a nonnegative integer.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

Let E be 0.
Let N be a nonnegative integer.
For each nonnegative integer i (starting with 0 and ending with (N - i)):

Add (1/(i!)) to E.
End For.
// E represents the approximate value of Euler's Number.
// As N approaches INFINITY, E approaches Euler's Number.

SOFTWARE_APPLICATION_COMPONENTS

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/eulers_number_approximation.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/eulers_number_approximation_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ source code into a new text editor document and save that document as the following file name:

eulers_number_approximation.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ eulers_number_approximation.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: After running the g++ command, run the executable file using the following command:

./app

STEP_5: Once the application is running, the following prompt will appear:

Enter a nonnegative integer which is no larger than 65:

STEP_6: Enter a value for N using the keyboard.

STEP_7: Observe program results on the command line terminal and in the output file.

PROGRAM_SOURCE_CODE

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/eulers_number_approximation.cpp

---

```
/**
 * file: eulers_number_approximation.cpp
 * type: C++ (source file)
 * date: 21_JUNE_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#include < iostream > // command line user interface input and output
#include < fstream > // file input and output
#define MAXIMUM_N 65 // constant which represents maximum N value

/* function prototypes */
unsigned long long compute_factorial_of_N(int N);
long double approximate_eulers_number(int N, std::ostream & output);

/**
```

```
 * Compute N factorial (N!) using an iterative algorithm.
 *
 * If N is a natural number, then N! is the product of exactly one instance
 * of each unique natural number which is smaller than or equal to N.
 * N! := N * (N - 1) * (N - 2) * (N - 3) * ... * 3 * 2 * 1.
 *
 * If N is zero, then N! is one.
 * 0! := 1.
 *
 * The value returned by this function is a nonnegative integer (which is N factorial).
 *
 * Assume that N is a nonnegative integer no larger than MAXIMUM_N.
 */
unsigned long long compute_factorial_of_N(int N)
{
   // Declare an int type variable (i.e. a variable for storing integer values) named i.
   // Set the initial value which is stored in i to zero.
   int i = 0;

   // Declare an unsigned long long type variable (i.e. a variable for storing nonnegative integer
values) named F.
   // Set the initial value which is stored in F to zero.
   unsigned long long int F = 0;

   // If N is larger than negative one and if N is not larger than MAXIMUM_N, set i to N.
   // Otherwise, set i to 0.
   i = ((N > -1) && (N  0) ? N : 1;

   // While i is larger than zero: execute the code block enclosed by the following curly braces.
   while (i > 0)
   {
         // If i is larger than 1, multiply F by (i - 1).
         if (i > 1) F *= i - 1;

         // Decrement i by 1.
         i -= 1;
   }

   // Return the value stored in F.
   // The value stored in F is factorial N.
   // Factorial N is denoted by N followed by the exclamation mark.
   return F; // Return the value represented by N!
}
```

```
/**
 * Generate an approximation of the the mathematical constant named e (i.e. Euler’s Number).
 *
 * Euler's Number can be defined as follows:
 * Let N be a natural number. Then,
 * as the value of N approaches INFINITY,
 * the value of (1 + (1/N)) ^ N approaches Euler's Number.
 *
 * Euler's Number can also be defined as follows:
 * Let N be a nonnegative integer. Then,
 * as the value of N approaches INFINITY,
 * the sum of each instance of N unique terms such that each term of those N terms is
 * 1 divided by the factorial of some nonnegative integer which is smaller than or equal to N
 * approaches Euler's Number.
 *
 * The value returned by this function is a positive floating-point number (which is the Nth
approximation of Euler's Number).
 *
 * Assume that N is a nonnegative integer no larger than MAXIMUM_N.
 */
long double approximate_eulers_number(int N, std::ostream & output)
{
    // Declare a long double type variable (i.e. a variable for storing floating-point number values)
named A.
    // Set the initial value which is stored in A to zero.
    long double A = 0.0;

    // Declare an int type variable (i.e. a variable for storing integer values) named i.
    // Set the intiial value which is stored in i to zero.
    int i = 0;

    // Declare a pointer to an unsigned long long type variable named T.
    // T initially stores the value 0 and will later be used to store the memory address of a
dynamically-allocated array of N unsigned long long type values.
    //
    // A pointer variable stores 0 or else the memory address of a variable of the corresponding
data type.
    // The value stored in a pointer is a sixteen-character memory address which denotes the first
memory cell in a chunk of contiguous memory cells
    // (and that chunk of contiguous memory cells is exactly as large as is the data capacity of a
variable of that pointer's corresponding data type).
    // Note that each memory cell has a data capacity of one byte.
    unsigned long long * T;
```

```
    // If N is smaller than one or if N is larger than MAXIMUM_N, set N to one.
    if ((N  MAXIMUM_N)) N = 1;

    // Dynamically allocate N contiguous unsigned long long sized chunks of memory to an array for storing N floating-point values.
    // Store the memory address of the first element of that array in T.
    T = new unsigned long long [N];

    // Print the previous command and comments associated with that command to the output stream.
    output << "\n\n// Dynamically allocate N contiguous unsigned long long sized chunks of memory to an array for storing N floating-point values.";
    output << "\n// Store the memory address of the first element of that array in T.";
    output << "\nT = new unsigned long long [N];";

    // Print "A := {A}. // variable to store the Nth approximation of Euler's Number" to the output stream.
    output << "\n\nA := " << A << ". // variable to store the Nth approximation of Euler's Number";

    // Print "N := {N}. // number of elements in the array of unsigned long long type values pointed to by T" to the output stream.
    output << "\n\nN := " << N << ". // number of elements in the array of unsigned long long type values pointed to by T";

    // Print "sizeof(unsigned long long) = {sizeof(unsigned long long)}. // number of bytes" to the output stream.
    output << "\n\nsizeof(unsigned long long) = " << sizeof(unsigned long long) << ". // number of bytes";

    // Print "&T = {&T}. // memory address of unsigned long long type variable named T" to the output stream.
    output << "\n\n&T = " << &T << ". // memory address of unsigned long long type variable named T";

    // Print "T := {T}. // memory address which is stored in T" to the output stream.
    output << "\n\nT = " << T << ". // memory address which is stored in T";

    // Print "(*T) := {*T}. // unsigned long long type value whose address is stored in T" to the output stream.
    output << "\n\n(*T) = " << (*T) << ". // unsigned long long type value whose address is stored in T";

    // Print "Displaying the memory address of each element of array T..." to the output stream.
    output << "\n\nDisplaying the memory address of each element of array T...\n";
```

```
    // For each integer value represented by i (starting with 0 and ending with (N - 1) in ascending
order):
    // print the memory address of the ith element of the array represented by T to the output
stream.
    for (i = 0; i < N; i += 1)
    {
        // Print "&T[{i}] = {&T[i]}. // memory address of T[{i}]" to the output stream.
        output << "\n&T[" << i << "] = " << &T[i] << ". // memory address of T[" << i << "]";
    }

    // Print "Storing the factorial of each nonnegative integer which is smaller than N in array T..."
to the output stream.
    output << "\n\nStoring the factorial of each nonnegative integer which is smaller than N in
array T...\n";

    // For each integer value represented by i (starting with 0 and ending with (N - 1) in ascending
order):
    // set the value of the ith element of array T to i! and
    // print the data value which is stored in the ith element of the array to the output stream.
    for (i = 0; i < N; i += 1)
    {
        // Print "T[{i}] := factorial({i}) = {i}! = " to the output stream.
        output << "\nT[" << i << "] := factorial(" << i << ") = (" << i << ")! = ";

        // Store i! in T[i].
        T[i] = compute_factorial_of_N(i);

        // Print "{T[i]}." to the output stream.
        output << T[i] << ".";
    }

    // Print "Computing the appoximate value of Euler's Number by adding up the reciprocal of
each value in array T..." to the output stream.
    output << "\n\nComputing the appoximate value of Euler's Number by adding up the
reciprocal of each value in array T...\n";

    // For each integer value represented by i (starting with 0 and ending with (N - 1) in ascending
order):
    // print the value of (1 / T[i]) to the output stream.
    for (i = 0; i < N; i += 1) output << "\n(1 / T[" << i << "]) = (1 / " << T[i] << ") = " << (long double)
1 / T[i] << ".";
```

```
   // For each integer value represented by i (starting with 0 and ending with (N - 1) in ascending
order):
   // add the value of (1 / (N - i)!) to A and print the data value which is stored in A to the output
stream.
   for (i = 0; i < N; i += 1)
   {
          output << "\n\nA := A + (1 / (" << i << ")!)";
          output << "\n   = " << A << " + (1 / " << T[i] << ")";
          output << "\n   = " << A << " + " << (long double) 1 / T[i];
          A += (long double) 1 / T[i];
          output << "\n   = " << A << ".";
   }

   // De-allocate memory which was dynamically assigned to the array named T.
   delete [] T;

   // Return the value which is stored in A.
   return A;
}

/* program entry point */
int main()
{
   // Declare a file output stream object named file.
   std::ofstream file;

   // Declare an int type variable (i.e. a variable for storing integer values) named N.
   // Set the initial value which is stored in N to one.
   int N = 1;

   // Declare a long double type variable (i.e. a variable for storing floating-point number values)
named B.
   // Set the initial value which is stored in B to one.
   long double B = 1.0;

         // Set the number of digits of floating-point numbers which are printed to the command
line terminal to 100 digits.
         std::cout.precision(100);

         // Set the number of digits of floating-point numbers which are printed to the file output
stream to 100 digits.
         file.precision(100);

   /**
```

```
     * If the file named eulers_number_approximation_output.txt does not already exist
     * inside of the same file directory as the file named eulers_number_approximation.cpp,
     * create a new file named eulers_number_approximation_output.txt in that directory.
     *
     * Open the plain-text file named eulers_number_approximation_output.txt
     * and set that file to be overwritten with program data.
     */
    file.open("eulers_number_approximation_output.txt");

    // Print an opening message to the command line terminal.
    std::cout << "\n\n--------------------------------";
    std::cout << "\nSTART OF PROGRAM";
    std::cout << "\n--------------------------------";

    // Print an opening message to the file output stream.
    file << "--------------------------------";
    file << "\nSTART OF PROGRAM";
    file << "\n--------------------------------";

    // Print "Enter a nonnegative integer which is no larger than {MAXIMUM_N}: " to the
command line terminal.
    std::cout << "\n\nEnter a nonnegative integer which is no larger than " << MAXIMUM_N <> N;

    // Compute the Nth approximation of Euler's Number and store the result in B.
    // Print the steps involved in generating the Nth approximation of Euler's Number to the
command line terminal.
    B = approximate_eulers_number(N, std::cout);

    // Print the steps involved in generating the Nth approximation of Euler's Number to the file
output stream.
    approximate_eulers_number(N, file);

    // Print "B ::= eulers_number_approximation(N) = {B}." to the command line terminal.
    std::cout << "\n\nB := eulers_number_approximation(N) := " << B << ".";

    // Print "B := eulers_number_approximation(N) := {B}." to the file output stream.
    file << "\n\nB = eulers_number_approximation(N) := " << B << ".";

    // Print a closing message to the command line terminal.
    std::cout << "\n\n--------------------------------";
    std::cout << "\nEND OF PROGRAM";
    std::cout << "\n--------------------------------\n\n";

    // Print a closing message to the file output stream.
```

```
    file << "\n\n--------------------------------";
    file << "\nEND OF PROGRAM";
    file << "\n--------------------------------";

    // Close the file output stream.
    file.close();

    // Exit the program.
    return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/eulers_number_approximation_output.txt

---

```
--------------------------------
START OF PROGRAM
--------------------------------

// Dynamically allocate N contiguous unsigned long long sized chunks of memory to an array for storing N floating-point values.
// Store the memory address of the first element of that array in T.
T = new unsigned long long [N];

A := 0. // variable to store the Nth approximation of Euler's Number

N := 65. // number of elements in the array of unsigned long long type values pointed to by T

sizeof(unsigned long long) = 8. // number of bytes

&T = 0x7ffe8d20ee88. // memory address of unsigned long long type variable named T

T = 0x556b195078c0. // memory address which is stored in T

(*T) = 22929315079. // unsigned long long type value whose address is stored in T
```

Displaying the memory address of each element of array T...

&T[0] = 0x556b195078c0. // memory address of T[0]
&T[1] = 0x556b195078c8. // memory address of T[1]
&T[2] = 0x556b195078d0. // memory address of T[2]
&T[3] = 0x556b195078d8. // memory address of T[3]
&T[4] = 0x556b195078e0. // memory address of T[4]
&T[5] = 0x556b195078e8. // memory address of T[5]
&T[6] = 0x556b195078f0. // memory address of T[6]
&T[7] = 0x556b195078f8. // memory address of T[7]
&T[8] = 0x556b19507900. // memory address of T[8]
&T[9] = 0x556b19507908. // memory address of T[9]
&T[10] = 0x556b19507910. // memory address of T[10]
&T[11] = 0x556b19507918. // memory address of T[11]
&T[12] = 0x556b19507920. // memory address of T[12]
&T[13] = 0x556b19507928. // memory address of T[13]
&T[14] = 0x556b19507930. // memory address of T[14]
&T[15] = 0x556b19507938. // memory address of T[15]
&T[16] = 0x556b19507940. // memory address of T[16]
&T[17] = 0x556b19507948. // memory address of T[17]
&T[18] = 0x556b19507950. // memory address of T[18]
&T[19] = 0x556b19507958. // memory address of T[19]
&T[20] = 0x556b19507960. // memory address of T[20]
&T[21] = 0x556b19507968. // memory address of T[21]
&T[22] = 0x556b19507970. // memory address of T[22]
&T[23] = 0x556b19507978. // memory address of T[23]
&T[24] = 0x556b19507980. // memory address of T[24]
&T[25] = 0x556b19507988. // memory address of T[25]
&T[26] = 0x556b19507990. // memory address of T[26]
&T[27] = 0x556b19507998. // memory address of T[27]
&T[28] = 0x556b195079a0. // memory address of T[28]
&T[29] = 0x556b195079a8. // memory address of T[29]
&T[30] = 0x556b195079b0. // memory address of T[30]
&T[31] = 0x556b195079b8. // memory address of T[31]
&T[32] = 0x556b195079c0. // memory address of T[32]
&T[33] = 0x556b195079c8. // memory address of T[33]
&T[34] = 0x556b195079d0. // memory address of T[34]
&T[35] = 0x556b195079d8. // memory address of T[35]
&T[36] = 0x556b195079e0. // memory address of T[36]
&T[37] = 0x556b195079e8. // memory address of T[37]
&T[38] = 0x556b195079f0. // memory address of T[38]
&T[39] = 0x556b195079f8. // memory address of T[39]
&T[40] = 0x556b19507a00. // memory address of T[40]

&T[41] = 0x556b19507a08. // memory address of T[41]
&T[42] = 0x556b19507a10. // memory address of T[42]
&T[43] = 0x556b19507a18. // memory address of T[43]
&T[44] = 0x556b19507a20. // memory address of T[44]
&T[45] = 0x556b19507a28. // memory address of T[45]
&T[46] = 0x556b19507a30. // memory address of T[46]
&T[47] = 0x556b19507a38. // memory address of T[47]
&T[48] = 0x556b19507a40. // memory address of T[48]
&T[49] = 0x556b19507a48. // memory address of T[49]
&T[50] = 0x556b19507a50. // memory address of T[50]
&T[51] = 0x556b19507a58. // memory address of T[51]
&T[52] = 0x556b19507a60. // memory address of T[52]
&T[53] = 0x556b19507a68. // memory address of T[53]
&T[54] = 0x556b19507a70. // memory address of T[54]
&T[55] = 0x556b19507a78. // memory address of T[55]
&T[56] = 0x556b19507a80. // memory address of T[56]
&T[57] = 0x556b19507a88. // memory address of T[57]
&T[58] = 0x556b19507a90. // memory address of T[58]
&T[59] = 0x556b19507a98. // memory address of T[59]
&T[60] = 0x556b19507aa0. // memory address of T[60]
&T[61] = 0x556b19507aa8. // memory address of T[61]
&T[62] = 0x556b19507ab0. // memory address of T[62]
&T[63] = 0x556b19507ab8. // memory address of T[63]
&T[64] = 0x556b19507ac0. // memory address of T[64]

Storing the factorial of each nonnegative integer which is smaller than N in array T...

T[0] := factorial(0) = (0)! = 1.
T[1] := factorial(1) = (1)! = 1.
T[2] := factorial(2) = (2)! = 2.
T[3] := factorial(3) = (3)! = 6.
T[4] := factorial(4) = (4)! = 24.
T[5] := factorial(5) = (5)! = 120.
T[6] := factorial(6) = (6)! = 720.
T[7] := factorial(7) = (7)! = 5040.
T[8] := factorial(8) = (8)! = 40320.
T[9] := factorial(9) = (9)! = 362880.
T[10] := factorial(10) = (10)! = 3628800.
T[11] := factorial(11) = (11)! = 39916800.
T[12] := factorial(12) = (12)! = 479001600.
T[13] := factorial(13) = (13)! = 6227020800.
T[14] := factorial(14) = (14)! = 87178291200.
T[15] := factorial(15) = (15)! = 1307674368000.
T[16] := factorial(16) = (16)! = 20922789888000.

T[17] := factorial(17) = (17)! = 355687428096000.
T[18] := factorial(18) = (18)! = 6402373705728000.
T[19] := factorial(19) = (19)! = 121645100408832000.
T[20] := factorial(20) = (20)! = 2432902008176640000.
T[21] := factorial(21) = (21)! = 14197454024290336768.
T[22] := factorial(22) = (22)! = 17196083355034583040.
T[23] := factorial(23) = (23)! = 8128291617894825984.
T[24] := factorial(24) = (24)! = 10611558092380307456.
T[25] := factorial(25) = (25)! = 7034535277573963776.
T[26] := factorial(26) = (26)! = 16877220553537093632.
T[27] := factorial(27) = (27)! = 12963097176472289280.
T[28] := factorial(28) = (28)! = 12478583540742619136.
T[29] := factorial(29) = (29)! = 11390785281054474240.
T[30] := factorial(30) = (30)! = 9682165104862298112.
T[31] := factorial(31) = (31)! = 4999213071378415616.
T[32] := factorial(32) = (32)! = 12400865694432886784.
T[33] := factorial(33) = (33)! = 3400198294675128320.
T[34] := factorial(34) = (34)! = 4926277576697053184.
T[35] := factorial(35) = (35)! = 6399018521010896896.
T[36] := factorial(36) = (36)! = 9003737871877668864.
T[37] := factorial(37) = (37)! = 1096907932701818880.
T[38] := factorial(38) = (38)! = 4789013295250014208.
T[39] := factorial(39) = (39)! = 2304077777655037952.
T[40] := factorial(40) = (40)! = 18376134811363311616.
T[41] := factorial(41) = (41)! = 15551764317513711616.
T[42] := factorial(42) = (42)! = 7538058755741581312.
T[43] := factorial(43) = (43)! = 10541877243825618944.
T[44] := factorial(44) = (44)! = 2673996885588443136.
T[45] := factorial(45) = (45)! = 9649395409222631424.
T[46] := factorial(46) = (46)! = 1150331055211806720.
T[47] := factorial(47) = (47)! = 17172071447535812608.
T[48] := factorial(48) = (48)! = 12602690238498734080.
T[49] := factorial(49) = (49)! = 8789267254022766592.
T[50] := factorial(50) = (50)! = 15188249005818642432.
T[51] := factorial(51) = (51)! = 18284192274659147776.
T[52] := factorial(52) = (52)! = 9994050523088551936.
T[53] := factorial(53) = (53)! = 13175843659825807360.
T[54] := factorial(54) = (54)! = 10519282829630636032.
T[55] := factorial(55) = (55)! = 6711489344688881664.
T[56] := factorial(56) = (56)! = 6908521828386340864.
T[57] := factorial(57) = (57)! = 6404118670120845312.
T[58] := factorial(58) = (58)! = 2504001392817995776.
T[59] := factorial(59) = (59)! = 162129586585337856.
T[60] := factorial(60) = (60)! = 9727775195120271360.

T[61] := factorial(61) = (61)! = 3098476543630901248.
T[62] := factorial(62) = (62)! = 7638104968020361216.
T[63] := factorial(63) = (63)! = 1585267068834414592.
T[64] := factorial(64) = (64)! = 9223372036854775808.

Computing the appoximate value of Euler's Number by adding up the reciprocal of each value in array T...

(1 / T[0]) = (1 / 1) = 1.
(1 / T[1]) = (1 / 1) = 1.
(1 / T[2]) = (1 / 2) = 0.5.
(1 / T[3]) = (1 / 6) =
0.166666666666666666671184175718689601808364386670291423797607421875.
(1 / T[4]) = (1 / 24) =
0.04166666666666666666779604392967240045209109666757285594940185546875.
(1 / T[5]) = (1 / 120) =
0.0083333333333333333337286153753853401582318838336504995822906494140625.
(1 / T[6]) = (1 / 720) =
0.0013888888888888888888488901108241024839884403263567946851253509521484375.
(1 / T[7]) = (1 / 5040) =
0.0001984126984126984126983706828895211160546097062251647002995014190673828125.
(1 / T[8]) = (1 / 40320) =
2.48015873015873015872963353611901395068262132781455875374376773834228515625e-05.
(1 / T[9]) = (1 / 362880) =
2.7557319223985890651862166557528187500747396398992350441403687000274658203125e-06.
(1 / T[10]) = (1 / 3628800) =
2.755731922398589065134517867468254520395276596644862365792505443096160888671875e-07.
(1 / T[11]) = (1 / 39916800) =
2.505210838544171877468452021966260117517670547027108796100947074592113494873046875e-08.
(1 / T[12]) = (1 / 479001600) =
2.0876756987868098978903766849718834312647254558559239967507892288267612457275390625e-09.
(1 / T[13]) = (1 / 6227020800) =
1.60590438368216145992538343035032278473177931761572967417350810137577354907989501953125e-10.
(1 / T[14]) = (1 / 87178291200) =
1.1470745597729724713978127618279737549630346144478865166860259705572389066219329833984375e-11.

(1 / T[15]) = (1 / 1307674368000) =
7.6471637318198164760182876165707005249790527865393595374765567385111353360116481781005859375e-13.
(1 / T[16]) = (1 / 20922789888000) =
4.7794773323873852975114297603566878281119079915870997109228479615694595850072801113128662109375e-14.
(1 / T[17]) = (1 / 355687428096000) =
2.8114572543455207631627145083821066578811703150624439991912828507025778890238143503665924072265625e-15.
(1 / T[18]) = (1 / 6402373705728000) =
1.561920696858622646281755141228416303811315922642396415600911374621517779814894311130046844482421875e-16.
(1 / T[19]) = (1 / 121645100408832000) =
8.220635246624329716718057091551259700512195415301490541412415477551256515198474517092108726501464844e-18.
(1 / T[20]) = (1 / 2432902008176640000) =
4.110317623312164858406048319808521350574847169139635097818954361046511758459587326797191053628921509e-19.
(1 / T[21]) = (1 / 14197454024290336768) =
7.043516381804132984552581733844016154884793905449790136826614643386981762240850457601482048630714417e-20.
(1 / T[22]) = (1 / 17196083355034583040) =
5.815277696401867087825620308901504154687317329562106344189912444453752216055875123856822028756141663e-20.
(1 / T[23]) = (1 / 8128291617894825984) =
1.230270820744732847600809726906469170638222121705104977548044832558193917293465347029268741607666016e-19.
(1 / T[24]) = (1 / 10611558092380307456) =
9.423686807294170693318948175954948019683466235544601296535251525304105468805460077419411391019821167e-20.
(1 / T[25]) = (1 / 7034535277573963776) =
1.421558014198878427804392419215390985622496694920617818255280764058040565700480328814592212438583374e-19.
(1 / T[26]) = (1 / 16877220553537093632) =
5.925146245662008780449842354395082473873705632309314560039469494409983263416563659120583906769752502e-20.
(1 / T[27]) = (1 / 12963097176472289280) =
7.714205844379351220442307590283867888447153857127854916092693027609983325021403288701549172401428223e-20.
(1 / T[28]) = (1 / 12478583540742619136) =
8.013730057862709208685990201180404102359911354745243625185241551512477231611342176620382815599441528e-20.

(1 / T[29]) = (1 / 11390785281054474240) =
8.779025987464030508121718994662328990235623784252274002513994418002429842573519636061973869800567627e-20.
(1 / T[30]) = (1 / 9682165104862298112) =
1.032826841072776997026950697730699208625361299463986189138260421926072962772735763792297802865505219e-19.
(1 / T[31]) = (1 / 4999213071378415616) =
2.000314820996964391002384152827705545795997986125651431532452634775784416909516494342824444174766541e-19.
(1 / T[32]) = (1 / 12400865694432886784) =
8.063953151665285768186808446574614784476978848083338270190630530992112467991717039694776758551597595e-20.
(1 / T[33]) = (1 / 3400198294675128320) =
2.941004945405823520516414193893468377707800576448082786165399320817831485541660185845103114843368853e-19.
(1 / T[34]) = (1 / 4926277576697053184) =
2.029930275813802546836213263140678056231305513535500619692556199856817156224053633195580914616584778e-19.
(1 / T[35]) = (1 / 6399018521010896896) =
1.562739655646477380892696552554155496761738949953131410708552116381464536232215323252603411674499512e-19.
(1 / T[36]) = (1 / 9003737871877668864) =
1.110649837023139300042549561205190561515953323270939067541196991093996326860349199705524370074272156e-19.
(1 / T[37]) = (1 / 10969079327018188880) =
9.116535400896201500929943441933132123874700032396657688457333604606369625855677440995350480079650879e-19.
(1 / T[38]) = (1 / 4789013295250014208) =
2.088112808105691869924731083569285259541630716029320990911684108098586576396371583541622385382652283e-19.
(1 / T[39]) = (1 / 2304077777655037952) =
4.340131265090123433015478527485500881114515851366072819926298178606904887288919780985452234745025635e-19.
(1 / T[40]) = (1 / 18376134811363311616) =
5.441840791141925413183004478172199152903907339519060119881344722830157634163583679764997214078903198e-20.
(1 / T[41]) = (1 / 15551764317513711616) =
6.430138597675660950353326517785726873973361451550566389280682460562188484942680588574148714542388916e-20.
(1 / T[42]) = (1 / 7538058755741581312) =
1.326601493041323093804705908412158628351599110254046220970487675360274804070570553449215367436408997e-19.

(1 / T[43]) = (1 / 10541877243825618944) =
9.485976518894681088585945984202713644634515871086129452208364070146459634536
029170703841373324394226e-20.
(1 / T[44]) = (1 / 2673996885588443136) =
3.739720137257896377418136692574652015537340387383811719072360135220078891649
109209538437426090240479e-19.
(1 / T[45]) = (1 / 9649395409222631424) =
1.036334358362210981733926214460651797062864929183711526852237064981812619812
728826218517497181892395e-19.
(1 / T[46]) = (1 / 1150331055211806720) =
8.693149641308025443189145265311138141228065069562171980261825821392762669859
166635433211922645568848e-19.
(1 / T[47]) = (1 / 17172071447535812608) =
5.823409266932089994910827473586939922642565582829483048515030398258052191096
112437662668526172637939e-20.
(1 / T[48]) = (1 / 12602690238498734080) =
7.934813766549598658208990425096079332701638958718522115960120841835281901843
757168535375967621803284e-20.
(1 / T[49]) = (1 / 8789267254022766592) =
1.137751272203390128107638181047964849359748082386584368096104198870080481675
870487379143014550209045e-19.
(1 / T[50]) = (1 / 15188249005818642432) =
6.584037433261058626315122747133625100946933977144928122763979037304279962050
657104555284604430198669e-20.
(1 / T[51]) = (1 / 18284192274659147776) =
5.469205229185558363221202926338644452360917504749919928199492245380133881305
084742052713409066200256e-20.
(1 / T[52]) = (1 / 9994050523088551936) =
1.000595301864614693116746035656340128581767350489224182304672495732490722364
360635765478946268558502e-19.
(1 / T[53]) = (1 / 13175843659825807360) =
7.589646825038455307200389676950583077870852261845009400057856808466047460193
237839121138677000999451e-20.
(1 / T[54]) = (1 / 10519282829630636032) =
9.506351489886816144216164308811908919545279361999955678807380202340049368814
334229682572185993194458e-20.
(1 / T[55]) = (1 / 6711489344688881664) =
1.489982250797056252874586363401948231626454775496495728635927216368731174078
732237830990925431251526e-19.
(1 / T[56]) = (1 / 6908521828386340864) =
1.447487646186644772414255907553433559482031998270172859763243089405854169271
492537518497556447982788e-19.

(1 / T[57]) = (1 / 6404118670120845312) =
1.561495111990374881012829509246659018763186032708235002030417283916396975484985887305811047554016113e-19.
(1 / T[58]) = (1 / 2504001392817995776) =
3.993608002248764533904721075727107814794064670717825874404711744211637913792856124928221106529235845e-19.
(1 / T[59]) = (1 / 162129586585337856) =
6.167905692361980780047696335557194624433334290686418085132894893740651554026044323109090328216552734e-18.
(1 / T[60]) = (1 / 9727775195120271360) =
1.027984282060330130051050848789682979364409388145885355708833552705085134792994949748390354216098785e-19.
(1 / T[61]) = (1 / 3098476543630901248) =
3.227392513445222501346303134530375502231830080379847252252055705545180641635738538752775639295578003e-19.
(1 / T[62]) = (1 / 7638104968020361216) =
1.309225264888156297662723225479803458091686396959119364087818874206950447991459896002197638154029846e-19.
(1 / T[63]) = (1 / 1585267068834414592) =
6.308085367188389434396160928817448140363634405414144825864533171894121821310363884549587965011596680e-19.
(1 / T[64]) = (1 / 9223372036854775808) =
1.0842021724855044340074528008699417114257812 5e-19.

A := A + (1 / (0)!)
= 0 + (1 / 1)
= 0 + 1
= 1.

A := A + (1 / (1)!)
= 1 + (1 / 1)
= 1 + 1
= 2.

A := A + (1 / (2)!)
= 2 + (1 / 2)
= 2 + 0.5
= 2.5.

A := A + (1 / (3)!)
= 2.5 + (1 / 6)
= 2.5 + 0.1666666666666666666711841757186896018083643866702914237976074218 75
= 2.6666666666666666667389468114990336289338301867246627807617187 5.

A := A + (1 / (4)!)
= 2.6666666666666666673894681149903362893383018672466278076171875 + (1 / 24)
= 2.6666666666666666673894681149903362893383018672466278076171875 +
0.0416666666666666666779604392967240045209109666757285594940185546875
= 2.7083333333333333334778936229980672578676603734493255615234375.

A := A + (1 / (5)!)
= 2.7083333333333333334778936229980672578676603734493255615234375 + (1 / 120)
= 2.7083333333333333334778936229980672578676603734493255615234375 +
0.008333333333333333337286153753853401582318838336504995822906494140625
= 2.71666666666666666691241915909671433837502263486385345458984375.

A := A + (1 / (6)!)
= 2.71666666666666666691241915909671433837502263486385345458984375 + (1 / 720)
= 2.71666666666666666691241915909671433837502263486385345458984375 +
0.001388888888888888888488901108241024839884403263567946851253509521484375
= 2.7180555555555555558543134875293389995931647717952728271484375.

A := A + (1 / (7)!)
= 2.7180555555555555558543134875293389995931647717952728271484375 + (1 / 5040)
= 2.7180555555555555558543134875293389995931647717952728271484375 +
0.000198412698412698412698370682889521116054609706225164700299501419067382812
5
= 2.71825396825396825421262969602054226925247348845005035400390625.

A := A + (1 / (8)!)
= 2.71825396825396825421262969602054226925247348845005035400390625 + (1 /
40320)
= 2.71825396825396825421262969602054226925247348845005035400390625 +
2.48015873015873015872963353611901395068262132781455875374376773834228515625e
-05
= 2.71827876984126984142610405914552984540932811796665191650390625.

A := A + (1 / (9)!)
= 2.71827876984126984142610405914552984540932811796665191650390625 + (1 /
362880)
= 2.71827876984126984142610405914552984540932811796665191650390625 +
2.755731922398589065186216655752818750074739639899235044140368700027465820312
5e-06
= 2.71828152557319224010175251482479552578297443687915802001953125.

A := A + (1 / (10)!)
= 2.71828152557319224010175251482479552578297443687915802001953125 + (1 /
3628800)

= 2.71828152557319224010175251482479552578297443687915802001953125 +
2.755731922398589065134517867468254520395276596644862365792505443096160888671875e-07
= 2.71828180114638447996931736039272209382033906877040863037109375.

A := A + (1 / (11)!)
= 2.71828180114638447996931736039272209382033906877040863037109375 + (1 / 39916800)
= 2.71828180114638447996931736039272209382033906877040863037109375 +
2.505210838544171877468452021966260117517670547027108796100947074592113494873046875e-08
= 2.718281826198492865352684955126960630877874791622161865234375.

A := A + (1 / (12)!)
= 2.718281826198492865352684955126960630877874791622161865234375 + (1 / 479001600)
= 2.718281826198492865352684955126960630877874791622161865234375 +
2.0876756987868098978903766849718834312647254558559239967507892288267612457275390625e-09
= 2.71828182828616856411656221848005543506587855517864227294921875.

A := A + (1 / (13)!)
= 2.71828182828616856411656221848005543506587855517864227294921875 + (1 / 6227020800)
= 2.71828182828616856411656221848005543506587855517864227294921875 +
1.60590438368216145992538343035032278473177931761572967417350810137577354907989501953125e-10
= 2.7182818284467590024162941819696470702183432877063751220703125.

A := A + (1 / (14)!)
= 2.7182818284467590024162941819696470702183432877063751220703125 + (1 / 87178291200)
= 2.7182818284467590024162941819696470702183432877063751220703125 +
1.1470745597729724713978127618279737549630346144478865166860259705572389066219329833984375e-11
= 2.71828182845822974799364357689768212367198430001735687255859375.

A := A + (1 / (15)!)
= 2.71828182845822974799364357689768212367198430001735687255859375 + (1 / 1307674368000)
= 2.71828182845822974799364357689768212367198430001735687255859375 +
7.6471637318198164760182876165707005249790527865393595374765567385111353360116481781005859375e-13
= 2.71828182845899446440883495679230463792919181287288665771484375.

A := A + (1 / (16)!)
   = 2.71828182845899446440883495679230463792919181287288665771484375 + (1 / 20922789888000)
   = 2.71828182845899446440883495679230463792919181287288665771484375 + 4.779477332387385297511429760356687828111907991587099710922847961569459585007280111313128662109375e-14
   = 2.718281828459042259076364200787168101669522002339363098144531 25.

A := A + (1 / (17)!)
   = 2.71828182845904225907636420078716810166952200233936309814453125 + (1 / 355687428096000)
   = 2.71828182845904225907636420078716810166952200233936309814453125 + 2.811457254345520763162714508382106657881170315062443999191282850702577889023814350366592407226562 5e-15
   = 2.71828182845904507062943789019726636979612521827220916748046875.

A := A + (1 / (18)!)
   = 2.71828182845904507062943789019726636979612521827220916748046875 + (1 / 6402373705728000)
   = 2.71828182845904507062943789019726636979612521827220916748046875 + 1.561920696858622646281755141228416303811315922642396415600911374621517779814894311130046844482421875e-16
   = 2.71828182845904522675455072810990486686932854354381561279296875.

A := A + (1 / (19)!)
   = 2.71828182845904522675455072810990486686932854354381561279296875 + (1 / 121645100408832000)
   = 2.71828182845904522675455072810990486686932854354381561279296875 + 8.220635246624329716718057091551259700512195415301490541412415477551256515198474517092108726501464844e-18
   = 2.71828182845904523499448723899973856532596983015537261962890625.

A := A + (1 / (20)!)
   = 2.71828182845904523499448723899973856532596983015537261962890625 + (1 / 2432902008176640000)
   = 2.71828182845904523499448723899973856532596983015537261962890625 + 4.110317623312164858406048319808521350574847169139635097818954361046511758459587326797191053628921509e-19
   = 2.71828182845904523542816810799394033892895095050334930419921875.

A := A + (1 / (21)!)
   = 2.71828182845904523542816810799394033892895095050334930419921875 + (1 / 14197454024290336768)

= 2.71828182845904523542816810799394033892895095050334930419921875 +
7.043516381804132984552581733844016154884793905449790136826614643386981762240
850457601482048630714417e-20
= 2.71828182845904523542816810799394033892895095050334930419921875.

A := A + (1 / (22)!)
= 2.71828182845904523542816810799394033892895095050334930419921875 + (1 /
17196083355034583040)
= 2.71828182845904523542816810799394033892895095050334930419921875 +
5.815277696401867087825620308901504154687317329562106344189912444453752216055
875123856822028756141663e-20
= 2.71828182845904523542816810799394033892895095050334930419921875.

A := A + (1 / (23)!)
= 2.71828182845904523542816810799394033892895095050334930419921875 + (1 /
8128291617894825984)
= 2.71828182845904523542816810799394033892895095050334930419921875 +
1.230270820744732847600809726906469170638222121705104977548044832558193917293
465347029268741607666016e-19
= 2.718281828459045235645008542491041225730441510677337646484375.

A := A + (1 / (24)!)
= 2.718281828459045235645008542491041225730441510677337646484375 + (1 /
10611558092380307456)
= 2.718281828459045235645008542491041225730441510677337646484375 +
9.423686807294170693318948175954948019683466235544601296535251525304105468805
460077419411391019821167e-20
= 2.718281828459045235645008542491041225730441510677337646484375.

A := A + (1 / (25)!)
= 2.718281828459045235645008542491041225730441510677337646484375 + (1 /
7034535277573963776)
= 2.718281828459045235645008542491041225730441510677337646484375 +
1.421558014198878427804392419215390985622496694920617818255280764058040565700
480328814592212438583374e-19
= 2.71828182845904523586184897698814211253193207085132598876953125.

A := A + (1 / (26)!)
= 2.71828182845904523586184897698814211253193207085132598876953125 + (1 /
16877220553537093632)
= 2.71828182845904523586184897698814211253193207085132598876953125 +
5.925146245662008780449842354395082473873705632309314560039469494409983263416
563659120583906769752502e-20
= 2.71828182845904523586184897698814211253193207085132598876953125.

A := A + (1 / (27)!)
   = 2.71828182845904523586184897698814211253193207085132598876953125 + (1 / 10888869450418352160768000000)
   = 2.71828182845904523586184897698814211253193207085132598876953125 +
7.7142058443793512204423075902838678884471538571278549160926930276099833250214032887015491724014282 23e-20
   = 2.71828182845904523586184897698814211253193207085132598876953125.

A := A + (1 / (28)!)
   = 2.71828182845904523586184897698814211253193207085132598876953125 + (1 / 12478583540742619136)
   = 2.71828182845904523586184897698814211253193207085132598876953125 +
8.0137300578627092086859902011804041023599113547452436251852415515124772316113421766203828155994415 28e-20
   = 2.71828182845904523586184897698814211253193207085132598876953125.

A := A + (1 / (29)!)
   = 2.71828182845904523586184897698814211253193207085132598876953125 + (1 / 11390785281054474240)
   = 2.71828182845904523586184897698814211253193207085132598876953125 +
8.7790259874640305081217189946623289902356237842522740025139944180024298425735196360619738698005676 27e-20
   = 2.71828182845904523586184897698814211253193207085132598876953125.

A := A + (1 / (30)!)
   = 2.71828182845904523586184897698814211253193207085132598876953125 + (1 / 9682165104862298112)
   = 2.71828182845904523586184897698814211253193207085132598876953125 +
1.0328268410727769970269506977306992086253612994639861891382604219260729627727357637922978028655052 19e-19
   = 2.71828182845904523586184897698814211253193207085132598876953125.

A := A + (1 / (31)!)
   = 2.71828182845904523586184897698814211253193207085132598876953125 + (1 / 4999213071378415616)
   = 2.71828182845904523586184897698814211253193207085132598876953125 +
2.0003148209969643910023841528277055457959979861256514315324526347757844169095164943428244441747665 41e-19
   = 2.71828182845904523607868941148524299333422631025314331054687 5.

A := A + (1 / (32)!)
   = 2.718281828459045236078689411485242999333422631025314331054687 5 + (1 / 12400865694432886784)

= 2.71828182845904523607868941148524299933342263102531433105468 75 +
8.0639531516652857681868084465746147844769788480833382701906305309921124679917170396947767585515975 95e-20
= 2.718281828459045236078689411485242999333422631025314331054687 5.

A := A + (1 / (33)!)
= 2.718281828459045236078689411485242999333422631025314331054687 5 + (1 / 3400198294675128320)
= 2.718281828459045236078689411485242999333422631025314331054687 5 +
2.94100494540582352051641419389346837770780057644808278616539932081783148554166018584510311484336853e-19
= 2.7182818284590452362955298459823438861349131911993026733398437 5.

A := A + (1 / (34)!)
= 2.7182818284590452362955298459823438861349131911993026733398437 5 + (1 / 4926277576697053184)
= 2.7182818284590452362955298459823438861349131911993026733398437 5 +
2.02993027581380254683621326314067805623130551353550061969255619985681715622405363319558091461658477 8e-19
= 2.71828182845904523651237028047944477293640375137329101562 5.

A := A + (1 / (35)!)
= 2.71828182845904523651237028047944477293640375137329101562 5 + (1 / 6399018521010896896)
= 2.71828182845904523651237028047944477293640375137329101562 5 +
1.56273965564647738089269655255415549676173894995313141070855211638146453623221532325260341167449951 2e-19
= 2.7182818284590452367292107149765456597378943115472793579101562 5.

A := A + (1 / (36)!)
= 2.7182818284590452367292107149765456597378943115472793579101562 5 + (1 / 9003737871877668864)
= 2.7182818284590452367292107149765456597378943115472793579101562 5 +
1.11064983702313930004254956120519056151595332327093906754119699109399632686034919970552437007427215 6e-19
= 2.71828182845904523694605114947364654653938487172126770019531 25.

A := A + (1 / (37)!)
= 2.71828182845904523694605114947364654653938487172126770019531 25 + (1 / 10969079327018188 80)
= 2.71828182845904523694605114947364654653938487172126770019531 25 +
9.11653540089620150092994344193313212387470003239665768845733360460636962585567744099535048007965087 9e-19
= 2.718281828459045237813412887462050093745347112417221069335937 5.

A := A + (1 / (38)!)
   = 2.7182818284590452378134128874620500937453471124172210693359375 + (1 / 4789013295250014208)
   = 2.7182818284590452378134128874620500937453471124172210693359375 + 2.088112808105691869924731083569285259541630716029320990911684108098586576396371583541622385382652283e-19
   = 2.71828182845904523803025332195915098054683767259120941162109375.

A := A + (1 / (39)!)
   = 2.71828182845904523803025332195915098054683767259120941162109375 + (1 / 2304077777655037952)
   = 2.71828182845904523803025332195915098054683767259120941162109375 + 4.340131265090123433015478527485500881114515851366072819926298178606904887288919780985452234745025635e-19
   = 2.71828182845904523846393419095335275414981879293918609619140625.

A := A + (1 / (40)!)
   = 2.71828182845904523846393419095335275414981879293918609619140625 + (1 / 18376134811363311616)
   = 2.71828182845904523846393419095335275414981879293918609619140625 + 5.441840791141925413183004478172199152903907339519060119881344722830157634163583679764997214078903198e-20
   = 2.71828182845904523846393419095335275414981879293918609619140625.

A := A + (1 / (41)!)
   = 2.71828182845904523846393419095335275414981879293918609619140625 + (1 / 15551764317513711616)
   = 2.71828182845904523846393419095335275414981879293918609619140625 + 6.430138597675660950353326517785726873973361451550566389280682460562188484942680588574148714542388916e-20
   = 2.71828182845904523846393419095335275414981879293918609619140625.

A := A + (1 / (42)!)
   = 2.71828182845904523846393419095335275414981879293918609619140625 + (1 / 7538058755741581312)
   = 2.71828182845904523846393419095335275414981879293918609619140625 + 1.326601493041323093804705908412158628351599110254046220970487675360274804070570553449215367436408997e-19
   = 2.7182818284590452386807746254504536409513093531131744384765625.

A := A + (1 / (43)!)
   = 2.7182818284590452386807746254504536409513093531131744384765625 + (1 / 10541877243825618944)

= 2.7182818284590452386807746254504536409513093531131744384765625 +
9.485976518894681088585945984202713644634515871086129452208364070146459634536
029170703841373324394226e-20
= 2.7182818284590452386807746254504536409513093531131744384765625.

A := A + (1 / (44)!)
= 2.7182818284590452386807746254504536409513093531131744384765625 + (1 /
2673996885588443136)
= 2.7182818284590452386807746254504536409513093531131744384765625 +
3.739720137257896377418136692574652015537340387383811719072360135220078891649
109209538437426090240479e-19
= 2.718281828459045239114455494444655414554290473461151123046875.

A := A + (1 / (45)!)
= 2.718281828459045239114455494444655414554290473461151123046875 + (1 /
9649395409222631424)
= 2.718281828459045239114455494444655414554290473461151123046875 +
1.036334358362210981733926214460651797062864929183711526852237064981812619812
728826218517497181892395e-19
= 2.718281828459045239114455494444655414554290473461151123046875.

A := A + (1 / (46)!)
= 2.718281828459045239114455494444655414554290473461151123046875 + (1 /
1150331055211806720)
= 2.718281828459045239114455494444655414554290473461151123046875 +
8.693149641308025443189145265311138141228065069562171980261825821392762669859
166635433211922645568848e-19
= 2.7182818284590452399818172324330589617602527141571044921875.

A := A + (1 / (47)!)
= 2.7182818284590452399818172324330589617602527141571044921875 + (1 /
17172071447535812608)
= 2.7182818284590452399818172324330589617602527141571044921875 +
5.823409266932089994910827473586939922642565582829483048515030398258052191096
112437662668526172637939e-20
= 2.7182818284590452399818172324330589617602527141571044921875.

A := A + (1 / (48)!)
= 2.7182818284590452399818172324330589617602527141571044921875 + (1 /
12602690238498734080)
= 2.7182818284590452399818172324330589617602527141571044921875 +
7.934813766549598658208990425096079332701638958718522115960120841835281901843
757168535375967621803284e-20
= 2.7182818284590452399818172324330589617602527141571044921875.

A := A + (1 / (49)!)
= 2.7182818284590452399818172324330589617602527141571044921875 + (1 / 8789267254022766592)
= 2.7182818284590452399818172324330589617602527141571044921875 + 1.137751272203390128107638181047964849359748082386584368096104198870080481675870487379143014550209045e-19
= 2.71828182845904524019865766693015984856174327433109283447265625.

A := A + (1 / (50)!)
= 2.71828182845904524019865766693015984856174327433109283447265625 + (1 / 15188249005818642432)
= 2.71828182845904524019865766693015984856174327433109283447265625 + 6.584037433261058626315122747133625100946933977144928122763979037304279962050657104555284604430198669e-20
= 2.71828182845904524019865766693015984856174327433109283447265625.

A := A + (1 / (51)!)
= 2.71828182845904524019865766693015984856174327433109283447265625 + (1 / 18284192274659147776)
= 2.71828182845904524019865766693015984856174327433109283447265625 + 5.469205229185558363221202926338644452360917504749919928199492245380133881305084742052713409066200256e-20
= 2.71828182845904524019865766693015984856174327433109283447265625.

A := A + (1 / (52)!)
= 2.71828182845904524019865766693015984856174327433109283447265625 + (1 / 9994050523088551936)
= 2.71828182845904524019865766693015984856174327433109283447265625 + 1.000595301864614693116746035656340128581767350489224182304672495732490722364360635765478946268558502e-19
= 2.71828182845904524019865766693015984856174327433109283447265625.

A := A + (1 / (53)!)
= 2.71828182845904524019865766693015984856174327433109283447265625 + (1 / 13175843659825807360)
= 2.71828182845904524019865766693015984856174327433109283447265625 + 7.589646825038455307200389676950583077870852261845009400057856808466047460193237839121138677000999451e-20
= 2.71828182845904524019865766693015984856174327433109283447265625.

A := A + (1 / (54)!)
= 2.71828182845904524019865766693015984856174327433109283447265625 + (1 / 10519282829630636032)

= 2.71828182845904524019865766693015984856174327433109283447265625 + 9.50635148988681614421616430881190891954527936199995567880738020234004936881433422968257218599319458e-20
= 2.71828182845904524019865766693015984856174327433109283447265625.

A := A + (1 / (55)!)
= 2.71828182845904524019865766693015984856174327433109283447265625 + (1 / 6711489344688881664)
= 2.71828182845904524019865766693015984856174327433109283447265625 + 1.489982250797056252874586363401948231626454775496495728635927216368731174078732237830990925431251526e-19
= 2.71828182845904524041549810142726073536323383450508117675781250.

A := A + (1 / (56)!)
= 2.71828182845904524041549810142726073536323383450508117675781250 + (1 / 6908521828386340864)
= 2.71828182845904524041549810142726073536323383450508117675781250 + 1.447487646186644772414255907553433559482031998270172859763243089405854169271492537518497556447982788e-19
= 2.71828182845904524063233853592436162216472439467906951904296875.

A := A + (1 / (57)!)
= 2.71828182845904524063233853592436162216472439467906951904296875 + (1 / 6404118670120845312)
= 2.71828182845904524063233853592436162216472439467906951904296875 + 1.561495111990374881012829509246659018763186032708235002030417283916396975484985887305811047554016113e-19
= 2.71828182845904524084917897042146250896621495485305786132812500.

A := A + (1 / (58)!)
= 2.71828182845904524084917897042146250896621495485305786132812500 + (1 / 2504001392817995776)
= 2.71828182845904524084917897042146250896621495485305786132812500 + 3.99360800224876453390472107572710781479406467071782587440471174421163791379285612492822110652923584e-19
= 2.71828182845904524128285983941566428256919607520103454589843750.

A := A + (1 / (59)!)
= 2.71828182845904524128285983941566428256919607520103454589843750 + (1 / 162129586585337856)
= 2.71828182845904524128285983941566428256919607520103454589843750 + 6.167905692361980780047696335557194624433334290686418085132894893740651554026044323109090328216552734e-18
= 2.71828182845904524735439200533448911301093176007270812988281250.

A := A + (1 / (60)!)
= 2.7182818284590452473543920053344891130109317600727081298828125 + (1 / 9727775195120271360)
= 2.7182818284590452473543920053344891130109317600727081298828125 + 1.027984282060330130051050848789682979364409388145885355708833552705085134792994949748390354216098785e-19
= 2.7182818284590452473543920053344891130109317600727081298828125.

A := A + (1 / (61)!)
= 2.7182818284590452473543920053344891130109317600727081298828125 + (1 / 3098476543630901248)
= 2.7182818284590452473543920053344891130109317600727081298828125 + 3.227392513445222501346303134530375502231830080379847252252055705545180641635738538752775639295578003e-19
= 2.71828182845904524757123243983158999981242232024669647216796875.

A := A + (1 / (62)!)
= 2.71828182845904524757123243983158999981242232024669647216796875 + (1 / 7638104968020361216)
= 2.71828182845904524757123243983158999981242232024669647216796875 + 1.309225264888156297662723225479803458091686396959119364087818874206950447991459896002197638154029846e-19
= 2.718281828459045247788072874328690886613912880420684814453125.

A := A + (1 / (63)!)
= 2.718281828459045247788072874328690886613912880420684814453125 + (1 / 1585267068834414592)
= 2.718281828459045247788072874328690886613912880420684814453125 + 6.30808536718838943439616092881744814036363440541414482586453317189412182131036388454958796501159668e-19
= 2.71828182845904524843859417781999354701838456094264984130859375.

A := A + (1 / (64)!)
= 2.71828182845904524843859417781999354701838456094264984130859375 + (1 / 9223372036854775808)
= 2.71828182845904524843859417781999354701838456094264984130859375 + 1.08420217248550443400745280086994171142578125e-19
= 2.71828182845904524865543461231709443381987512111663818359375.

B = eulers_number_approximation(N) :=
2.71828182845904524865543461231709443381987512111663818359375.

END OF PROGRAM

This web page was last updated on 17_OCTOBER_2023. The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

[End of abridged plain-text content from EULERS_NUMBER_APPROXIMATION]

NATURE

---

image_link: https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/cube_comprised_of_eight_equally_sized_cubes_03_may_2023.jpeg

---

The following terms and their respective definitions describe nature (i.e. the whole of reality) as being the set which contains all phenomena, all noumena, and all of pure nothingness.

NOTHINGNESS: the ubiquitous, changeless, and featureless substrate which permeates and encompasses all phenomena.

According to panpsychism (i.e. the hypothesis which posits that the entirety of nature is contained inside of one ubiquitous mind), pure nothingness is synonymous with pure consciousness (which means that nature is fundamentally sentient and that consciousness (i.e. awareness (i.e. sentience)) is not an emergent property of sufficiently complex physical structures, but instead, the fundamental substrate which encompasses and which constitutes all physical structures).

(The following quoted text was published as a Twitter post by karbytes on 06_AUGUST_2023: “I think that it is possible that what appears to a human to be that human’s own lifetime is really just a dream (and that whatever is dreaming that dream is an object which has always existed and always will exist (and which can dream up limitlessly many scenarios which are limitlessly varied)).”)

PHENOMENON: (plural: phenomena) a finite region of space which is distinguishable from pure nothingness due to the fact that the finite region of space is perceived (by some frame of reference) as containing specific quantifiable features.

An example of a phenomenon is a light sensor inside of an electronic circuit detecting incoming light whose wavelengths are exactly 700 nanometers and not some other wavelength such as 470 nanometers.

A human nervous system which perceives incoming light (which enters the eyes and which causes a corresponding electrical signal to be sent across the optical nerves and to the visual processing circuits of the brain) whose wavelengths are exactly 700 nanometers does not consciously register that the incoming light has a wavelength of exactly 700 nanometers. Instead, what that human nervous system immediately consciously experiences is the qualitative experience referred to as the color red.

The qualitative phenomenon referred to as the color red is distinct from the quantitative phenomenon referred to as the number 700.

FRAME_OF_REFERENCE: an allocation of pure nothingness which detects the presence of or the absence of a particular phenomenon.

A partial rather than omniscient frame of reference experiences the passage of time and time as progressing in exactly one direction (from past to future while that partial frame of reference functions as a barrier between what that frame of reference perceives to be the past and what that frame of reference perceives to be the future while that partial frame of reference appears to itself to always be dwelling in the present) and such that a relatively fixed rate of finite (in scope) spatial frames per second are being rendered by that partial frame of reference.

An omniscient rather than partial frame of reference would simultaneously render all partial frames of reference from a perspective which transcends all space-time continuums.

NOUMENON: (plural: noumena) a specific phenomenon which is not being observed by some partial frame of reference (according to that particular frame of reference).

Suppose that two unique and partial frames of reference exist and that one of those frames of reference, A, is observing some phenomenon, X, while the other frame of reference, B, is not observing X. Then, according to B, X is a noumenon.

This web page was last updated on 06_AUGUST_2023. The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

[End of abridged plain-text content from NATURE]

EXPONENTIATION

The C++ program featured in this tutorial web page computes the approximate value of some real number base (which the program user inputs) raised to the power of some real number exponent (which the program user also inputs). Such a mathematical operation is referred to as exponentiation. The C++ program featured in this web page uses the natural logarithm and the base of the natural logarithm (which is Euler's Number) raised to the power of some number to approximate base to the power of exponent. (Note that the C++ program featured in this web page does not include either cmath of math.h (which are each C++ libraries)).

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

Let base be a real number.
Let exponent be a real number.
Let power(base,exponent) return a real number equivalent to base raised to the power of exponent (i.e. base ^ exponent) using the following procedure:
procedure power(base, exponent):

    Let e be approximately equal to Euler's Number.
    Let exp(x) be e raised to the power of some real number x.
    Let log(x) be the base-e logarithm of some real number x.
    If exponent is zero: return one.
    If exponent is one: return base.
    If exponent is a whole number:
    If exponent is a positive number:
    return base multiplied by base exponent times.
    End if.
    If exponent is a negative number:
    return the reciprocal of the number represented by base multiplied by base exponent times.
    End if.
    End if.
    If exponent is not a whole number:
    If exponent is a negative number:
    return exp(log(base) * exponent).

End if.
If exponent is a positive number:
return exp(exp(log(base) * absolute_value(exponent))).
End if.
End if.
End procedure.

SOFTWARE_APPLICATION_COMPONENTS

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_extension_pack_2/main/power.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/power_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ source code into a new text editor document and save that document as the following file name:

power.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ power.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: After running the g++ command, run the executable file using the following command:

./app

STEP_5: Once the application is running, the following prompt will appear:

Enter a real number value for base which is no larger than 100 and no smaller than -100:

STEP_6: Enter a value for base using the using the keyboard.

STEP_7: After a value for base is entered, the following prompt will appear:

Enter a real number value for exponent which is no larger than 100 and no smaller than -100:

STEP_8: Enter a value for exponent using the keyboard.

STEP_9: A statement showing the value returned by the power function which computes the approximate value of base raised to the power of exponent will be printed to the command line terminal and then the following prompt will appear:

Would you like to continue inputting program values? (Enter 1 if YES. Enter 0 if NO):

STEP_10: Enter a value according to your preference until you decide to close the program (and save your program data to the output text file).

PROGRAM_SOURCE_CODE

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_extension_pack_2/main/power.cpp

---

```
/**
 * file: power.cpp
 * type: C++ (source file)
```

```
 * date: 18_OCTOBER_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#include < iostream > // standard input (std::cin), standard output (std::cout)
#include < fstream > // file input, file output
// #include < cmath > // exp() and log() functions
#define MAXIMUM_ABSOLUTE_VALUE_BASE 100 // constant which represents maximum
absolute value for base
#define MAXIMUM_ABSOLUTE_VALUE_EXPONENT 100 // constant which represents
maximum absolute value for exponent

/* function prototypes */
bool is_whole_number(double x);
double absolute_value(double x);
double power_of_e_to_x(double x);
float ln(float x);
double power(double base, double exponent);

/* program entry point */
int main()
{
	// Declare a file output stream object named file.
	std::ofstream file;

	// Declare three variables for storing floating-point number values.
	double base = 0.0, exponent = 0.0, result = 0.0;

	// Declare a variable for storing the program user's answer of whether or not to continue
inputting values.
	int input_additional_values = 1;

	// Set the number of digits of floating-point numbers which are printed to the command
line terminal to 100 digits.
	std::cout.precision(100);

	// Set the number of digits of floating-point numbers which are printed to the file output
stream to 100 digits.
	file.precision(100);

	/**
	* If the file named power_output.txt does not already exist
```

```
* inside of the same file directory as the file named power.cpp,
* create a new file named power_output.txt in that directory.
*
* Open the plain-text file named power_output.txt
* and set that file to be overwritten with program data.
*/
file.open("power_output.txt");

// Print an opening message to the command line terminal.
std::cout << "\n\n--------------------------------";
std::cout << "\nSTART OF PROGRAM";
std::cout << "\n--------------------------------";

// Print an opening message to the file output stream.
file << "--------------------------------";
file << "\nSTART OF PROGRAM";
file << "\n--------------------------------";

// Print some program-related data to the command line terminal.
std::cout << "\n\npower(base, exponent) = base ^ exponent.";

// Print some program-related data to the file output stream.
file << "\n\npower(base, exponent) = base ^ exponent.";

while (input_additional_values != 0)
{
// Print a divider line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a divider line to the file output stream.
file << "\n\n--------------------------------";

// Prompt the user to enter a value to store in the variable named base (to the command
line terminal).
std::cout << "\n\nEnter a real number value for base which is no larger than ";
std::cout << MAXIMUM_ABSOLUTE_VALUE_BASE;
std::cout << " and no smaller than ";
std::cout << (-1 * MAXIMUM_ABSOLUTE_VALUE_BASE) << ": ";

// Print the prompt for entering a base value to the file output stream.
file << "\n\nEnter a real number value for base which is no larger than ";
file << MAXIMUM_ABSOLUTE_VALUE_BASE;
file << " and no smaller than ";
file << (-1 * MAXIMUM_ABSOLUTE_VALUE_BASE) <> base;
```

```
// Print the most recently input keyboard value to the command line terminal.
std::cout << base;

// Print the most recently input keyboard value to the file output stream.
file << base;

// Prompt the user to enter a value to store in the variable named exponent (to the command line terminal).
std::cout << "\n\nEnter a real number value for exponent which is no larger than ";
std::cout << MAXIMUM_ABSOLUTE_VALUE_EXPONENT;
std::cout << " and no smaller than ";
std::cout << (-1 * MAXIMUM_ABSOLUTE_VALUE_EXPONENT) << ": ";

// Print the prompt for entering an exponent value to the output file.
file << "\n\nEnter a real number value for exponent which is no larger than ";
file << MAXIMUM_ABSOLUTE_VALUE_EXPONENT;
file << " and no smaller than ";
file << (-1 * MAXIMUM_ABSOLUTE_VALUE_EXPONENT) <> exponent;

// Print the most recently input keyboard value to the command line terminal.
std::cout << exponent;

// Print the most recently input keyboard value to the file output stream.
file << exponent;

// If base is not within the range of accepted values, set base to 1.
if ((base  MAXIMUM_ABSOLUTE_VALUE_BASE))
{
base = 1;
std::cout << "\n\nBecause the input value for base was not within the range of accepted values, base was set to the default value of 1.";
file << "\n\nBecause the input value for base was not within the range of accepted values, base was set to the default value of 1.";
}

// If exponent is not within the range of accepted values, set exponent to 0.
if ((exponent  MAXIMUM_ABSOLUTE_VALUE_EXPONENT))
{
exponent = 0;
std::cout << "\n\nBecause the input value for exponent was not within the range of accepted values, exponent was set to the default value of 0.";
file << "\n\nBecause the input value for exponent was not within the range of accepted values, exponent was set to the default value of 0.";
```

```
	}

	// Compute base to the power of exponent.
	result = power(base, exponent);

	// Print the result returned by the power function defined in this program to the command line terminal.
	std::cout << "\n\nresult = power(base,exponent) = power(" << base << ", " << exponent << ") = " << base << " ^ " << exponent << " = " << result << ".";

	// Print the result returned by the power function defined in this program to the file output stream.
	file << "\n\nresult = power(base,exponent) = power(" << base << ", " << exponent << ") = " << base << " ^ " << exponent << " = " << result << ".";

	// Ask the user whether or not to continue inputing values.
	std::cout <> input_additional_values;
	}

	// Print a closing message to the command line terminal.
	std::cout << "\n--------------------------------";
	std::cout << "\nEND OF PROGRAM";
	std::cout << "\n--------------------------------\n\n";

	// Print a closing message to the file output stream.
	file << "\n\n--------------------------------";
	file << "\nEND OF PROGRAM";
	file << "\n--------------------------------";

	// Close the file output stream.
	file.close();

	// Exit the program.
	return 0;
}

/**
 * If x is determined to be a whole number, return true.
 * Otherwise, return false.
 */
bool is_whole_number(double x)
{
	return (x == (long int) x);
}
```

```
/**
 * Return the absolute value of a real number input, x.
 */
double absolute_value(double x)
{
	if (x < 0) return -1 * x;
	return x;
}

/**
 * Return the approximate value of Euler's Number to the power of some real number x.
 *
 * This function is essentially identical to the C++ library math.h function exp().
 */
double power_of_e_to_x(double x) {
	double a = 1.0, e = a;
	int n = 1;
	int invert = x < 0;
	x = absolute_value(x);
	for (n = 1; e != e + a; n += 1) {
	a = a * x / n;
	e += a;
	}
	return invert ? (1 / e) : e;
}

//--------------------------------------------------------------------------------------------
// The following function and associated comments were not written by karbytes.
//
// The following function is essentially identical to the C++ library math.h function log().
//--------------------------------------------------------------------------------------------

// ln.c
//
// simple, fast, accurate natural log approximation
// when without

// featuring * floating point bit level hacking,
//	* x=m*2^p => ln(x)=ln(m)+ln(2)p,
//	* Remez algorithm

// by Lingdong Huang, 2020. Public domain.
```

```
// ============================================

float ln(float x) {
  unsigned int bx = * (unsigned int *) (&x);
  unsigned int ex = bx >> 23;
  signed int t = (signed int)ex-(signed int)127;
  unsigned int s = (t < 0) ? (-t) : t;
  bx = 1065353216 | (bx & 8388607);
  x = * (float *) (&bx);
  return -1.49278+(2.11263+(-0.729104+0.10969*x)*x)*x+0.6931471806*t;
}
// done.

//--------------------------------------------------------------------------------------------
// End of code which was not written by karbytes.
//--------------------------------------------------------------------------------------------

/**
 * Reverse engineer the cmath pow() function
 * using the following properties of natural logarithms:
 *
 * ln(x ^ y) = y * ln(x).
 *
 * ln(e ^ x) = x. // e is approximately Euler's Number.
 *
 * Note that the base of the logarithmic function
 * used by the cmath log() function is e.
 *
 * Hence, log(x) is approximately the
 * natural log of x (i.e. ln(x)).
 *
 * Note that the base of the exponential function
 * used by the cmath exp() function is
 * (approximately) Euler's Number.
 *
 * Hence, exp(x) is approximately
 * x ^ e (where e is approximately Euler's Number).
 *
 * Note that any number, x, raised to the power of 0 is 1.
 * In more succinct terms, x ^ 0 = 1.
 *
 * Note that any number, x, raised to the power of 1 is x.
 * In more succinct terms, x ^ 1 = x.
 *
```

```
 * Note that any whole number, x,
 * raised to the power of a positive whole number exponent, y,
 * is x multiplied by itself y times.
 * For example, if x is 2 and y is 3,
 * 2 ^ 3 = power(2, 3) = 2 * 2 * 2 = 8.
 *
 * Note that any whole number, x,
 * raised to the power of a negative exponent, y,
 * is 1 / (x ^ (-1 * y)).
 * For example, if x is 2 and y is -3,
 * 2 ^ -3 = power(2, -3) = 1 / (2 * 2 * 2) = 1 / 8 = 0.125.
 */
double power(double base, double exponent)
{
	double output = 1.0;
	if (exponent == 0) return 1;
	if (exponent == 1) return base;
	// if ((base == 0) && (exponent  0)
	{
	while (exponent > 0)
	{
		output *= base;
		exponent -= 1;
	}
	return output;
	}
	else
	{
	exponent = absolute_value(exponent);
	while (exponent > 0)
	{
		output *= base;
		exponent -= 1;
	}
	return 1 / output;
	}
	}
	if (exponent > 0) return power_of_e_to_x(ln(base) * exponent); // Return e ^ (ln(base) *
exponent).
	return power_of_e_to_x(power_of_e_to_x(ln(base) * absolute_value(exponent))); //
Return e ^ (e ^ (ln(base) * absolute_value(exponent))).
}
```

SAMPLE_PROGRAM_OUTPUT

Note that the value returned by the function call pow(-1, 0.5) (and pow() is a function encapsulated by the C++ libraries cmath and math.h) returned the value -nan.

By contrast, the value returned by the function call power(-1, 0.5) (and power() is a function defined in power.cpp) returned the value 340357746532624088885911455895174250496.

That is because -1 to the power of 0.5 is an imaginary number instead of a real number.

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/power_output.txt

----
START OF PROGRAM
----

power(base, exponent) = base ^ exponent.

Enter a real number value for base which is no larger than 100 and no smaller than -100: 0

Enter a real number value for exponent which is no larger than 100 and no smaller than -100: 0

result = power(base,exponent) = power(0, 0) = 0 ^ 0 = 1.

Enter a real number value for base which is no larger than 100 and no smaller than -100: 2

Enter a real number value for exponent which is no larger than 100 and no smaller than -100: 3

result = power(base,exponent) = power(2, 3) = 2 ^ 3 = 8.

Enter a real number value for base which is no larger than 100 and no smaller than -100: -2

Enter a real number value for exponent which is no larger than 100 and no smaller than -100: 3

result = power(base,exponent) = power(-2, 3) = -2 ^ 3 = -8.

Enter a real number value for base which is no larger than 100 and no smaller than -100: 2

Enter a real number value for exponent which is no larger than 100 and no smaller than -100: -3

result = power(base,exponent) = power(2, -3) = 2 ^ -3 = 0.125.

Enter a real number value for base which is no larger than 100 and no smaller than -100: -2

Enter a real number value for exponent which is no larger than 100 and no smaller than -100: -3

result = power(base,exponent) = power(-2, -3) = -2 ^ -3 = -0.125.

Enter a real number value for base which is no larger than 100 and no smaller than -100: 2

Enter a real number value for exponent which is no larger than 100 and no smaller than -100:
0.5

result = power(base,exponent) = power(2, 0.5) = 2 ^ 0.5 =
1.41452190152525414390538571751676499843597412109375.

Enter a real number value for base which is no larger than 100 and no smaller than -100: 2

Enter a real number value for exponent which is no larger than 100 and no smaller than -100: -2

result = power(base,exponent) = power(2, -2) = 2 ^ -2 = 0.25.

Enter a real number value for base which is no larger than 100 and no smaller than -100: 2

Enter a real number value for exponent which is no larger than 100 and no smaller than -100: -1

result = power(base,exponent) = power(2, -1) = 2 ^ -1 = 0.5.

Enter a real number value for base which is no larger than 100 and no smaller than -100: 0.5

Enter a real number value for exponent which is no larger than 100 and no smaller than -100: -1

result = power(base,exponent) = power(0.5, -1) = 0.5 ^ -1 = 2.

Enter a real number value for base which is no larger than 100 and no smaller than -100: 0.5

Enter a real number value for exponent which is no larger than 100 and no smaller than -100:
0.5

result = power(base,exponent) = power(0.5, 0.5) = 0.5 ^ 0.5 =
0.7072609494155397413805985706858336925506591796875.

Enter a real number value for base which is no larger than 100 and no smaller than -100: 100

Enter a real number value for exponent which is no larger than 100 and no smaller than -100:
0.5

result = power(base,exponent) = power(100, 0.5) = 100 ^ 0.5 =
10.0015620834400120742202489054761826992034912109375.

Enter a real number value for base which is no larger than 100 and no smaller than -100: -100

Enter a real number value for exponent which is no larger than 100 and no smaller than -100:
0.5

result = power(base,exponent) = power(-100, 0.5) = -100 ^ 0.5 =
3403358482654129315574312994291840974848.

END OF PROGRAM

[End of abridged plain-text content from EXPONENTIATION]

POINTERS_AND_ARRAYS

The C++ program featured in this tutorial web page illustrates how to use pointer variables to instantiate arrays during program runtime. The program first prompts the user to enter a natural number value to store in a variable named S. Then the program prompts the user to enter a natural number value to store in a variable named T. Then the program will create an array named A consisting of exactly S integer values such that each of those S integer values is a random nonnegative integer value which is no larger than (T – 1). Then the program will sort the values which are stored in A in ascending order using the Bubble Sort algorithm. Then an array consisting of exactly T elements will store the number of times each unique value occurred as an element value of A. Finally, a histogram (i.e. bar graph) representation of C will be created using the values of B.

A pointer is a variable which stores the memory address of a variable. An array is a variable which is used to store some natural number of data values of the same data type. At the hardware level, an array comprised of N elements whose data type is DATA_TYPE is a contiguous block of DATA_TYPE multiplied by N memory cells.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

```
int N = 99; // an int type variable which stores the initial value 99
int * P = &N; // a pointer-to-int type variable which stores the address of N
std::cout << P; // 0x559343ab78fc (memory address of one byte-sized memory cell (and the first of four contiguous memory cells allocated to N))
std::cout << * P; // 99 (retrieved data value which is stored at the memory address which P stores)
int * K = new int [N]; // A pointer-to-int type variable named K is used to store the memory address of the first of N int-sized contiguous chunks of memory.
```

SOFTWARE_APPLICATION_COMPONENTS

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/arrays.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/arrays_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ source code into a new text editor document and save that document as the following file name:

arrays.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ arrays.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: After running the g++ command, run the executable file using the following command:

./app

STEP_5: Once the application is running, the following prompt will appear:

Enter a natural number, S, for representing the number of elements to include in an array which is no larger than 1000:

STEP_6: Enter a value for S using the keyboard.

STEP_7: Another prompt for keyboard input will appear after the first input value is entered:

Enter a natural number, T, for representing the number of unique states which each element of the array can store exactly one of which is no larger than 1000:

STEP_8: Enter a value for T using the keyboard.

STEP_9: Observe program results on the command line terminal and in the output file.

PROGRAM_SOURCE_CODE

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/arrays.cpp

```
/**
 * file: arrays.cpp
 * type: C++ (source file)
 * date: 05_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#include < iostream > // standard input (std::cin), standard output (std::cout)
#include < fstream > // file input, file output
#include < stdio.h > // NULL macro
#include < stdlib.h > // srand(), rand()
#include < time.h > // time()
#define MAXIMUM_S 1000 // constant which represents maximum value for S
#define MAXIMUM_T 1000 // constant which represents maximum value for T
```

```
/* function prototype */
void bubble_sort(int * A, int S);

/**
 * Use the Bubble Sort algorithm to arrange the elements of an int type array, A,
 * in ascending order
 * such that A[0] represents the smallest integer value in that array and
 * such that A[S - 1] represents the largest integer value in that array.
 *
 * Assume that S is a natural number no larger than MAXIMUM_S.
 *
 * Assume that A is a pointer to an int type variable and that
 * A stores the memory address of the first element, A[0],
 * of an int type array comprised of exactly S elements.
 * (In other words, assume that exactly S consecutive int-sized
 * chunks of memory are allocated to the array represented by A).
 *
 * Although this function returns no value,
 * the array which the pointer variable, A, points to is updated
 * if the elements of that array are not already arranged in ascending order.
 */
void bubble_sort(int * A, int S)
{
	int i = 0, placeholder = 0;
	bool array_is_sorted = false, adjacent_elements_were_swapped = false;
	while (!array_is_sorted)
	{
	adjacent_elements_were_swapped = false;
	for (i = 1; i < S; i += 1)
	{
	if (A[i] < A[i - 1])
	{
		placeholder = A[i];
		A[i] = A[i - 1];
		A[i - 1] = placeholder;
		adjacent_elements_were_swapped = true;
	}
	}
	if (!adjacent_elements_were_swapped) array_is_sorted = true;
	}
}

/* program entry point */
int main()
```

```
{
	// Declare four int type variables and set each of their initial values to 0.
	int S = 0, T = 0, i = 0, k = 0;

	// Declare two pointer-to-int type variables.
	int * A, * B;

	// Declare one pointer-to-pointer-to-char type variable.
	char ** C;

	// Declare a file output stream object.
	std::ofstream file;

	/**
	* If the file named arrays_output.txt does not already exist
	* inside of the same file directory as the file named arrays.cpp,
	* create a new file named arrays_output.txt in that directory.
	*
	* Open the plain-text file named arrays_output.txt
	* and set that file to be overwritten with program data.
	*/
	file.open("arrays_output.txt");

	// Print an opening message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nStart Of Program";
	std::cout << "\n--------------------------------";

	// Print an opening message to the file output stream.
	file << "--------------------------------";
	file << "\nStart Of Program";
	file << "\n--------------------------------";

	// Print "The following statements describe the data capacities of various primitive C++ data types:" to the command line terminal.
	std::cout << "\n\nThe following statements describe the data capacities of various primitive C++ data types:";

	// Print "The following statements describe the data capacities of various primitive C++ data types:" to the file output stream.
	file << "\n\nThe following statements describe the data capacities of various primitive C++ data types:";

	// Print the data size of a bool type variable to the command line terminal.
```

std::cout << "\n\nsizeof(bool) = " << sizeof(bool) << ". // number of bytes which a bool type variable occupies";

// Print the data size of a bool type variable to the file output stream.
file << "\n\nsizeof(bool) = " << sizeof(bool) << ". // number of bytes which a bool type variable occupies";

// Print the data size of a char type variable to the command line terminal.
std::cout << "\n\nsizeof(char) = " << sizeof(char) << ". // number of bytes which a char type variable occupies";

// Print the data size of a char type variable to the file output stream.
file << "\n\nsizeof(char) = " << sizeof(char) << ". // number of bytes which a char type variable occupies";

// Print the data size of an int type variable to the command line terminal.
std::cout << "\n\nsizeof(int) = " << sizeof(int) << ". // number of bytes which an int type variable occupies";

// Print the data size of an int type variable to the file output stream.
file << "\n\nsizeof(int) = " << sizeof(int) << ". // number of bytes which an int type variable occupies";

// Print the data size of a long type variable to the command line terminal.
std::cout << "\n\nsizeof(long) = " << sizeof(long) << ". // number of bytes which a long type variable occupies";

// Print the data size of a long type variable to the file output stream.
file << "\n\nsizeof(long) = " << sizeof(long) << ". // number of bytes which a long type variable occupies";

// Print the data size of a float type variable to the command line terminal.
std::cout << "\n\nsizeof(float) = " << sizeof(float) << ". // number of bytes which a float type variable occupies";

// Print the data size of a float type variable to the file output stream.
file << "\n\nsizeof(float) = " << sizeof(float) << ". // number of bytes which a float type variable occupies";

// Print the data size of a double type variable to the command line terminal.
std::cout << "\n\nsizeof(double) = " << sizeof(double) << ". // number of bytes which a double type variable occupies";

// Print the data size of a doudle type variable to the file output stream.

file << "\n\nsizeof(double) = " << sizeof(double) << ". // number of bytes which a double type variable occupies";

// Print the data size of a pointer-to-bool type variable to the command line terminal.
std::cout << "\n\nsizeof(bool *) = " << sizeof(bool *) << ". // number of bytes which a pointer-to-bool type variable occupies";

// Print the data size of a pointer-to-bool type variable to the file output stream.
file << "\n\nsizeof(bool *) = " << sizeof(bool *) << ". // number of bytes which a pointer-to-bool type variable occupies";

// Print the data size of a pointer-to-char type variable to the command line terminal.
std::cout << "\n\nsizeof(char *) = " << sizeof(char *) << ". // number of bytes which a pointer-to-char type variable occupies";

// Print the data size of a pointer-to-char type variable to the file output stream.
file << "\n\nsizeof(char *) = " << sizeof(char *) << ". // number of bytes which a pointer-to-char type variable occupies";

// Print the data size of a pointer-to-int type variable to the command line terminal.
std::cout << "\n\nsizeof(int *) = " << sizeof(int *) << ". // number of bytes which a pointer-to-int type variable occupies";

// Print the data size of a pointer-to-int type variable to the file output stream.
file << "\n\nsizeof(int *) = " << sizeof(int *) << ". // number of bytes which a pointer-to-int type variable occupies";

// Print the data size of a pointer-to-long type variable to the command line terminal.
std::cout << "\n\nsizeof(long *) = " << sizeof(long *) << ". // number of bytes which a pointer-to-long type variable occupies";

// Print the data size of a pointer-to-long type variable to the file output stream.
file << "\n\nsizeof(long *) = " << sizeof(long *) << ". // number of bytes which a pointer-to-long type variable occupies";

// Print the data size of a pointer-to-float type variable to the command line terminal.
std::cout << "\n\nsizeof(float *) = " << sizeof(float *) << ". // number of bytes which a pointer-to-float type variable occupies";

// Print the data size of a pointer-to-float type variable to the file output stream.
file << "\n\nsizeof(float *) = " << sizeof(float *) << ". // number of bytes which a pointer-to-float type variable occupies";

// Print the data size of a pointer-to-double type variable to the command line terminal.

std::cout << "\n\nsizeof(double *) = " << sizeof(double *) << ". // number of bytes which a pointer-to-double type variable occupies";

// Print the data size of a pointer-to-double type variable to the file output stream.
file << "\n\nsizeof(double *) = " << sizeof(double *) << ". // number of bytes which a pointer-to-double type variable occupies";

// Print the data size of a pointer-to-pointer-to-bool type variable to the command line terminal.
std::cout << "\n\nsizeof(bool **) = " << sizeof(bool **) << ". // number of bytes which a pointer-to-pointer-to-bool type variable occupies";

// Print the data size of a pointer-to-pointer-to-bool type variable to the file output stream.
file << "\n\nsizeof(bool **) = " << sizeof(bool **) << ". // number of bytes which a pointer-to-pointer-to-bool type variable occupies";

// Print the data size of a pointer-to-pointer-to-char type variable to the command line terminal.
std::cout << "\n\nsizeof(char **) = " << sizeof(char **) << ". // number of bytes which a pointer-to-pointer-to-char type variable occupies";

// Print the data size of a pointer-to-pointer-to-char type variable to the file output stream.
file << "\n\nsizeof(char **) = " << sizeof(char **) << ". // number of bytes which a pointer-to-pointer-to-char type variable occupies";

// Print the data size of a pointer-to-pointer-to-int type variable to the command line terminal.
std::cout << "\n\nsizeof(int **) = " << sizeof(int **) << ". // number of bytes which a pointer-to-pointer-to-int type variable occupies";

// Print the data size of a pointer-to-pointer-to-int type variable to the file output stream.
file << "\n\nsizeof(int **) = " << sizeof(int **) << ". // number of bytes which a pointer-to-pointer-to-int type variable occupies";

// Print the data size of a pointer-to-pointer-to-long type variable to the command line terminal.
std::cout << "\n\nsizeof(long **) = " << sizeof(long **) << ". // number of bytes which a pointer-to-pointer-to-long type variable occupies";

// Print the data size of a pointer-to-pointer-to-long type variable to the file output stream.
file << "\n\nsizeof(long **) = " << sizeof(long **) << ". // number of bytes which a pointer-to-pointer-to-long type variable occupies";

// Print the data size of a pointer-to-pointer-to-float type variable to the command line terminal.
std::cout << "\n\nsizeof(float **) = " << sizeof(float **) << ". // number of bytes which a pointer-to-pointer-to-float type variable occupies";

// Print the data size of a pointer-to-pointer-to-float type variable to the file output stream.
file << "\n\nsizeof(float **) = " << sizeof(float **) << ". // number of bytes which a pointer-to-pointer-to-float type variable occupies";

// Print the data size of a pointer-to-pointer-to-double type variable to the command line terminal.
std::cout << "\n\nsizeof(double **) = " << sizeof(double **) << ". // number of bytes which a pointer-to-pointer-to-double type variable occupies";

// Print the data size of a pointer-to-pointer-to-double type variable to the file output stream.
file << "\n\nsizeof(double **) = " << sizeof(double **) << ". // number of bytes which a pointer-to-pointer-to-double type variable occupies";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Print "STEP_0: CREATE A DYNAMIC ARRAY WHICH IS NAMED A AND WHICH IS COMPRISED OF S INT TYPE VALUES." to the command line terminal.
std::cout << "\n\nSTEP_0: CREATE A DYNAMIC ARRAY WHICH IS NAMED A AND WHICH IS COMPRISED OF S INT TYPE VALUES.";

// Print "STEP_0: CREATE A DYNAMIC ARRAY WHICH IS NAMED A AND WHICH IS COMRPISED OF S INT TYPE VALUES." to the file output stream.
file << "\n\nSTEP_0: CREATE A DYNAMIC ARRAY WHICH IS NAMED A AND WHICH IS COMPRISED OF S INT TYPE VALUES.";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Print "Enter a natural number, S, for representing the number of elements to include in an array which is no larger than than {MAXIMUM_S}: " to the command line terminal.

std::cout << "\n\nEnter a natural number, S, for representing the number of elements to include in an array which is no larger than " << MAXIMUM_S <> S;

// Print "The value which was entered for S is {S}." to the command line terminal.
std::cout << "\nThe value which was entered for S is " << S << ".";

// Print "The value which was entered for S is {S}." to the file output stream.
file << "\n\nThe value which was entered for S is " << S << ".";

// If S is smaller than 1 or if S is larger than MAXIMUM_S, set S to 10.
S = ((S MAXIMUM_S)) ? 10 : S;

// Print "S := {S}. // number of consecutive int-sized chunks of memory to allocate to an array such that the memory address of the first element of that array, A[0], is stored in a pointer-to-int type variable named A" to the command line terminal.
std::cout << "\n\nS := " << S << ". // number of consecutive int-sized chunks of memory to allocate to an array such that the memory address of the first element of that array, A[0], is stored in a pointer-to-int type variable named A";

// Print "S := {S}. // number of consecutive int-sized chunks of memory to allocate to an array such that the memory address of the first element of that array, A[0], is stored in a pointer-to-int type variable named A" to the file output stream.
file << "\n\nS := " << S << ". // number of consecutive int-sized chunks of memory to allocate to an array such that the memory address of the first element of that array, A[0], is stored in a pointer-to-int type variable named A";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Allocate S contiguous int-sized chunks of memory and store the memory address of the first int-sized chunk of memory, A[0]. inside the pointer-to-int type variable named A.
A = new int [S];

// Print the program instruction used to generate the dynamic array represented by A to the command line terminal.
std::cout << "\n\n// Declare a pointer-to-int type variable named A.";
std::cout << "\nint * A;";
std::cout << "\n\n// Allocate S contiguous int-sized chunks of memory and store the memory address of the first int-sized chunk of memory, A[0], inside the pointer-to-int type variable named A.";
std::cout << "\nA = new int [S];";

```
	// Print the program instruction used to generate the dynamic array represented by A to the file output stream.
	file << "\n\n// Declare a pointer-to-int type variable named A.";
	file << "\nint * A;";
	file << "\n\n// Allocate S contiguous int-sized chunks of memory and store the memory address of the first int-sized chunk of memory, A[0], inside the pointer-to-int type variable named A.";
	file << "\nA = new int [S];";

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print the contents of A to the command line terminal.
	std::cout << "\n\nA = " << A << ". // memory address of A[0]\n";

	// Print the contents of A to the file output stream.
	file << "\n\nA = " << A << ". // memory address of A[0]\n";

	/**
	* For each element, i, of the array represented by A,
	* print the contents of the ith element of the array, A[i],
	* and the memory address of that array element
	* to the command line terminal and to the file output stream.
	*/
	for (i = 0; i < S; i += 1)
	{
	std::cout << "\nA[" << i << "] = " << A[i] << ". \t\t// &A[" << i << "] = " << &A[i] << ". (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[" << i << "]).";
	file << "\nA[" << i << "] = " << A[i] << ". \t\t// &A[" << i << "] = " << &A[i] << ". (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[" << i << "]).";
	}

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";
```

// Print "STEP_1: RANDOMLY ASSIGN ONE OF THE FIRST T RANDOM NONNEGATIVE INTEGERS TO EACH ELEMENT OF THE ARRAY NAMED A." to the command line terminal.
std::cout << "\n\nSTEP_1: RANDOMLY ASSIGN ONE OF THE FIRST T RANDOM NONNEGATIVE INTEGERS TO EACH ELEMENT OF THE ARRAY NAMED A.";

// Print "STEP_1: RANDOMLY ASSIGN ONE OF THE FIRST T RANDOM NONNEGATIVE INTEGERS TO EACH ELEMENT OF THE ARRAY NAMED A." to the file output stream.
file << "\n\nSTEP_1: RANDOMLY ASSIGN ONE OF THE FIRST T RANDOM NONNEGATIVE INTEGERS TO EACH ELEMENT OF THE ARRAY NAMED A.";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Print "Enter a natural number, T, for representing the number of unique states which each element of the array can store exactly one of which is no larger than {MAXIMUM_T}: " to the command line terminal.
std::cout << "\n\nEnter a natural number, T, for representing the number of unique states which each element of the array can store exactly one of which is no larger than " << MAXIMUM_T <> T;

// Print "The value which was entered for T is {T}." to the command line terminal.
std::cout << "\nThe value which was entered for T is " << T << ".";

// Print "The value which was entered for T is {T}." to the file output stream.
file << "\n\nThe value which was entered for T is " << T << ".";

// If T is smaller than 1 or if T is larger than MAXIMUM_T, set T to 100.
T = ((T MAXIMUM_T)) ? 100 : T;

// Print "T := {T}. // number of unique states which each element of array A can represent" to the command line terminal.
std::cout << "\n\nT := " << T << ". // number of unique states which each element of array A can represent";

// Print "T := {T}. // number of unique states which each element of array A can represent" to the file output stream.
file << "\n\nT := " << T << ". // number of unique states which each element of array A can represent";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Seed the pseudo random number generator with the integer number of seconds which have elapsed since the Unix Epoch (i.e. midnight of 01_JANUARY_1970).
srand(time(NULL));

// Print the command to seed the pseudo random number generator to the command line.
std::cout << "\n\n// Seed the pseudo random number generator with the integer number of seconds which have elapsed since the Unix Epoch (i.e. midnight of 01_JANUARY_1970).";
std::cout << "\nsrand(time(NULL));";

// Print the command to seed the pseudo random number generator to the file output stream.
file << "\n\n// Seed the pseudo random number generator with the integer number of seconds which have elapsed since the Unix Epoch (i.e. midnight of 01_JANUARY_1970).";
file << "\nsrand(time(NULL));";

// For each element, A[i], of the array named A, set A[i] to a randomly generated integer which is no smaller than 0 and no larger than (T - 1).
for (i = 0; i < S; i += 1) A[i] = rand() % T;

// Print the command to populate each element of the array named A with a randomly generated integer which is no smaller than 0 and no larger than (T - 1) to the command line terminal.
std::cout << "\n\n// For each element, A[i], of the array named A, set A[i] to a randomly generated integer which is no smaller than 0 and no larger than (T - 1).";
std::cout << "\nfor (i = 0; i < S; i += 1) A[i] = rand() % T;";

// Print the command to populate each element of the array named A with a randomly generated integer which is no smaller than 0 and no larger than (T - 1) to the file output stream.
file << "\n\n// For each element, A[i], of the array named A, set A[i] to a randomly generated integer which is no smaller than 0 and no larger than (T - 1).";
file << "\nfor (i = 0; i < S; i += 1) A[i] = rand() % T;";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

```
// Print the contents of A to the command line terminal.
std::cout << "\n\nA = " << A << ". // memory address of A[0]\n";

// Print the contents of A to the file output stream.
file << "\n\nA = " << A << ". // memory address of A[0]\n";

/**
* For each element, i, of the array represented by A,
* print the contents of the ith element of the array, A[i],
* and the memory address of that array element
* to the command line terminal and to the file output stream.
*/
for (i = 0; i < S; i += 1)
{
std::cout << "\nA[" << i << "] = " << A[i] << ". \t\t// &A[" << i << "] = " << &A[i] << ". (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[" << i << "]).";
file << "\nA[" << i << "] = " << A[i] << ". \t\t// &A[" << i << "] = " << &A[i] << ". (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[" << i << "]).";
}

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Print "STEP_2: SORT THE ELEMENT VALUES OF THE ARRAY NAMED A TO BE IN ASCENDING ORDER." to the command line terminal.
std::cout << "\n\nSTEP_2: SORT THE ELEMENT VALUES OF THE ARRAY NAMED A TO BE IN ASCENDING ORDER.";

// Print "STEP_2: SORT THE ELEMENT VALUES OF THE ARRAY NAMED A TO BE IN ASCENDING ORDER." to the file output stream.
file << "\n\nSTEP_2: SORT THE ELEMENT VALUES OF THE ARRAY NAMED A TO BE IN ASCENDING ORDER.";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";
```

```
	// Sort the integer values stored in array A to be in ascending order using the Bubble Sort algorithm.
	bubble_sort(A, S);

	// Print the command to sort the integer values stored in array A in ascending order to the command line.
	std::cout << "\n\n// Sort the integer values stored in array A to be in ascending order using the Bubble Sort algorithm.";
	std::cout << "\nbubble_sort(A, S);";

	// Print the command to sort the integer values stored in array A in ascending order to the file output stream.
	file << "\n\n// Sort the integer values stored in array A to be in ascending order using the Bubble Sort algorithm.";
	file << "\nbubble_sort(A, S);";

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print the contents of A to the command line terminal.
	std::cout << "\n\nA = " << A << ". // memory address of A[0]\n";

	// Print the contents of A to the file output stream.
	file << "\n\nA = " << A << ". // memory address of A[0]\n";

	/**
	* For each element, i, of the array represented by A,
	* print the contents of the ith element of the array, A[i],
	* and the memory address of that array element
	* to the command line terminal and to the file output stream.
	*/
	for (i = 0; i < S; i += 1)
	{
	std::cout << "\nA[" << i << "] = " << A[i] << ".\t\t// &A[" << i << "] = " << &A[i] << ". (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[" << i << "]).";
	file << "\nA[" << i << "] = " << A[i] << ".\t\t// &A[" << i << "] = " << &A[i] << ". (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[" << i << "]).";
	}
```

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Print "STEP_3: CREATE A DYNAMIC ARRAY WHICH IS NAMED B AND WHICH IS COMPRISED OF T INT TYPE VALUES." to the command line terminal.
std::cout << "\n\nSTEP_3: CREATE A DYNAMIC ARRAY WHICH IS NAMED B AND WHICH IS COMPRISED OF T INT TYPE VALUES.";

// Print "STEP_3: CREATE A DYNAMIC ARRAY WHICH IS NAMED B AND WHICH IS COMPRISED OF T INT TYPE VALUES." to the file output stream.
file << "\n\nSTEP_3: CREATE A DYNAMIC ARRAY WHICH IS NAMED B AND WHICH IS COMPRISED OF T INT TYPE VALUES.";

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Allocate T contiguous int-sized chunks of memory and store the memory address of the first int-sized chunk of memory, B[0]. inside the pointer-to-int type variable named B.
B = new int [T];

// Print the program instruction used to generate the dynamic array represented by B to the command line terminal.
std::cout << "\n\n// Declare a pointer-to-int type variable named B.";
std::cout << "\nint * B;";
std::cout << "\n\n// Allocate T contiguous int-sized chunks of memory and store the memory address of the first int-sized chunk of memory, B[0], inside the pointer-to-int type variable named B.";
std::cout << "\nB = new int [T];";

// Print the program instruction used to generate the dynamic array represented by B to the file output stream.
file << "\n\n// Declare a pointer-to-int type variable named B.";
file << "\nint * B;";
file << "\n\n// Allocate T contiguous int-sized chunks of memory and store the memory address of the first int-sized chunk of memory, B[0], inside the pointer-to-int type variable named B.";
file << "\nB = new int [T];";

```
// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Print the contents of B to the command line terminal.
std::cout << "\n\nB = " << B << ". // memory address of B[0]\n";

// Print the contents of B to the file output stream.
file << "\n\nB = " << B << ". // memory address of B[0]\n";

/**
* For each element, i, of the array represented by B,
* print the contents of the ith element of the array, B[i],
* and the memory address of that array element
* to the command line terminal and to the file output stream.
*/
for (i = 0; i < T; i += 1)
{
std::cout << "\nB[" << i << "] = " << B[i] << ".\t\t// &B[" << i << "] = " << &B[i] << ". (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[" << i << "]).";
file << "\nB[" << i << "] = " << B[i] << ".\t\t// &B[" << i << "] = " << &B[i] << ". (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[" << i << "]).";
}

// Print a horizontal line to the command line terminal.
std::cout << "\n\n--------------------------------";

// Print a horizontal line to the command line terminal.
file << "\n\n--------------------------------";

// Print "STEP_4: FOR EACH ELEMENT B[i] OF THE ARRAY NAMED B, STORE THE NUMBER OF TIMES i APPEARS AS AN ELEMENT VALUE IN THE ARRAY NAMED A." to the command line terminal.
std::cout << "\n\nSTEP_4: FOR EACH ELEMENT B[i] OF THE ARRAY NAMED B, STORE THE NUMBER OF TIMES i APPEARS AS AN ELEMENT VALUE IN THE ARRAY NAMED A.";
```

```
	// Print "STEP_4: FOR EACH ELEMENT B[i] OF THE ARRAY NAMED B, STORE THE NUMBER OF TIMES i APPEARS AS AN ELEMENT VALUE IN THE ARRAY NAMED A." to the file output stream.
	file << "\n\nSTEP_4: FOR EACH ELEMENT B[i] OF THE ARRAY NAMED B, STORE THE NUMBER OF TIMES i APPEARS AS AN ELEMENT VALUE IN THE ARRAY NAMED A.";

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	/**
	* For each element, i, of the array represented by B,
	* store the number of times i appears as an element value in the array represented by A
	* in B[i].
	*/
	for (i = 0; i < T; i += 1)
	{
	for (k = 0; k < S; k += 1)
	{
	if (i == A[k]) B[i] += 1;
	}
	}

	// Print the contents of B to the command line terminal.
	std::cout << "\n\nB = " << B << ". // memory address of B[0]\n";

	// Print the contents of B to the file output stream.
	file << "\n\nB = " << B << ". // memory address of B[0]\n";

	/**
	* For each element, i, of the array represented by B,
	* print the contents of the ith element of the array, B[i],
	* and the memory address of that array element
	* to the command line terminal and to the file output stream.
	*/
	for (i = 0; i < T; i += 1)
	{
	std::cout << "\nB[" << i << "] = " << B[i] << ".\t\t// &B[" << i << "] = " << &B[i] << ". (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[" << i << "]).";
```

```
	file << "\nB[" << i << "] = " << B[i] << ".\t\t// &B[" << i << "] = " << &B[i] << ". (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[" << i << "]).";
	}

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print "STEP_5: CREATE A DYNAMIC ARRAY WHICH IS NAMED C AND WHICH IS COMPRISED OF T POINTER-TO-CHAR TYPE VALUES." to the command line terminal.
	std::cout << "\n\nSTEP_5: CREATE A DYNAMIC ARRAY WHICH IS NAMED C AND WHICH IS COMPRISED OF T POINTER-TO-CHAR TYPE VALUES.";

	// Print "STEP_5: CREATE A DYNAMIC ARRAY WHICH IS NAMED C AND WHICH IS COMPRISED OF T POINTER-TO-CHAR TYPE VALUES." to the file output stream.
	file << "\n\nSTEP_5: CREATE A DYNAMIC ARRAY WHICH IS NAMED C AND WHICH IS COMPRISED OF T POINTER-TO-CHAR TYPE VALUES.";

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Allocate T contiguous pointer-to-char-sized chunks of memory and store the memory address of the first pointer-to-char-sized chunk of memory, C[0], inside the pointer-to-pointer-to-char type variable named C.
	C = new char * [T];

	// C is a two-dimensional array which depicts a histogram (i.e. bar graph) such the length of the ith row is identical to the value stored in B[i].
	for (i = 0; i < T; i += 1)
	{
	C[i] = new char [B[i]];
	for (k = 0; k < B[i]; k += 1) C[i][k] = 'X';
	}

	// Print the program instruction used to generate the dynamic array represented by C to the command line terminal.
	std::cout << "\n\n// Declare one pointer-to-pointer-to-char type variable.";
	std::cout << "\nchar ** C;";
```

```
	std::cout << "\n\n// Allocate T contiguous pointer-to-char-sized chunks of memory and store the memory address of the first pointer-to-char-sized chunk of memory, C[0], inside the pointer-to-pointer-to-char type variable named C.";
	std::cout << "\nC = new char * [T];";
	std::cout << "\n\n// C is a two-dimensional array which depicts a histogram (i.e. bar graph) such the length of the ith row is identical to the value stored in B[i].";
	std::cout << "\nfor (i = 0; i < T; i += 1)";
	std::cout << "\n{";
	std::cout << "\n	C[i] = new char [B[i]];";
	std::cout << "\n	for (k = 0; k < B[i]; k += 1) C[i][k] = 'X';";
	std::cout << "\n}";

	// Print the program instruction used to generate the dynamic array represented by C to the file output stream.
	file << "\n\n// Declare one pointer-to-pointer-to-char type variable.";
	file << "\nchar ** C;";
	file << "\n\n// Allocate T contiguous pointer-to-char-sized chunks of memory and store the memory address of the first pointer-to-char-sized chunk of memory, C[0], inside the pointer-to-pointer-to-char type variable named C.";
	file << "\nC = new char * [T];";
	file << "\n\n// C is a two-dimensional array which depicts a histogram (i.e. bar graph) such the length of the ith row is identical to the value stored in B[i].";
	file << "\nfor (i = 0; i < T; i += 1)";
	file << "\n{";
	file << "\n	C[i] = new char [B[i]];";
	file << "\n	for (k = 0; k < B[i]; k += 1) C[i][k] = 'X';";
	file << "\n}";

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print the contents of C to the command line terminal.
	std::cout << "\n\nC = " << C << ". // memory address of C[0]\n";

	// Print the contents of C to the file output stream.
	file << "\n\nC = " << C << ". // memory address of C[0]\n";

	/**
	* For each element, i, of the array represented by C,
	* print the contents of the ith element of the array, C[i],
	* and the memory address of that array element
```

```
	* to the command line terminal and to the file output stream.
	*/
	for (i = 0; i < T; i += 1)
	{
	std::cout << "\nC[" << i << "] = " << C[i] << ".\t\t// &C[" << i << "] = " << &C[i] << ". (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[" << i << "]).";
	file << "\nC[" << i << "] = " << C[i] << ".\t\t// &C[" << i << "] = " << &C[i] << ". (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[" << i << "]).";
	}

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	// Print "STEP_6: RELEASE MEMORY WHICH WAS ALLOCATED TO THE DYNAMIC ARRAYS NAMED A, B, AND C." to the command line terminal.
	std::cout << "\n\nSTEP_6: RELEASE MEMORY WHICH WAS ALLOCATED TO THE DYNAMIC ARRAYS NAMED A, B, AND C.";

	// Print "STEP_6: RELEASE MEMORY WHICH WAS ALLOCATED TO THE DYNAMIC ARRAYS NAMED A, B, AND C." to the file output stream.
	file << "\n\nSTEP_6: RELEASE MEMORY WHICH WAS ALLOCATED TO THE DYNAMIC ARRAYS NAMED A, B, AND C.";

	// Print a horizontal line to the command line terminal.
	std::cout << "\n\n--------------------------------";

	// Print a horizontal line to the command line terminal.
	file << "\n\n--------------------------------";

	/**
	* Note that, unlike a static array, a dynamic array is instantiated during program runtime instead of during program compile time.
	* (A static array is assigned memory during program compilation while a dynamic array is assigned memory during program runtime).
	* At compile time, the computer does not know how much memory space to allocate to a dynamic array because the number of elements
	* in that array may vary and is not specified in the program source code.
	*/
```

// De-allocate memory which was assigned to the dynamically-allocated array of S int type values
delete [] A;

// Print the command to de-allocate memory which was assigned to the dynamically-allocated array of S int type values to the command line terminal.
std::cout << "\n\n// De-allocate memory which was assigned to the dynamically-allocated array of S int type values.";
std::cout << "\ndelete [] A; // Free up S contiguous int-sized chunks of memory which were assigned to the dynamic array named A.";

// Print the command to de-allocate memory which was assigned to the dynamically-allocated array of S int type values to the file output stream.
file << "\n\n// De-allocate memory which was assigned to the dynamically-allocated array of S int type values.";
file << "\ndelete [] A; // Free up S contiguous int-sized chunks of memory which were assigned to the dynamic array named A.";

// De-allocate memory which was assigned to the dynamically-allocated array of T int type values.
delete [] B;

// Print the command to de-allocate memory which was assigned to the dynamically-allocated array of T int type values to the command line terminal.
std::cout << "\n\nDe-allocate memory which was assigned to the dynamically-allocated array of T int type values.";
std::cout << "\ndelete [] B; // Free up T contiguous int-sized chunks of memory which were assigned to the dynamic array named B.";

// Print the command to de-allocate memory which was assigned to the dynamically-allocated array of T int type values to the file output stream.
file << "\n\nDe-allocate memory which was assigned to the dynamically-allocated array of T int type values.";
file << "\ndelete [] B; // Free up T contiguous int-sized chunks of memory which were assigned to the dynamic array named B.";

// De-allocate memory which was assigned to the dynamically-allocated array of T pointer-to-char type values.
for (i = 0; i < T; i += 1) delete [] C[i];
delete [] C;

// Print the command to de-allocate memory which was assigned to the dynamically-allocated array of T pointer-to-char type values to the command line terminal.

```
	std::cout << "\n\n// De-allocate memory which was assigned to the dynamically-allocated array of T pointer-to-char type values.";
	std::cout << "\nfor (i = 0; i < T; i += 1) delete [] C[i]; // Free up B[i] char-sized chunks of memory which were assigned to the dynamic array named C[i].";
	std::cout << "\ndelete [] C; // Free up T contiguous pointer-to-char-sized chunks of memory which were assigned to the dynamic array named C.";

	// Print the command to de-allocate memory which was assigned to the dynamically-allocated array of T pointer-to-char type values to the file output stream.
	file << "\n\n// De-allocate memory which was assigned to the dynamically-allocated array of T pointer-to-char type values.";
	file << "\nfor (i = 0; i < T; i += 1) delete [] C[i]; // Free up B[i] char-sized chunks of memory which were assigned to the dynamic array named C[i].";
	file << "\ndelete [] C; // Free up T contiguous pointer-to-char-sized chunks of memory which were assigned to the dynamic array named C.";

	// Print a closing message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nEnd Of Program";
	std::cout << "\n--------------------------------\n\n";

	// Print a closing message to the file output stream.
	file << "\n\n--------------------------------";
	file << "\nEnd Of Program";
	file << "\n--------------------------------";

	// Close the file output stream.
	file.close();

	// Exit the program.
	return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/arrays_output.txt

----
Start Of Program

The following statements describe the data capacities of various primitive C++ data types:

sizeof(bool) = 1. // number of bytes which a bool type variable occupies

sizeof(char) = 1. // number of bytes which a char type variable occupies

sizeof(int) = 4. // number of bytes which an int type variable occupies

sizeof(long) = 8. // number of bytes which a long type variable occupies

sizeof(float) = 4. // number of bytes which a float type variable occupies

sizeof(double) = 8. // number of bytes which a double type variable occupies

sizeof(bool *) = 8. // number of bytes which a pointer-to-bool type variable occupies

sizeof(char *) = 8. // number of bytes which a pointer-to-char type variable occupies

sizeof(int *) = 8. // number of bytes which a pointer-to-int type variable occupies

sizeof(long *) = 8. // number of bytes which a pointer-to-long type variable occupies

sizeof(float *) = 8. // number of bytes which a pointer-to-float type variable occupies

sizeof(double *) = 8. // number of bytes which a pointer-to-double type variable occupies

sizeof(bool **) = 8. // number of bytes which a pointer-to-pointer-to-bool type variable occupies

sizeof(char **) = 8. // number of bytes which a pointer-to-pointer-to-char type variable occupies

sizeof(int **) = 8. // number of bytes which a pointer-to-pointer-to-int type variable occupies

sizeof(long **) = 8. // number of bytes which a pointer-to-pointer-to-long type variable occupies

sizeof(float **) = 8. // number of bytes which a pointer-to-pointer-to-float type variable occupies

sizeof(double **) = 8. // number of bytes which a pointer-to-pointer-to-double type variable occupies

----

STEP_0: CREATE A DYNAMIC ARRAY WHICH IS NAMED A AND WHICH IS COMPRISED OF S INT TYPE VALUES.

The value which was entered for S is 100.

S := 100. // number of consecutive int-sized chunks of memory to allocate to an array such that the memory address of the first element of that array, A[0], is stored in a pointer-to-int type variable named A

// Declare a pointer-to-int type variable named A.
int * A;

// Allocate S contiguous int-sized chunks of memory and store the memory address of the first int-sized chunk of memory, A[0], inside the pointer-to-int type variable named A.
A = new int [S];

A = 0x5607d37868c0. // memory address of A[0]

A[0] = 0. // &A[0] = 0x5607d37868c0. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[0]).
A[1] = 0. // &A[1] = 0x5607d37868c4. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[1]).
A[2] = 0. // &A[2] = 0x5607d37868c8. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[2]).
A[3] = 0. // &A[3] = 0x5607d37868cc. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[3]).
A[4] = 0. // &A[4] = 0x5607d37868d0. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[4]).
A[5] = 0. // &A[5] = 0x5607d37868d4. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[5]).
A[6] = 0. // &A[6] = 0x5607d37868d8. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[6]).
A[7] = 0. // &A[7] = 0x5607d37868dc. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[7]).

A[8] = 0. // &A[8] = 0x5607d37868e0. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[8]).
A[9] = 0. // &A[9] = 0x5607d37868e4. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[9]).
A[10] = 0. // &A[10] = 0x5607d37868e8. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[10]).
A[11] = 0. // &A[11] = 0x5607d37868ec. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[11]).
A[12] = 0. // &A[12] = 0x5607d37868f0. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[12]).
A[13] = 0. // &A[13] = 0x5607d37868f4. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[13]).
A[14] = 0. // &A[14] = 0x5607d37868f8. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[14]).
A[15] = 0. // &A[15] = 0x5607d37868fc. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[15]).
A[16] = 0. // &A[16] = 0x5607d3786900. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[16]).
A[17] = 0. // &A[17] = 0x5607d3786904. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[17]).
A[18] = 0. // &A[18] = 0x5607d3786908. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[18]).
A[19] = 0. // &A[19] = 0x5607d378690c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[19]).
A[20] = 0. // &A[20] = 0x5607d3786910. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[20]).
A[21] = 0. // &A[21] = 0x5607d3786914. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[21]).
A[22] = 0. // &A[22] = 0x5607d3786918. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[22]).
A[23] = 0. // &A[23] = 0x5607d378691c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[23]).
A[24] = 0. // &A[24] = 0x5607d3786920. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[24]).
A[25] = 0. // &A[25] = 0x5607d3786924. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[25]).
A[26] = 0. // &A[26] = 0x5607d3786928. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[26]).
A[27] = 0. // &A[27] = 0x5607d378692c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[27]).
A[28] = 0. // &A[28] = 0x5607d3786930. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[28]).
A[29] = 0. // &A[29] = 0x5607d3786934. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[29]).

A[30] = 0. // &A[30] = 0x5607d3786938. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[30]).
A[31] = 0. // &A[31] = 0x5607d378693c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[31]).
A[32] = 0. // &A[32] = 0x5607d3786940. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[32]).
A[33] = 0. // &A[33] = 0x5607d3786944. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[33]).
A[34] = 0. // &A[34] = 0x5607d3786948. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[34]).
A[35] = 0. // &A[35] = 0x5607d378694c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[35]).
A[36] = 0. // &A[36] = 0x5607d3786950. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[36]).
A[37] = 0. // &A[37] = 0x5607d3786954. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[37]).
A[38] = 0. // &A[38] = 0x5607d3786958. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[38]).
A[39] = 0. // &A[39] = 0x5607d378695c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[39]).
A[40] = 0. // &A[40] = 0x5607d3786960. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[40]).
A[41] = 0. // &A[41] = 0x5607d3786964. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[41]).
A[42] = 0. // &A[42] = 0x5607d3786968. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[42]).
A[43] = 0. // &A[43] = 0x5607d378696c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[43]).
A[44] = 0. // &A[44] = 0x5607d3786970. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[44]).
A[45] = 0. // &A[45] = 0x5607d3786974. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[45]).
A[46] = 0. // &A[46] = 0x5607d3786978. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[46]).
A[47] = 0. // &A[47] = 0x5607d378697c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[47]).
A[48] = 0. // &A[48] = 0x5607d3786980. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[48]).
A[49] = 0. // &A[49] = 0x5607d3786984. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[49]).
A[50] = 0. // &A[50] = 0x5607d3786988. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[50]).
A[51] = 0. // &A[51] = 0x5607d378698c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[51]).

A[52] = 0.   // &A[52] = 0x5607d3786990. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[52]).
A[53] = 0.   // &A[53] = 0x5607d3786994. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[53]).
A[54] = 0.   // &A[54] = 0x5607d3786998. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[54]).
A[55] = 0.   // &A[55] = 0x5607d378699c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[55]).
A[56] = 0.   // &A[56] = 0x5607d37869a0. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[56]).
A[57] = 0.   // &A[57] = 0x5607d37869a4. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[57]).
A[58] = 0.   // &A[58] = 0x5607d37869a8. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[58]).
A[59] = 0.   // &A[59] = 0x5607d37869ac. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[59]).
A[60] = 0.   // &A[60] = 0x5607d37869b0. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[60]).
A[61] = 0.   // &A[61] = 0x5607d37869b4. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[61]).
A[62] = 0.   // &A[62] = 0x5607d37869b8. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[62]).
A[63] = 0.   // &A[63] = 0x5607d37869bc. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[63]).
A[64] = 0.   // &A[64] = 0x5607d37869c0. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[64]).
A[65] = 0.   // &A[65] = 0x5607d37869c4. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[65]).
A[66] = 0.   // &A[66] = 0x5607d37869c8. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[66]).
A[67] = 0.   // &A[67] = 0x5607d37869cc. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[67]).
A[68] = 0.   // &A[68] = 0x5607d37869d0. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[68]).
A[69] = 0.   // &A[69] = 0x5607d37869d4. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[69]).
A[70] = 0.   // &A[70] = 0x5607d37869d8. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[70]).
A[71] = 0.   // &A[71] = 0x5607d37869dc. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[71]).
A[72] = 0.   // &A[72] = 0x5607d37869e0. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[72]).
A[73] = 0.   // &A[73] = 0x5607d37869e4. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[73]).

A[74] = 0. // &A[74] = 0x5607d37869e8. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[74]).
A[75] = 0. // &A[75] = 0x5607d37869ec. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[75]).
A[76] = 0. // &A[76] = 0x5607d37869f0. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[76]).
A[77] = 0. // &A[77] = 0x5607d37869f4. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[77]).
A[78] = 0. // &A[78] = 0x5607d37869f8. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[78]).
A[79] = 0. // &A[79] = 0x5607d37869fc. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[79]).
A[80] = 0. // &A[80] = 0x5607d3786a00. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[80]).
A[81] = 0. // &A[81] = 0x5607d3786a04. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[81]).
A[82] = 0. // &A[82] = 0x5607d3786a08. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[82]).
A[83] = 0. // &A[83] = 0x5607d3786a0c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[83]).
A[84] = 0. // &A[84] = 0x5607d3786a10. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[84]).
A[85] = 0. // &A[85] = 0x5607d3786a14. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[85]).
A[86] = 0. // &A[86] = 0x5607d3786a18. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[86]).
A[87] = 0. // &A[87] = 0x5607d3786a1c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[87]).
A[88] = 0. // &A[88] = 0x5607d3786a20. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[88]).
A[89] = 0. // &A[89] = 0x5607d3786a24. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[89]).
A[90] = 0. // &A[90] = 0x5607d3786a28. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[90]).
A[91] = 0. // &A[91] = 0x5607d3786a2c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[91]).
A[92] = 0. // &A[92] = 0x5607d3786a30. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[92]).
A[93] = 0. // &A[93] = 0x5607d3786a34. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[93]).
A[94] = 0. // &A[94] = 0x5607d3786a38. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[94]).
A[95] = 0. // &A[95] = 0x5607d3786a3c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[95]).

A[96] = 0. // &A[96] = 0x5607d3786a40. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[96]).
A[97] = 0. // &A[97] = 0x5607d3786a44. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[97]).
A[98] = 0. // &A[98] = 0x5607d3786a48. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[98]).
A[99] = 0. // &A[99] = 0x5607d3786a4c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to A[99]).

STEP_1: RANDOMLY ASSIGN ONE OF THE FIRST T RANDOM NONNEGATIVE INTEGERS TO EACH ELEMENT OF THE ARRAY NAMED A.

The value which was entered for T is 10.

T := 10. // number of unique states which each element of array A can represent

----

// Seed the pseudo random number generator with the integer number of seconds which have elapsed since the Unix Epoch (i.e. midnight of 01_JANUARY_1970).
srand(time(NULL));

// For each element, A[i], of the array named A, set A[i] to a randomly generated integer which is no smaller than 0 and no larger than (T - 1).
for (i = 0; i < S; i += 1) A[i] = rand() % T;

A = 0x5607d37868c0. // memory address of A[0]

A[0] = 1. // &A[0] = 0x5607d37868c0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[0]).
A[1] = 6. // &A[1] = 0x5607d37868c4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[1]).
A[2] = 2. // &A[2] = 0x5607d37868c8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[2]).
A[3] = 8. // &A[3] = 0x5607d37868cc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[3]).
A[4] = 8. // &A[4] = 0x5607d37868d0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[4]).

A[5] = 3. // &A[5] = 0x5607d37868d4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[5]).
A[6] = 1. // &A[6] = 0x5607d37868d8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[6]).
A[7] = 8. // &A[7] = 0x5607d37868dc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[7]).
A[8] = 3. // &A[8] = 0x5607d37868e0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[8]).
A[9] = 3. // &A[9] = 0x5607d37868e4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[9]).
A[10] = 4. // &A[10] = 0x5607d37868e8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[10]).
A[11] = 7. // &A[11] = 0x5607d37868ec. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[11]).
A[12] = 4. // &A[12] = 0x5607d37868f0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[12]).
A[13] = 7. // &A[13] = 0x5607d37868f4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[13]).
A[14] = 0. // &A[14] = 0x5607d37868f8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[14]).
A[15] = 1. // &A[15] = 0x5607d37868fc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[15]).
A[16] = 8. // &A[16] = 0x5607d3786900. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[16]).
A[17] = 7. // &A[17] = 0x5607d3786904. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[17]).
A[18] = 2. // &A[18] = 0x5607d3786908. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[18]).
A[19] = 5. // &A[19] = 0x5607d378690c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[19]).
A[20] = 7. // &A[20] = 0x5607d3786910. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[20]).
A[21] = 5. // &A[21] = 0x5607d3786914. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[21]).
A[22] = 9. // &A[22] = 0x5607d3786918. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[22]).
A[23] = 1. // &A[23] = 0x5607d378691c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[23]).
A[24] = 8. // &A[24] = 0x5607d3786920. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[24]).
A[25] = 0. // &A[25] = 0x5607d3786924. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[25]).
A[26] = 0. // &A[26] = 0x5607d3786928. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[26]).

A[27] = 9. // &A[27] = 0x5607d378692c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[27]).
A[28] = 1. // &A[28] = 0x5607d3786930. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[28]).
A[29] = 2. // &A[29] = 0x5607d3786934. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[29]).
A[30] = 9. // &A[30] = 0x5607d3786938. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[30]).
A[31] = 2. // &A[31] = 0x5607d378693c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[31]).
A[32] = 8. // &A[32] = 0x5607d3786940. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[32]).
A[33] = 3. // &A[33] = 0x5607d3786944. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[33]).
A[34] = 0. // &A[34] = 0x5607d3786948. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[34]).
A[35] = 8. // &A[35] = 0x5607d378694c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[35]).
A[36] = 7. // &A[36] = 0x5607d3786950. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[36]).
A[37] = 1. // &A[37] = 0x5607d3786954. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[37]).
A[38] = 7. // &A[38] = 0x5607d3786958. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[38]).
A[39] = 2. // &A[39] = 0x5607d378695c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[39]).
A[40] = 7. // &A[40] = 0x5607d3786960. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[40]).
A[41] = 1. // &A[41] = 0x5607d3786964. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[41]).
A[42] = 1. // &A[42] = 0x5607d3786968. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[42]).
A[43] = 1. // &A[43] = 0x5607d378696c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[43]).
A[44] = 0. // &A[44] = 0x5607d3786970. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[44]).
A[45] = 4. // &A[45] = 0x5607d3786974. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[45]).
A[46] = 4. // &A[46] = 0x5607d3786978. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[46]).
A[47] = 0. // &A[47] = 0x5607d378697c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[47]).
A[48] = 1. // &A[48] = 0x5607d3786980. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[48]).

A[49] = 6.	// &A[49] = 0x5607d3786984. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[49]).
A[50] = 5.	// &A[50] = 0x5607d3786988. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[50]).
A[51] = 0.	// &A[51] = 0x5607d378698c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[51]).
A[52] = 4.	// &A[52] = 0x5607d3786990. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[52]).
A[53] = 6.	// &A[53] = 0x5607d3786994. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[53]).
A[54] = 1.	// &A[54] = 0x5607d3786998. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[54]).
A[55] = 2.	// &A[55] = 0x5607d378699c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[55]).
A[56] = 7.	// &A[56] = 0x5607d37869a0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[56]).
A[57] = 1.	// &A[57] = 0x5607d37869a4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[57]).
A[58] = 1.	// &A[58] = 0x5607d37869a8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[58]).
A[59] = 0.	// &A[59] = 0x5607d37869ac. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[59]).
A[60] = 3.	// &A[60] = 0x5607d37869b0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[60]).
A[61] = 3.	// &A[61] = 0x5607d37869b4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[61]).
A[62] = 2.	// &A[62] = 0x5607d37869b8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[62]).
A[63] = 4.	// &A[63] = 0x5607d37869bc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[63]).
A[64] = 6.	// &A[64] = 0x5607d37869c0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[64]).
A[65] = 4.	// &A[65] = 0x5607d37869c4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[65]).
A[66] = 4.	// &A[66] = 0x5607d37869c8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[66]).
A[67] = 5.	// &A[67] = 0x5607d37869cc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[67]).
A[68] = 8.	// &A[68] = 0x5607d37869d0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[68]).
A[69] = 1.	// &A[69] = 0x5607d37869d4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[69]).
A[70] = 9.	// &A[70] = 0x5607d37869d8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[70]).

A[71] = 5. // &A[71] = 0x5607d37869dc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[71]).
A[72] = 4. // &A[72] = 0x5607d37869e0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[72]).
A[73] = 1. // &A[73] = 0x5607d37869e4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[73]).
A[74] = 8. // &A[74] = 0x5607d37869e8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[74]).
A[75] = 6. // &A[75] = 0x5607d37869ec. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[75]).
A[76] = 7. // &A[76] = 0x5607d37869f0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[76]).
A[77] = 2. // &A[77] = 0x5607d37869f4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[77]).
A[78] = 6. // &A[78] = 0x5607d37869f8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[78]).
A[79] = 0. // &A[79] = 0x5607d37869fc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[79]).
A[80] = 1. // &A[80] = 0x5607d3786a00. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[80]).
A[81] = 4. // &A[81] = 0x5607d3786a04. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[81]).
A[82] = 0. // &A[82] = 0x5607d3786a08. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[82]).
A[83] = 7. // &A[83] = 0x5607d3786a0c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[83]).
A[84] = 2. // &A[84] = 0x5607d3786a10. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[84]).
A[85] = 1. // &A[85] = 0x5607d3786a14. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[85]).
A[86] = 9. // &A[86] = 0x5607d3786a18. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[86]).
A[87] = 9. // &A[87] = 0x5607d3786a1c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[87]).
A[88] = 5. // &A[88] = 0x5607d3786a20. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[88]).
A[89] = 2. // &A[89] = 0x5607d3786a24. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[89]).
A[90] = 9. // &A[90] = 0x5607d3786a28. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[90]).
A[91] = 0. // &A[91] = 0x5607d3786a2c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[91]).
A[92] = 5. // &A[92] = 0x5607d3786a30. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[92]).

A[93] = 3.   // &A[93] = 0x5607d3786a34. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[93]).
A[94] = 6.   // &A[94] = 0x5607d3786a38. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[94]).
A[95] = 2.   // &A[95] = 0x5607d3786a3c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[95]).
A[96] = 8.   // &A[96] = 0x5607d3786a40. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[96]).
A[97] = 1.   // &A[97] = 0x5607d3786a44. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[97]).
A[98] = 7.   // &A[98] = 0x5607d3786a48. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[98]).
A[99] = 6.   // &A[99] = 0x5607d3786a4c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[99]).

STEP_2: SORT THE ELEMENT VALUES OF THE ARRAY NAMED A TO BE IN ASCENDING ORDER.

// Sort the integer values stored in array A to be in ascending order using the Bubble Sort algorithm.
bubble_sort(A, S);

A = 0x5607d37868c0. // memory address of A[0]

A[0] = 0.   // &A[0] = 0x5607d37868c0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[0]).
A[1] = 0.   // &A[1] = 0x5607d37868c4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[1]).
A[2] = 0.   // &A[2] = 0x5607d37868c8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[2]).
A[3] = 0.   // &A[3] = 0x5607d37868cc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[3]).
A[4] = 0.   // &A[4] = 0x5607d37868d0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[4]).
A[5] = 0.   // &A[5] = 0x5607d37868d4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[5]).
A[6] = 0.   // &A[6] = 0x5607d37868d8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[6]).

A[7] = 0. // &A[7] = 0x5607d37868dc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[7]).
A[8] = 0. // &A[8] = 0x5607d37868e0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[8]).
A[9] = 0. // &A[9] = 0x5607d37868e4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[9]).
A[10] = 0. // &A[10] = 0x5607d37868e8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[10]).
A[11] = 1. // &A[11] = 0x5607d37868ec. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[11]).
A[12] = 1. // &A[12] = 0x5607d37868f0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[12]).
A[13] = 1. // &A[13] = 0x5607d37868f4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[13]).
A[14] = 1. // &A[14] = 0x5607d37868f8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[14]).
A[15] = 1. // &A[15] = 0x5607d37868fc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[15]).
A[16] = 1. // &A[16] = 0x5607d3786900. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[16]).
A[17] = 1. // &A[17] = 0x5607d3786904. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[17]).
A[18] = 1. // &A[18] = 0x5607d3786908. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[18]).
A[19] = 1. // &A[19] = 0x5607d378690c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[19]).
A[20] = 1. // &A[20] = 0x5607d3786910. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[20]).
A[21] = 1. // &A[21] = 0x5607d3786914. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[21]).
A[22] = 1. // &A[22] = 0x5607d3786918. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[22]).
A[23] = 1. // &A[23] = 0x5607d378691c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[23]).
A[24] = 1. // &A[24] = 0x5607d3786920. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[24]).
A[25] = 1. // &A[25] = 0x5607d3786924. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[25]).
A[26] = 1. // &A[26] = 0x5607d3786928. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[26]).
A[27] = 1. // &A[27] = 0x5607d378692c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[27]).
A[28] = 1. // &A[28] = 0x5607d3786930. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[28]).

A[29] = 2. // &A[29] = 0x5607d3786934. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[29]).
A[30] = 2. // &A[30] = 0x5607d3786938. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[30]).
A[31] = 2. // &A[31] = 0x5607d378693c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[31]).
A[32] = 2. // &A[32] = 0x5607d3786940. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[32]).
A[33] = 2. // &A[33] = 0x5607d3786944. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[33]).
A[34] = 2. // &A[34] = 0x5607d3786948. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[34]).
A[35] = 2. // &A[35] = 0x5607d378694c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[35]).
A[36] = 2. // &A[36] = 0x5607d3786950. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[36]).
A[37] = 2. // &A[37] = 0x5607d3786954. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[37]).
A[38] = 2. // &A[38] = 0x5607d3786958. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[38]).
A[39] = 2. // &A[39] = 0x5607d378695c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[39]).
A[40] = 3. // &A[40] = 0x5607d3786960. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[40]).
A[41] = 3. // &A[41] = 0x5607d3786964. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[41]).
A[42] = 3. // &A[42] = 0x5607d3786968. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[42]).
A[43] = 3. // &A[43] = 0x5607d378696c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[43]).
A[44] = 3. // &A[44] = 0x5607d3786970. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[44]).
A[45] = 3. // &A[45] = 0x5607d3786974. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[45]).
A[46] = 3. // &A[46] = 0x5607d3786978. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[46]).
A[47] = 4. // &A[47] = 0x5607d378697c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[47]).
A[48] = 4. // &A[48] = 0x5607d3786980. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[48]).
A[49] = 4. // &A[49] = 0x5607d3786984. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[49]).
A[50] = 4. // &A[50] = 0x5607d3786988. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[50]).

A[51] = 4. // &A[51] = 0x5607d378698c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[51]).
A[52] = 4. // &A[52] = 0x5607d3786990. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[52]).
A[53] = 4. // &A[53] = 0x5607d3786994. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[53]).
A[54] = 4. // &A[54] = 0x5607d3786998. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[54]).
A[55] = 4. // &A[55] = 0x5607d378699c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[55]).
A[56] = 4. // &A[56] = 0x5607d37869a0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[56]).
A[57] = 5. // &A[57] = 0x5607d37869a4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[57]).
A[58] = 5. // &A[58] = 0x5607d37869a8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[58]).
A[59] = 5. // &A[59] = 0x5607d37869ac. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[59]).
A[60] = 5. // &A[60] = 0x5607d37869b0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[60]).
A[61] = 5. // &A[61] = 0x5607d37869b4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[61]).
A[62] = 5. // &A[62] = 0x5607d37869b8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[62]).
A[63] = 5. // &A[63] = 0x5607d37869bc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[63]).
A[64] = 6. // &A[64] = 0x5607d37869c0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[64]).
A[65] = 6. // &A[65] = 0x5607d37869c4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[65]).
A[66] = 6. // &A[66] = 0x5607d37869c8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[66]).
A[67] = 6. // &A[67] = 0x5607d37869cc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[67]).
A[68] = 6. // &A[68] = 0x5607d37869d0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[68]).
A[69] = 6. // &A[69] = 0x5607d37869d4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[69]).
A[70] = 6. // &A[70] = 0x5607d37869d8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[70]).
A[71] = 6. // &A[71] = 0x5607d37869dc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[71]).
A[72] = 7. // &A[72] = 0x5607d37869e0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[72]).

A[73] = 7.   // &A[73] = 0x5607d37869e4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[73]).
A[74] = 7.   // &A[74] = 0x5607d37869e8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[74]).
A[75] = 7.   // &A[75] = 0x5607d37869ec. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[75]).
A[76] = 7.   // &A[76] = 0x5607d37869f0. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[76]).
A[77] = 7.   // &A[77] = 0x5607d37869f4. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[77]).
A[78] = 7.   // &A[78] = 0x5607d37869f8. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[78]).
A[79] = 7.   // &A[79] = 0x5607d37869fc. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[79]).
A[80] = 7.   // &A[80] = 0x5607d3786a00. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[80]).
A[81] = 7.   // &A[81] = 0x5607d3786a04. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[81]).
A[82] = 7.   // &A[82] = 0x5607d3786a08. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[82]).
A[83] = 8.   // &A[83] = 0x5607d3786a0c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[83]).
A[84] = 8.   // &A[84] = 0x5607d3786a10. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[84]).
A[85] = 8.   // &A[85] = 0x5607d3786a14. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[85]).
A[86] = 8.   // &A[86] = 0x5607d3786a18. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[86]).
A[87] = 8.   // &A[87] = 0x5607d3786a1c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[87]).
A[88] = 8.   // &A[88] = 0x5607d3786a20. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[88]).
A[89] = 8.   // &A[89] = 0x5607d3786a24. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[89]).
A[90] = 8.   // &A[90] = 0x5607d3786a28. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[90]).
A[91] = 8.   // &A[91] = 0x5607d3786a2c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[91]).
A[92] = 8.   // &A[92] = 0x5607d3786a30. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[92]).
A[93] = 9.   // &A[93] = 0x5607d3786a34. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[93]).
A[94] = 9.   // &A[94] = 0x5607d3786a38. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[94]).

A[95] = 9.   // &A[95] = 0x5607d3786a3c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[95]).
A[96] = 9.   // &A[96] = 0x5607d3786a40. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[96]).
A[97] = 9.   // &A[97] = 0x5607d3786a44. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[97]).
A[98] = 9.   // &A[98] = 0x5607d3786a48. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[98]).
A[99] = 9.   // &A[99] = 0x5607d3786a4c. (memory address of the first memory cell comprising the block of 4 contiguous memory cells allocated to A[99]).

STEP_3: CREATE A DYNAMIC ARRAY WHICH IS NAMED B AND WHICH IS COMPRISED OF T INT TYPE VALUES.

// Declare a pointer-to-int type variable named B.
int * B;

// Allocate T contiguous int-sized chunks of memory and store the memory address of the first int-sized chunk of memory, B[0], inside the pointer-to-int type variable named B.
B = new int [T];

B = 0x5607d3786a60. // memory address of B[0]

B[0] = 0.   // &B[0] = 0x5607d3786a60. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[0]).
B[1] = 0.   // &B[1] = 0x5607d3786a64. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[1]).
B[2] = 0.   // &B[2] = 0x5607d3786a68. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[2]).
B[3] = 0.   // &B[3] = 0x5607d3786a6c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[3]).
B[4] = 0.   // &B[4] = 0x5607d3786a70. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[4]).
B[5] = 0.   // &B[5] = 0x5607d3786a74. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[5]).
B[6] = 0.   // &B[6] = 0x5607d3786a78. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[6]).

B[7] = 0.   // &B[7] = 0x5607d3786a7c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[7]).
B[8] = 0.   // &B[8] = 0x5607d3786a80. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[8]).
B[9] = 0.   // &B[9] = 0x5607d3786a84. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[9]).

STEP_4: FOR EACH ELEMENT B[i] OF THE ARRAY NAMED B, STORE THE NUMBER OF TIMES i APPEARS AS AN ELEMENT VALUE IN THE ARRAY NAMED A.

B = 0x5607d3786a60. // memory address of B[0]

B[0] = 11.   // &B[0] = 0x5607d3786a60. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[0]).
B[1] = 18.   // &B[1] = 0x5607d3786a64. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[1]).
B[2] = 11.   // &B[2] = 0x5607d3786a68. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[2]).
B[3] = 7.   // &B[3] = 0x5607d3786a6c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[3]).
B[4] = 10.   // &B[4] = 0x5607d3786a70. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[4]).
B[5] = 7.   // &B[5] = 0x5607d3786a74. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[5]).
B[6] = 8.   // &B[6] = 0x5607d3786a78. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[6]).
B[7] = 11.   // &B[7] = 0x5607d3786a7c. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[7]).
B[8] = 10.   // &B[8] = 0x5607d3786a80. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[8]).
B[9] = 7.   // &B[9] = 0x5607d3786a84. (memory address of the first byte-sized memory cell comprising the block of 4 contiguous byte-sized memory cells allocated to B[9]).

STEP_5: CREATE A DYNAMIC ARRAY WHICH IS NAMED C AND WHICH IS COMPRISED OF T POINTER-TO-CHAR TYPE VALUES.

```
// Declare one pointer-to-pointer-to-char type variable.
char ** C;

// Allocate T contiguous pointer-to-char-sized chunks of memory and store the memory address of the first pointer-to-char-sized chunk of memory, C[0], inside the pointer-to-pointer-to-char type variable named C.
C = new char * [T];

// C is a two-dimensional array which depicts a histogram (i.e. bar graph) such the length of the ith row is identical to the value stored in B[i].
for (i = 0; i < T; i += 1)
{
	C[i] = new char [B[i]];
	for (k = 0; k < B[i]; k += 1) C[i][k] = 'X';
}
```

C = 0x5607d3786a90. // memory address of C[0]

C[0] = XXXXXXXXXXX. // &C[0] = 0x5607d3786a90. (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[0]).
C[1] = XXXXXXXXXXXXXXXXXX. // &C[1] = 0x5607d3786a98. (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[1]).
C[2] = XXXXXXXXXXX. // &C[2] = 0x5607d3786aa0. (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[2]).
C[3] = XXXXXXX. // &C[3] = 0x5607d3786aa8. (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[3]).
C[4] = XXXXXXXXXX. // &C[4] = 0x5607d3786ab0. (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[4]).
C[5] = XXXXXXX. // &C[5] = 0x5607d3786ab8. (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[5]).
C[6] = XXXXXXXX. // &C[6] = 0x5607d3786ac0. (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[6]).
C[7] = XXXXXXXXXXX. // &C[7] = 0x5607d3786ac8. (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[7]).
C[8] = XXXXXXXXXX. // &C[8] = 0x5607d3786ad0. (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[8]).
C[9] = XXXXXXX. // &C[9] = 0x5607d3786ad8. (memory address of the first byte-sized memory cell comprising the block of 8 contiguous byte-sized memory cells allocated to C[9]).

STEP_6: RELEASE MEMORY WHICH WAS ALLOCATED TO THE DYNAMIC ARRAYS NAMED A, B, AND C.

// De-allocate memory which was assigned to the dynamically-allocated array of S int type values.
delete [] A; // Free up S contiguous int-sized chunks of memory which were assigned to the dynamic array named A.

De-allocate memory which was assigned to the dynamically-allocated array of T int type values.
delete [] B; // Free up T contiguous int-sized chunks of memory which were assigned to the dynamic array named B.

// De-allocate memory which was assigned to the dynamically-allocated array of T pointer-to-char type values.
for (i = 0; i < T; i += 1) delete [] C[i]; // Free up B[i] char-sized chunks of memory which were assigned to the dynamic array named C[i].
delete [] C; // Free up T contiguous pointer-to-char-sized chunks of memory which were assigned to the dynamic array named C.

End Of Program

[End of abridged plain-text content from POINTERS_AND_ARRAYS]

POINT

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/points_on_cartesian_grid.png

---

The C++ program featured in this tutorial web page demonstrates the concept of Object Oriented Programming (OOP). The program implements a user defined data type for instantiating POINT type objects. Each POINT type object represents a two-dimensional point plotted on a Cartesian grid such that the X value of a POINT object represents a whole number position along the horizontal axis of the Cartesian grid while the Y value of a POINT object represents a whole number position along the vertical axis of the Cartesian grid. A POINT object can execute various functions including the ability to compute the distance between itself and some input POINT object and the ability to compute the slope of the line which intersects itself and some input POINT object.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

---

class : object :: data_type : variable.

SOFTWARE_APPLICATION_COMPONENTS

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.cpp

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point_class_tester.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point_class_tester_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ code from the files named point.h, point.cpp, and point_class_tester.cpp into their own new text editor documents and save those documents using their corresponding file names:

point.h

point.cpp

point_class_tester.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ point_class_tester.cpp point.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: Observe program results on the command line terminal and in the output file.

POINT_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the POINT class.

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.h

---

/**

```
 * file: point.h
 * type: C++ (header file)
 * author: karbytes
 * date: 07_JULY_2023
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#ifndef POINT_H // If point.h has not already been linked to a source file (.cpp),
#define POINT_H // then link this header file to the source file(s) which include this header file.

/* preprocessing directives */
#include < iostream > // library for defining objects which handle command line input and
command line output
#include < fstream > // library for defining objects which handle file input and file output
#include < cmath > // library which defines various math functions such as square root (sqrt())
and sine (sin())
#include < string > // library which defines a sequence of text characters (i.e. char type values)
as a string type variable
#define MINIMUM_X -999 // constant which represents minimum X value
#define MAXIMUM_X 999 // constant which represents maximum X value
#define MINIMUM_Y -999 // constant which represents minimum Y value
#define MAXIMUM_Y 999 // constant which represents maximum Y value
#define PI 3.14159 // constant which represents the approximate value of a circle's
circumference divided by that circle's diameter

/**
 * Define a class which is used to instantiate POINT type objects.
 *
 * (An object is a variable whose data type is user defined rather than native to the C++
programming language).
 *
 * A POINT object represents a whole number coordinate pair in the form (X,Y).
 *
 * X represents a specific whole number position along the x-axis (i.e. horizontal dimension) of a
two-dimensional Cartesian grid.
 * Y represents a specific whole number position along the y-axis (i.e. vertical dimension) of the
same two-dimensional Cartesian grid.
 *
 * X stores one integer value at a time which is no smaller than MINIMUM_X and which is no
larger than MAXIMUM_X.
 * Y stores one integer value at a time which is no smaller than MINIMUM_Y and which is no
larger than MAXIMUM_Y.
 */
```

```
class POINT
{
private:
   int X, Y; // data attributes
public:
   POINT(); // default constructor
   POINT(int X, int Y); // normal constructor
   POINT(const POINT & point); // copy constructor
   int get_X(); // getter method
   int get_Y(); // getter method
   bool set_X(int X); // setter method
   bool set_Y(int Y); // setter method
   double get_distance_from(POINT & point); // getter method
   double get_slope_of_line_to(POINT & point); // getter method
   void print(std::ostream & output = std::cout); // descriptor method
   friend std::ostream & operator << (std::ostream & output, POINT & point); // descriptor
method
   ~POINT(); // destructor
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

POINT_CLASS_SOURCE_CODE

The following source code defines the functions of the POINT class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.cpp

---

```
/**
 * file: point.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
```

```
#include "point.h" // Include the C++ header file which contains preprocessing directives,
variable declarations, and function prototypes for the POINT class.

/**
 * The default constructor method of the POINT class
 * instantiates POINT type objects
 * whose X value is initially set to 0 and
 * whose Y value is initially set to 0.
 *
 * The default constructor method of the POINT class is invoked
 * when a POINT type variable is declared as follows:
 *
 * // variable declaration example one
 * POINT point_0;
 *
 * // variable declaration example two
 * POINT point_1 = POINT();
 */
POINT::POINT()
{
	std::cout << "\n\nCreating the POINT type object whose memory address is " << this <<
"...";
	X = 0;
	Y = 0;
}

/**
 * The normal constructor method of the POINT class
 * instantiates POINT type objects
 * whose X value is set to the leftmost function input value (if that input value is no smaller than
MINIMUM_X and no larger than MAXIMUM_X) and
 * whose Y value is set to the rightmost function input value (if that input value is no smaller than
MINIMUM_Y and no larger than MAXIMUM_Y).
 *
 * If a function input value is out of its specified range, then set the corresponding int type
property of this to 0.
 *
 * (The keyword this refers to the POINT object which is returned by this function).
 *
 * The normal constructor method of the POINT class is invoked when a POINT type variable is
declared as follows:
 *
 * // variable definition example one
 * POINT point_0 = POINT(-55,84);
```

```
 *
 * // variable definition example two
 * POINT point_1 = POINT(3,-4);
 *
 * // variable definition example three
 * POINT point_2 = POINT(-1000, 999); // point_2 = POINT(0,999).
 *
 * // variable definition example four
 * POINT point_3 = POINT(1000,-999); // point_3 = POINT(0,-999).
 *
 * // variable definition example five
 * POINT point_4 = POINT(999,-1000); // point_4 = POINT(999,0).
 *
 * // variable definition example six
 * POINT point_5 = POINT(-999,1000); // point_5 = POINT(-999,0).
 */
POINT::POINT(int X, int Y)
{
	std::cout << "\n\nCreating the POINT type object whose memory address is " << this <<
"...";
	this -> X = ((X < MINIMUM_X) || (X > MAXIMUM_X)) ? 0 : X; // Set the X property of the
POINT instance being created to 0 if the function input X value is out of range.
	this -> Y = ((Y < MINIMUM_Y) || (Y > MAXIMUM_Y)) ? 0 : Y; // Set the Y property of the
POINT instance being created to 0 if the function input Y value is out of range.
}

/**
 * The copy constructor method of the POINT class
 * instantiates POINT type objects
 * whose X value is set to the X value of the input POINT object and
 * whose Y value is set to the Y value of the input POINT object.
 *
 * The copy constructor method of the POINT class is invoked when a POINT type variable is
declared as follows:
 *
 * // variable definition example one
 * POINT point_0 = POINT(33,55);
 * POINT point_1 = POINT(point_0); // point_1 = POINT(33,55).
 *
 * // variable definition example two
 * POINT point_2 = POINT(point_1); // point_2 = POINT(33,55).
 */
POINT::POINT(const POINT & point)
{
```

```
	std::cout << "\n\nCreating the POINT type object whose memory address is " << this <<
"...";
	X = point.X;
	Y = point.Y;
}

/**
 * The getter method of the POINT class returns the value of the caller POINT object's X
property.
 *
 * X is an int type variable which stores exactly one integer value at a time which is no smaller
than MINIMUM_X and which is no larger than MAXIMUM_X.
 *
 * X represents a specific whole number position along the x-axis (i.e. horizontal dimension) of a
two-dimensional Cartesian grid.
 */
int POINT::get_X()
{
	return X;
}

/**
 * The getter method of the POINT class returns the value of the caller POINT object's Y
property.
 *
 * Y is an int type variable which stores exactly one integer value at a time which is no smaller
than MINIMUM_Y and which is no larger than MAXIMUM_Y.
 *
 * Y represents a specific whole number position along the y-axis (i.e. vertical dimension) of a
two-dimensional Cartesian grid.
 */
int POINT::get_Y()
{
	return Y;
}

/**
 * The setter method of the POINT class sets the POINT object's X property to the function input
value
 * if that value is no smaller than MINIMUM_X and which is no larger than MAXIMUM_X.
 *
 * If the input value is in range, then return true.
 * Otherwise, do not change the caller POINT object's X value and return false.
 *
```

```
 * (The keyword this refers to the POINT object which calls this function).
 *
 * (X represents a specific whole number position along the x-axis (i.e. horizontal dimension) of
a two-dimensional Cartesian grid).
 */
bool POINT::set_X(int X)
{
	if ((X >= MINIMUM_X) && (X <= MAXIMUM_X))
	{
	this -> X = X;
	return true;
	}
	return false;
}

/**
 * The setter method of the POINT class sets the POINT object's Y property to the function input
value
 * if that value is no smaller than MINIMUM_Y and which is no larger than MAXIMUM_Y.
 *
 * If the input value is in range, then return true.
 * Otherwise, do not change the caller POINT object's Y value and return false.
 *
 * (The keyword this refers to the POINT object which calls this function).
 *
 * (Y represents a specific whole number position along the y-axis (i.e. vertical dimension) of a
two-dimensional Cartesian grid).
 */
bool POINT::set_Y(int Y)
{
	if ((Y >= MINIMUM_Y) && (Y <= MAXIMUM_Y))
	{
	this -> Y = Y;
	return true;
	}
	return false;
}

/**
 * The getter method of the POINT class returns the length of the shortest path
 * between the two-dimensional point represented by the the caller POINT object (i.e. this)
 * and the two-dimensional point represented by the input POINT object (i.e. point).
 *
 * Use the Pythagorean Theorem to compute the length of a right triangle's hypotenuse
```

```
 * such that the two end points of that hypotenuse are represented by this and point.
 *
 * (A hypotenuse is the only side of a right triangle which does not form a right angle
 * with any other side of that triangle).
 *
 * (A hypotenuse is the longest side of a triangle (and a triangle is a three-sided polygon
 * in which three unique line segments connect three unique points)).
 *
 * // c is the length of a right triangle's hypotenuse.
 * // a is the length of that right triangle's horizontal leg.
 * // b is the length of that triangle's vertical leg.
 * (c * c) = (a * a) + (b * b).
 *
 * // sqrt() is a native C++ function defined in the cmath library.
 * c = square_root( (a * a) + (b * b)).
 */
double POINT::get_distance_from(POINT & point)
{
	int horizontal_difference = 0.0, vertical_difference = 0.0;
	horizontal_difference = X - point.X; // a
	vertical_difference = Y - point.Y; // b
	return sqrt((horizontal_difference * horizontal_difference) + (vertical_difference *
vertical_difference)); // c
}

/**
 * The getter method of the POINT class returns the slope of the line which intersects
 * the two-dimensional point represented by the caller POINT instance (i.e. this)
 * and the two-dimensional point represented by the input POINT instance (i.e. point).
 *
 * // y := f(x),
 * // b := f(0),
 * // f is a function whose input is an x-axis position and whose output is a y-axis position.
 * y := mx + b.
 *
 * // m is a constant which represents the rate at which y changes in relation to x changing.
 * m := (y - b) / x.
 *
 * // m represents the difference of the two y-values divided by the difference of the two x-values.
 * m := (point.Y - this.Y) / (point.X - this.X).
 */
double POINT::get_slope_of_line_to(POINT & point)
{
	double vertical_difference = 0.0, horizontal_difference = 0.0, result = 0.0;
```

```
	vertical_difference = point.Y - Y;
	horizontal_difference = point.X - X;
	result = vertical_difference / horizontal_difference;
	if (result == -0) result = 0; // Signed zeros sometimes occur inside of C++ program runtime instances.
	return result;
}

/**
 * The print method of the POINT class prints a description of the caller POINT object to the output stream.
 *
 * Note that the default value of the function input parameter is the standard command line output stream (std::cout).
 *
 * The default parameter is defined in the POINT class header file (i.e. point.h) and not in the POINT class source file (i.e. point.cpp).
 */
void POINT::print(std::ostream & output)
{
	output << "\n\n--------------------------------------------------------------------------------------------------";
	output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.";
	output << "\n&X = " << &X << ". // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.";
	output << "\n&Y = " << &Y << ". // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.";
	output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(POINT) = " << sizeof(POINT) << ". // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nX = " << X << ". // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.";
	output << "\nY = " << Y << ". // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.";
	output << "\n--------------------------------------------------------------------------------------------------";
}
```

```
/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (<<).
 *
 * (Overloading an operator is assigning a different function to a native operator other than the
function which that operator is used to represent by default).
 *
 * Note that the default value of the leftmost function input parameter is the standard command
line output stream (std::cout).
 * The default parameter is defined in the POINT class header file (i.e. point.h).
 *
 * The friend function is not a member of the POINT class,
 * but the friend function has access to the private and protected members
 * of the POINT class and not just to the public members of the POINT class.
 *
 * The friend keyword only prefaces the function prototype of this function
 * (and the prototype of this function is declared in the POINT class header file (i.e. point.h)).
 *
 * The friend keyword does not preface the definition of this function
 * (and the definition of this function is specified in the POINT class source file (i.e. point.cpp)).
 *
 * // overloaded print function example one
 * POINT point_0;
 * std::cout << point_0; // identical to point_0.print();
 *
 * // overloaded print function example two
 * std::ofstream file;
 * POINT point_1;
 * file << point_1; // identical to point_1.print(file);
 */
std::ostream & operator << (std::ostream & output, POINT & point)
{
	point.print(output);
	return output;
}

/**
 * The destructor method of the POINT class de-allocates memory which was used to
 * instantiate the POINT object which is calling this function.
 *
 * The destructor method of the POINT class is automatically called when
 * the program scope in which the caller POINT object was instantiated terminates.
 */
POINT::~POINT()
```

{
	std::cout << "\n\nDeleting the POINT type object whose memory address is " << this << "...";
}

PROGRAM_SOURCE_CODE

The following source code defines the client which implements the POINT class. The client executes a series of unit tests which demonstrate how the POINT class methods work.

C++_source_file: https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point_class_tester.cpp

---

```
/**
 * file: point_class_tester.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

#include "point.h" // Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the POINT class.

/* function prototypes */
void unit_test_0(std::ostream & output);
void unit_test_1(std::ostream & output);
void unit_test_2(std::ostream & output);
void unit_test_3(std::ostream & output);
void unit_test_4(std::ostream & output);
void unit_test_5(std::ostream & output);
void unit_test_6(std::ostream & output);
void unit_test_7(std::ostream & output);
void unit_test_8(std::ostream & output);
void unit_test_9(std::ostream & output);
void unit_test_10(std::ostream & output);
void unit_test_11(std::ostream & output);
```

```
// Unit Test # 0: POINT class default constructor, POINT class print method, and POINT class
destructor.
void unit_test_0(std::ostream & output)
{
    output << "\n\n----------------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 0: POINT class default constructor, POINT class print method, and
POINT class destructor.";
    output << "\n----------------------------------------------------------------------------------------------------";
    output << "\nPOINT point;";
    output << "\npoint.print(output);";
    POINT point;
    point.print(output);
}

// Unit Test # 1: POINT class default constructor, POINT class print method (with default
parameter), and POINT class destructor.
void unit_test_1(std::ostream & output)
{
    output << "\n\n----------------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 1: POINT class default constructor, POINT class print method (with
default parameter), and POINT class destructor.";
    output << "\n----------------------------------------------------------------------------------------------------";
    output << "\nPOINT point;";
    output << "\npoint.print(); // Standard command line output (std::cout) is the default parameter
for the POINT print method.";
    POINT point;
    point.print(); // Standard command line output (std::cout) is the default parameter for the
POINT print method.
}

// Unit Test # 2: POINT class default constructor, POINT class overloaded ostream operator
method (which is functionally the same as the POINT class print method), and POINT class
destructor.
void unit_test_2(std::ostream & output)
{
    output << "\n\n----------------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 2: POINT class default constructor, POINT class overloaded ostream
operator method (which is functionally the same as the POINT class print method), and POINT
class destructor.";
    output << "\n----------------------------------------------------------------------------------------------------";
    output << "\nPOINT point;";
    output << "\noutput << point; // functionally equivalent to: point.print(output);";
    POINT point;
    output << point;
```

```
}

// Unit Test # 3: POINT class default constructor (using the function explicity rather than implicitly), POINT class overloaded ostream operator method, and POINT class destructor.
void unit_test_3(std::ostream & output)
{
    output << "\n\n----------------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 3: POINT class default constructor (using that function explicity rather than implicitly), POINT class overloaded ostream operator method, and POINT class destructor.";
    output << "\n----------------------------------------------------------------------------------------------------";
    output << "\nPOINT point = POINT(); // functionally equivalent to: POINT point;";
    output << "\noutput << point;";
    POINT point = POINT();
    output << point;
}

// Unit Test # 4: POINT class normal constructor (using only valid function inputs), POINT class overloaded ostream operator method, and POINT class destructor.
void unit_test_4(std::ostream & output)
{
    output << "\n\n----------------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 4: POINT class normal constructor (using only valid function inputs), POINT class overloaded ostream operator method, and POINT class destructor.";
    output << "\n----------------------------------------------------------------------------------------------------";
    output << "\nPOINT point = POINT(-503,404);";
    output << "\noutput << point;";
    POINT point = POINT(-503,404);
    output << point;
}

// Unit Test # 5: POINT class normal constructor (using only valid function inputs), POINT class overloaded ostream operator method, and POINT class destructor.
void unit_test_5(std::ostream & output)
{
    output << "\n\n----------------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 5: POINT class normal constructor (using only valid function inputs), POINT class overloaded ostream operator method, and POINT class destructor.";
    output << "\n----------------------------------------------------------------------------------------------------";
    output << "\nPOINT point_0 = POINT(-999,-999);";
    output << "\nPOINT point_1 = POINT(999, 999);";
    output << "\nPOINT point_2 = POINT(-999, 999);";
    output << "\nPOINT point_3 = POINT(999, -999);";
    output << "\noutput << point_0;";
```

```
    output << "\noutput << point_1;";
    output << "\noutput << point_2;";
    output << "\noutput << point_3;";
    POINT point_0 = POINT(-999,-999);
    POINT point_1 = POINT(999, 999);
    POINT point_2 = POINT(-999, 999);
    POINT point_3 = POINT(999, -999);
    output << point_0;
    output << point_1;
    output << point_2;
    output << point_3;
}

// Unit Test # 6: POINT class normal constructor (using both valid and invalid function inputs), POINT class overloaded ostream operator method, and POINT class destructor.
void unit_test_6(std::ostream & output)
{
    output << "\n\n--------------------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 6: POINT class normal constructor (using both valid and invalid function inputs), POINT class overloaded ostream operator method, and POINT class destructor.";
    output << "\n--------------------------------------------------------------------------------------------------------";
    output << "\nPOINT point_0 = POINT(-1000, -999); // point_0 = POINT(0,-999).";
    output << "\nPOINT point_1 = POINT(1000, -999); // point_1 = POINT(0,-999).";
    output << "\nPOINT point_2 = POINT(-999, -1000); // point_2 = POINT(-999,0).";
    output << "\nPOINT point_3 = POINT(-999, 1000); // point_3 = POINT(-999,0).";
    output << "\nPOINT point_4 = POINT(-1000, -1000); // point_4 = POINT(0,0).";
    output << "\nPOINT point_5 = POINT(1000, 1000); // point_5 = POINT(0,0).";
    output << "\nPOINT point_6 = POINT(999, 999); // point_6 = POINT(999,999).";
    output << "\noutput << point_0;";
    output << "\noutput << point_1;";
    output << "\noutput << point_2;";
    output << "\noutput << point_3;";
    output << "\noutput << point_4;";
    output << "\noutput << point_5;";
    output << "\noutput << point_6;";
    POINT point_0 = POINT(-1000, -999); // point_0 = POINT(0,-999).
    POINT point_1 = POINT(1000, -999); // point_1 = POINT(0,-999).
    POINT point_2 = POINT(-999, -1000); // point_2 = POINT(-999,0).
    POINT point_3 = POINT(-999, 1000); // point_3 = POINT(-999,0).
    POINT point_4 = POINT(-1000, -1000); // point_4 = POINT(0,0).
    POINT point_5 = POINT(1000, 1000); // point_5 = POINT(0,0).
    POINT point_6 = POINT(999, 999); // point_6 = POINT(999,999).
    output << point_0;
```

```
    output << point_1;
    output << point_2;
    output << point_3;
    output << point_4;
    output << point_5;
    output << point_6;
}

// Unit Test # 7: POINT class normal constructor, POINT class copy constructor, POINT class overloaded ostream operator method, and POINT class destructor.
void unit_test_7(std::ostream & output)
{
    output << "\n\n--------------------------------------------------------------------------------";
    output << "\nUnit Test # 7: POINT class normal constructor, POINT class copy constructor, POINT class overloaded ostream operator method, and POINT class destructor.";
    output << "\n--------------------------------------------------------------------------------";
    output << "\nPOINT point_0 = POINT(333, -666);";
    output << "\nPOINT point_1 = POINT(point_0);";
    output << "\noutput << point_0;";
    output << "\noutput << point_1;";
    POINT point_0 = POINT(333, -666);
    POINT point_1 = POINT(point_0);
    output << point_0;
    output << point_1;
}

// Unit Test # 8: POINT class normal constructor, POINT class distance getter method, POINT class overloaded ostream operator method, and POINT class destructor.
void unit_test_8(std::ostream & output)
{
    output << "\n\n--------------------------------------------------------------------------------";
    output << "\nUnit Test # 8: POINT class normal constructor, POINT class distance getter method, POINT class overloaded ostream operator method, and POINT class destructor.";
    output << "\n--------------------------------------------------------------------------------";
    output << "\nPOINT point_0 = POINT(1, 1);";
    output << "\nPOINT point_1 = POINT(-1, -1);";
    output << "\noutput << point_0;";
    output << "\noutput << point_1;";
    POINT point_0 = POINT(1, 1);
    POINT point_1 = POINT(-1, -1);
    output << point_0;
    output << point_1;
    output << "\npoint_0.get_distance_from(point_1) = " << point_0.get_distance_from(point_1) << ".";
```

```
   output << "\npoint_1.get_distance_from(point_0) = " << point_1.get_distance_from(point_0)
<< ".";
   output << "\npoint_0.get_distance_from(point_0) = " << point_0.get_distance_from(point_0)
<< ".";
   output << "\npoint_1.get_distance_from(point_1) = " << point_1.get_distance_from(point_1)
<< ".";
}

// Unit Test # 9: POINT class normal constructor, POINT class distance getter method, POINT
class slope getter method, POINT class overloaded ostream operator method, and POINT class
destructor.
void unit_test_9(std::ostream & output)
{
   output << "\n\n---------------------------------------------------------------------------------------------------";
   output << "\nUnit Test # 9: POINT class normal constructor, POINT class distance getter
method, POINT class slope getter method, POINT class overloaded ostream operator method,
and POINT class destructor.";
   output << "\n---------------------------------------------------------------------------------------------------";
   output << "\nPOINT point_0 = POINT(0, 4);";
   output << "\nPOINT point_1 = POINT(3, 0);";
   output << "\nPOINT point_2 = POINT(0, 0);";
   output << "\noutput << point_0;";
   output << "\noutput << point_1;";
   output << "\noutput << point_2;";
   POINT point_0 = POINT(0, 4);
   POINT point_1 = POINT(3, 0);
   POINT point_2 = POINT(0, 0);
   output << point_0;
   output << point_1;
   output << point_2;
   output << "\npoint_0.get_distance_from(point_1) = " << point_0.get_distance_from(point_1)
<< ".";
   output << "\npoint_1.get_distance_from(point_0) = " << point_1.get_distance_from(point_0)
<< ".";
   output << "\npoint_1.get_distance_from(point_2) = " << point_1.get_distance_from(point_2)
<< ".";
   output << "\npoint_2.get_distance_from(point_1) = " << point_2.get_distance_from(point_1)
<< ".";
   output << "\npoint_2.get_distance_from(point_0) = " << point_2.get_distance_from(point_0)
<< ".";
   output << "\npoint_0.get_distance_from(point_2) = " << point_0.get_distance_from(point_2)
<< ".";
   output << "\npoint_0.get_distance_from(point_0) = " << point_0.get_distance_from(point_0)
<< ".";
```

```
    output << "\npoint_1.get_distance_from(point_1) = " << point_1.get_distance_from(point_1)
<< ".";
    output << "\npoint_2.get_distance_from(point_2) = " << point_2.get_distance_from(point_2)
<< ".";
    output << "\npoint_0.get_slope_of_line_to(point_1) = " <<
point_0.get_slope_of_line_to(point_1) << ".";
    output << "\npoint_1.get_slope_of_line_to(point_0) = " <<
point_1.get_slope_of_line_to(point_0) << ".";
    output << "\npoint_1.get_slope_of_line_to(point_2) = " <<
point_1.get_slope_of_line_to(point_2) << ".";
    output << "\npoint_2.get_slope_of_line_to(point_1) = " <<
point_2.get_slope_of_line_to(point_1) << ".";
    output << "\npoint_2.get_slope_of_line_to(point_0) = " <<
point_2.get_slope_of_line_to(point_0) << ".";
    output << "\npoint_0.get_slope_of_line_to(point_2) = " <<
point_0.get_slope_of_line_to(point_2) << ".";
    output << "\npoint_0.get_slope_of_line_to(point_0) = " <<
point_0.get_slope_of_line_to(point_0) << ".";
    output << "\npoint_1.get_slope_of_line_to(point_1) = " <<
point_1.get_slope_of_line_to(point_1) << ".";
    output << "\npoint_2.get_slope_of_line_to(point_2) = " <<
point_2.get_slope_of_line_to(point_2) << ".";
}

// Unit Test # 10: POINT class normal constructor, POINT class data attribute getter methods,
POINT class overloaded ostream operator method, and POINT class destructor.
void unit_test_10(std::ostream & output)
{
    output << "\n\n--------------------------------------------------------------------------------";
    output << "\nUnit Test # 10: POINT class normal constructor, POINT class data attribute
getter methods, POINT class overloaded ostream operator method, and POINT class
destructor.";
    output << "\n--------------------------------------------------------------------------------";
    output << "\nPOINT point = POINT(33.3, 88.8); // point = POINT(33, 88).";
    output << "\noutput << point;";
    POINT point = POINT(33.3, 88.8);
    output << point;
    output << "\npoint.get_X() = " << point.get_X() << ".";
    output << "\npoint.get_Y() = " << point.get_Y() << ".";
}

// Unit Test # 11: POINT class normal constructor, POINT class data attribute setter methods,
POINT class overloaded ostream operator method, and POINT class destructor.
void unit_test_11(std::ostream & output)
```

```
{
    output << "\n\n--------------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 10: POINT class normal constructor, POINT class data attribute
setter methods, POINT class overloaded ostream operator method, and POINT class
destructor.";
    output << "\n--------------------------------------------------------------------------------------------------";
    output << "\nPOINT point = POINT(666,777);";
    output << "\noutput << point;";
    output << "\npoint.set_X(999);";
    output << "\noutput << point;";
    output << "\npoint.set_Y(-999);";
    output << "\noutput << point;";
    output << "\npoint.set_X(-1000);";
    output << "\noutput << point;";
    output << "\npoint.set_X(200);";
    output << "\noutput << point;";
    output << "\npoint.set_Y(-1000);";
    output << "\noutput << point;";
    output << "\npoint.set_Y(444);";
    output << "\noutput << point;";
    output << "\npoint.set_X(1000);";
    output << "\noutput << point;";
    output << "\npoint.set_Y(1000);";
    output << "\noutput << point;";
    POINT point = POINT(666,777);
    output << point;
    point.set_X(999);
    output << point;
    point.set_Y(-999);
    output << point;
    point.set_X(-1000);
    output << point;
    point.set_X(200);
    output << point;
    point.set_Y(-1000);
    output << point;
    point.set_Y(444);
    output << point;
    point.set_X(1000);
    output << point;
    point.set_Y(1000);
    output << point;
}
```

```
/* program entry point */
int main()
{
   // Declare a file output stream object.
   std::ofstream file;

   // Set the number of digits of floating-point numbers which are printed to the command line
terminal to 100 digits.
   std::cout.precision(100);

   // Set the number of digits of floating-point numbers which are printed to the file output stream
to 100 digits.
   file.precision(100);

   /**
    * If point_class_tester_output.txt does not already exist in the same directory as
point_class_tester.cpp,
    * create a new file named point_class_tester_output.txt.
    *
    * Open the plain-text file named point_class_tester_output.txt
    * and set that file to be overwritten with program data.
    */
   file.open("point_class_tester_output.txt");

   // Print an opening message to the command line terminal.
   std::cout << "\n\n--------------------------------";
   std::cout << "\nStart Of Program";
   std::cout << "\n--------------------------------";

   // Print an opening message to the file output stream.
   file << "--------------------------------";
   file << "\nStart Of Program";
   file << "\n--------------------------------";

   // Implement a series of unit tests which demonstrate the functionality of POINT class
variables.
   unit_test_0(std::cout);
   unit_test_0(file);
   unit_test_1(std::cout);
   unit_test_1(file);
   unit_test_2(std::cout);
   unit_test_2(file);
   unit_test_3(std::cout);
   unit_test_3(file);
```

```
    unit_test_4(std::cout);
    unit_test_4(file);
    unit_test_5(std::cout);
    unit_test_5(file);
    unit_test_6(std::cout);
    unit_test_6(file);
    unit_test_7(std::cout);
    unit_test_7(file);
    unit_test_8(std::cout);
    unit_test_8(file);
    unit_test_9(std::cout);
    unit_test_9(file);
    unit_test_10(std::cout);
    unit_test_10(file);
    unit_test_11(std::cout);
    unit_test_11(file);

    // Print a closing message to the command line terminal.
    std::cout << "\n\n--------------------------------";
    std::cout << "\nEnd Of Program";
    std::cout << "\n--------------------------------\n\n";

    // Print a closing message to the file output stream.
    file << "\n\n--------------------------------";
    file << "\nEnd Of Program";
    file << "\n--------------------------------";

    // Close the file output stream.
    file.close();

    // Exit the program.
    return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point_class_tester_output.txt

---

Start Of Program

Unit Test # 0: POINT class default constructor, POINT class print method, and POINT class destructor.

POINT point;
point.print(output);

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

Unit Test # 1: POINT class default constructor, POINT class print method (with default parameter), and POINT class destructor.

POINT point;
point.print(); // Standard command line output (std::cout) is the default parameter for the POINT print method.

Unit Test # 2: POINT class default constructor, POINT class overloaded ostream operator method (which is functionally the same as the POINT class print method), and POINT class destructor.

POINT point;
output << point; // functionally equivalent to: point.print(output);

----

this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

Unit Test # 3: POINT class default constructor (using that function explicity rather than implicitly), POINT class overloaded ostream operator method, and POINT class destructor.

POINT point = POINT(); // functionally equivalent to: POINT point;
output << point;

----

this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.

sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

Unit Test # 4: POINT class normal constructor (using only valid function inputs), POINT class overloaded ostream operator method, and POINT class destructor.

POINT point = POINT(-503,404);
output << point;

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = -503. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 404. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

Unit Test # 5: POINT class normal constructor (using only valid function inputs), POINT class overloaded ostream operator method, and POINT class destructor.

POINT point_0 = POINT(-999,-999);
POINT point_1 = POINT(999, 999);
POINT point_2 = POINT(-999, 999);

POINT point_3 = POINT(999, -999);
output << point_0;
output << point_1;
output << point_2;
output << point_3;

----
this = 0x7ffe4927b238. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b238. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b23c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = -999. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = -999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b240. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b240. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b244. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 999. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----

this = 0x7ffe4927b248. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.

&X = 0x7ffe4927b248. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.

&Y = 0x7ffe4927b24c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.

sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).

X = -999. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.

Y = 999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----

this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.

&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.

&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.

sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).

X = 999. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.

Y = -999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

Unit Test # 6: POINT class normal constructor (using both valid and invalid function inputs), POINT class overloaded ostream operator method, and POINT class destructor.

POINT point_0 = POINT(-1000, -999); // point_0 = POINT(0,-999).
POINT point_1 = POINT(1000, -999); // point_1 = POINT(0,-999).
POINT point_2 = POINT(-999, -1000); // point_2 = POINT(-999,0).
POINT point_3 = POINT(-999, 1000); // point_3 = POINT(-999,0).
POINT point_4 = POINT(-1000, -1000); // point_4 = POINT(0,0).
POINT point_5 = POINT(1000, 1000); // point_5 = POINT(0,0).
POINT point_6 = POINT(999, 999); // point_6 = POINT(999,999).
output << point_0;
output << point_1;
output << point_2;
output << point_3;
output << point_4;
output << point_5;
output << point_6;

----

this = 0x7ffe4927b220. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b220. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b224. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = -999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----

this = 0x7ffe4927b228. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.

&X = 0x7ffe4927b228. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b22c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = -999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b230. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b230. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b234. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = -999. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b238. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b238. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.

&Y = 0x7ffe4927b23c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = -999. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b240. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b240. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b244. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b248. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b248. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b24c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.

sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 999. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

Unit Test # 7: POINT class normal constructor, POINT class copy constructor, POINT class overloaded ostream operator method, and POINT class destructor.

POINT point_0 = POINT(333, -666);
POINT point_1 = POINT(point_0);
output << point_0;
output << point_1;

----
this = 0x7ffe4927b248. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.

&X = 0x7ffe4927b248. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b24c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 333. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = -666. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 333. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = -666. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

Unit Test # 8: POINT class normal constructor, POINT class distance getter method, POINT class overloaded ostream operator method, and POINT class destructor.

POINT point_0 = POINT(1, 1);
POINT point_1 = POINT(-1, -1);
output << point_0;
output << point_1;

----
this = 0x7ffe4927b248. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b248. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b24c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 1. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 1. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = -1. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = -1. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.
----
point_0.get_distance_from(point_1) =
2.828427124746190290949243717477656900882720947265625.
point_1.get_distance_from(point_0) =
2.828427124746190290949243717477656900882720947265625.

point_0.get_distance_from(point_0) = 0.
point_1.get_distance_from(point_1) = 0.

Unit Test # 9: POINT class normal constructor, POINT class distance getter method, POINT class slope getter method, POINT class overloaded ostream operator method, and POINT class destructor.

POINT point_0 = POINT(0, 4);
POINT point_1 = POINT(3, 0);
POINT point_2 = POINT(0, 0);
output << point_0;
output << point_1;
output << point_2;

----
this = 0x7ffe4927b240. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b240. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b244. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 4. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b248. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b248. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.

&Y = 0x7ffe4927b24c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 3. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.
----
point_0.get_distance_from(point_1) = 5.
point_1.get_distance_from(point_0) = 5.
point_1.get_distance_from(point_2) = 3.
point_2.get_distance_from(point_1) = 3.
point_2.get_distance_from(point_0) = 4.
point_0.get_distance_from(point_2) = 4.
point_0.get_distance_from(point_0) = 0.
point_1.get_distance_from(point_1) = 0.
point_2.get_distance_from(point_2) = 0.
point_0.get_slope_of_line_to(point_1) =
-1.3333333333333332593184650249895639717578887939453125.

point_1.get_slope_of_line_to(point_0) =
-1.3333333333333332593184650249895639717578887939453125.
point_1.get_slope_of_line_to(point_2) = 0.
point_2.get_slope_of_line_to(point_1) = 0.
point_2.get_slope_of_line_to(point_0) = inf.
point_0.get_slope_of_line_to(point_2) = -inf.
point_0.get_slope_of_line_to(point_0) = -nan.
point_1.get_slope_of_line_to(point_1) = -nan.
point_2.get_slope_of_line_to(point_2) = -nan.

Unit Test # 10: POINT class normal constructor, POINT class data attribute getter methods, POINT class overloaded ostream operator method, and POINT class destructor.

POINT point = POINT(33.3, 88.8); // point = POINT(33, 88).
output << point;

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 33. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 88. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.
----
point.get_X() = 33.
point.get_Y() = 88.

Unit Test # 10: POINT class normal constructor, POINT class data attribute setter methods, POINT class overloaded ostream operator method, and POINT class destructor.

POINT point = POINT(666,777);

```
output << point;
point.set_X(999);
output << point;
point.set_Y(-999);
output << point;
point.set_X(-1000);
output << point;
point.set_X(200);
output << point;
point.set_Y(-1000);
output << point;
point.set_Y(444);
output << point;
point.set_X(1000);
output << point;
point.set_Y(1000);
output << point;
```

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 666. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 777. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.

&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 999. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 777. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 999. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = -999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.

&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 999. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = -999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 200. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = -999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.

sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 200. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = -999. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 200. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 444. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).

X = 200. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 444. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----
this = 0x7ffe4927b250. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffe4927b250. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffe4927b254. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 200. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 444. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

End Of Program

[End of abridged plain-text content from POINT]

TRIANGLE

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/triangle_image.png

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/points_on_cartesian_grid.png

---

The C++ program featured in this tutorial web page demonstrates the concept of Object Oriented Programming (OOP). The program implements a user defined data type for instantiating TRIANGLE type objects. Each TRIANGLE type object represents three instances of the POINT class named A, B, and C which each represent a unique whole number coordinate pair. A TRIANGLE object can execute various functions including the ability to compute the area of the two-dimensional region whose boundaries are the line segments which connect the points which the three POINT type variables of the TRIANGLE object represent and the ability to compute the interior angle measurement of any one of the three interior angles of the triangle which the TRIANGLE object represents.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

---

class : object :: data_type : variable.

SOFTWARE_APPLICATION_COMPONENTS

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.cpp

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/triangle.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/triangle.cpp

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/triangle_class_tester.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/triangle_class_tester_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ code from the files named point.h, point.cpp, triangle.h, triangle.cpp, and triangle_class_tester.cpp into their own new text editor documents and save those documents using their corresponding file names:

point.h

point.cpp

triangle.h

triangle.cpp

triangle_class_tester.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ triangle_class_tester.cpp triangle.cpp point.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: Observe program results on the command line terminal and in the output file.

POINT_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the POINT class.

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.h

---

```
/**
 * file: point.h
 * type: C++ (header file)
 * author: karbytes
 * date: 07_JULY_2023
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#ifndef POINT_H // If point.h has not already been linked to a source file (.cpp),
#define POINT_H // then link this header file to the source file(s) which include this header file.

/* preprocessing directives */
#include < iostream > // library for defining objects which handle command line input and command line output
#include < fstream > // library for defining objects which handle file input and file output
#include < cmath > // library which defines various math functions such as square root (sqrt()) and sine (sin())
#include < string > // library which defines a sequence of text characters (i.e. char type values) as a string type variable
#define MINIMUM_X -999 // constant which represents minimum X value
```

```
#define MAXIMUM_X 999 // constant which represents maximum X value
#define MINIMUM_Y -999 // constant which represents minimum Y value
#define MAXIMUM_Y 999 // constant which represents maximum Y value
#define PI 3.14159 // constant which represents the approximate value of a circle's
circumference divided by that circle's diameter

/**
 * Define a class which is used to instantiate POINT type objects.
 *
 * (An object is a variable whose data type is user defined rather than native to the C++
programming language).
 *
 * A POINT object represents a whole number coordinate pair in the form (X,Y).
 *
 * X represents a specific whole number position along the x-axis (i.e. horizontal dimension) of a
two-dimensional Cartesian grid.
 * Y represents a specific whole number position along the y-axis (i.e. vertical dimension) of the
same two-dimensional Cartesian grid.
 *
 * X stores one integer value at a time which is no smaller than MINIMUM_X and which is no
larger than MAXIMUM_X.
 * Y stores one integer value at a time which is no smaller than MINIMUM_Y and which is no
larger than MAXIMUM_Y.
 */
class POINT
{
private:
   int X, Y; // data attributes
public:
   POINT(); // default constructor
   POINT(int X, int Y); // normal constructor
   POINT(const POINT & point); // copy constructor
   int get_X(); // getter method
   int get_Y(); // getter method
   bool set_X(int X); // setter method
   bool set_Y(int Y); // setter method
   double get_distance_from(POINT & point); // getter method
   double get_slope_of_line_to(POINT & point); // getter method
   void print(std::ostream & output = std::cout); // descriptor method
   friend std::ostream & operator << (std::ostream & output, POINT & point); // descriptor
method
   ~POINT(); // destructor
};
```

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.

POINT_CLASS_SOURCE_CODE

The following source code defines the functions of the POINT class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.cpp

---

```
/**
 * file: point.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#include "point.h" // Include the C++ header file which contains preprocessing directives,
variable declarations, and function prototypes for the POINT class.

/**
 * The default constructor method of the POINT class
 * instantiates POINT type objects
 * whose X value is initially set to 0 and
 * whose Y value is initially set to 0.
 *
 * The default constructor method of the POINT class is invoked
 * when a POINT type variable is declared as follows:
 *
 * // variable declaration example one
 * POINT point_0;
 *
 * // variable declaration example two
 * POINT point_1 = POINT();
 */
POINT::POINT()
{
```

```
	std::cout << "\n\nCreating the POINT type object whose memory address is " << this << "...";
	X = 0;
	Y = 0;
}

/**
 * The normal constructor method of the POINT class
 * instantiates POINT type objects
 * whose X value is set to the leftmost function input value (if that input value is no smaller than MINIMUM_X and no larger than MAXIMUM_X) and
 * whose Y value is set to the rightmost function input value (if that input value is no smaller than MINIMUM_Y and no larger than MAXIMUM_Y).
 *
 * If a function input value is out of its specified range, then set the corresponding int type property of this to 0.
 *
 * (The keyword this refers to the POINT object which is returned by this function).
 *
 * The normal constructor method of the POINT class is invoked when a POINT type variable is declared as follows:
 *
 * // variable definition example one
 * POINT point_0 = POINT(-55,84);
 *
 * // variable definition example two
 * POINT point_1 = POINT(3,-4);
 *
 * // variable definition example three
 * POINT point_2 = POINT(-1000, 999); // point_2 = POINT(0,999).
 *
 * // variable definition example four
 * POINT point_3 = POINT(1000,-999); // point_3 = POINT(0,-999).
 *
 * // variable definition example five
 * POINT point_4 = POINT(999,-1000); // point_4 = POINT(999,0).
 *
 * // variable definition example six
 * POINT point_5 = POINT(-999,1000); // point_5 = POINT(-999,0).
 */
POINT::POINT(int X, int Y)
{
	std::cout << "\n\nCreating the POINT type object whose memory address is " << this << "...";
```

```
	this -> X = ((X < MINIMUM_X) || (X > MAXIMUM_X)) ? 0 : X; // Set the X property of the POINT instance being created to 0 if the function input X value is out of range.
	this -> Y = ((Y < MINIMUM_Y) || (Y > MAXIMUM_Y)) ? 0 : Y; // Set the Y property of the POINT instance being created to 0 if the function input Y value is out of range.
}

/**
 * The copy constructor method of the POINT class
 * instantiates POINT type objects
 * whose X value is set to the X value of the input POINT object and
 * whose Y value is set to the Y value of the input POINT object.
 *
 * The copy constructor method of the POINT class is invoked when a POINT type variable is declared as follows:
 *
 * // variable definition example one
 * POINT point_0 = POINT(33,55);
 * POINT point_1 = POINT(point_0); // point_1 = POINT(33,55).
 *
 * // variable definition example two
 * POINT point_2 = POINT(point_1); // point_2 = POINT(33,55).
 */
POINT::POINT(const POINT & point)
{
	std::cout << "\n\nCreating the POINT type object whose memory address is " << this << "...";
	X = point.X;
	Y = point.Y;
}

/**
 * The getter method of the POINT class returns the value of the caller POINT object's X property.
 *
 * X is an int type variable which stores exactly one integer value at a time which is no smaller than MINIMUM_X and which is no larger than MAXIMUM_X.
 *
 * X represents a specific whole number position along the x-axis (i.e. horizontal dimension) of a two-dimensional Cartesian grid.
 */
int POINT::get_X()
{
	return X;
}
```

```
/**
 * The getter method of the POINT class returns the value of the caller POINT object's Y
property.
 *
 * Y is an int type variable which stores exactly one integer value at a time which is no smaller
than MINIMUM_Y and which is no larger than MAXIMUM_Y.
 *
 * Y represents a specific whole number position along the y-axis (i.e. vertical dimension) of a
two-dimensional Cartesian grid.
 */
int POINT::get_Y()
{
	return Y;
}

/**
 * The setter method of the POINT class sets the POINT object's X property to the function input
value
 * if that value is no smaller than MINIMUM_X and which is no larger than MAXIMUM_X.
 *
 * If the input value is in range, then return true.
 * Otherwise, do not change the caller POINT object's X value and return false.
 *
 * (The keyword this refers to the POINT object which calls this function).
 *
 * (X represents a specific whole number position along the x-axis (i.e. horizontal dimension) of
a two-dimensional Cartesian grid).
 */
bool POINT::set_X(int X)
{
	if ((X >= MINIMUM_X) && (X <= MAXIMUM_X))
	{
	this -> X = X;
	return true;
	}
	return false;
}

/**
 * The setter method of the POINT class sets the POINT object's Y property to the function input
value
 * if that value is no smaller than MINIMUM_Y and which is no larger than MAXIMUM_Y.
 *
```

```
 * If the input value is in range, then return true.
 * Otherwise, do not change the caller POINT object's Y value and return false.
 *
 * (The keyword this refers to the POINT object which calls this function).
 *
 * (Y represents a specific whole number position along the y-axis (i.e. vertical dimension) of a
two-dimensional Cartesian grid).
 */
bool POINT::set_Y(int Y)
{
	if ((Y >= MINIMUM_Y) && (Y <= MAXIMUM_Y))
	{
	this -> Y = Y;
	return true;
	}
	return false;
}

/**
 * The getter method of the POINT class returns the length of the shortest path
 * between the two-dimensional point represented by the the caller POINT object (i.e. this)
 * and the two-dimensional point represented by the input POINT object (i.e. point).
 *
 * Use the Pythagorean Theorem to compute the length of a right triangle's hypotenuse
 * such that the two end points of that hypotenuse are represented by this and point.
 *
 * (A hypotenuse is the only side of a right triangle which does not form a right angle
 * with any other side of that triangle).
 *
 * (A hypotenuse is the longest side of a triangle (and a triangle is a three-sided polygon
 * in which three unique line segments connect three unique points)).
 *
 * // c is the length of a right triangle's hypotenuse.
 * // a is the length of that right triangle's horizontal leg.
 * // b is the length of that triangle's vertical leg.
 * (c * c) = (a * a) + (b * b).
 *
 * // sqrt() is a native C++ function defined in the cmath library.
 * c = square_root( (a * a) + (b * b)).
 */
double POINT::get_distance_from(POINT & point)
{
	int horizontal_difference = 0.0, vertical_difference = 0.0;
	horizontal_difference = X - point.X; // a
```

```
	vertical_difference = Y - point.Y; // b
	return sqrt((horizontal_difference * horizontal_difference) + (vertical_difference * 
vertical_difference)); // c
}

/**
 * The getter method of the POINT class returns the slope of the line which intersects
 * the two-dimensional point represented by the caller POINT instance (i.e. this)
 * and the two-dimensional point represented by the input POINT instance (i.e. point).
 *
 * // y := f(x),
 * // b := f(0),
 * // f is a function whose input is an x-axis position and whose output is a y-axis position.
 * y := mx + b.
 *
 * // m is a constant which represents the rate at which y changes in relation to x changing.
 * m := (y - b) / x.
 *
 * // m represents the difference of the two y-values divided by the difference of the two x-values.
 * m := (point.Y - this.Y) / (point.X - this.X).
 */
double POINT::get_slope_of_line_to(POINT & point)
{
	double vertical_difference = 0.0, horizontal_difference = 0.0, result = 0.0;
	vertical_difference = point.Y - Y;
	horizontal_difference = point.X - X;
	result = vertical_difference / horizontal_difference;
	if (result == -0) result = 0; // Signed zeros sometimes occur inside of C++ program 
runtime instances.
	return result;
}

/**
 * The print method of the POINT class prints a description of the caller POINT object to the 
output stream.
 *
 * Note that the default value of the function input parameter is the standard command line 
output stream (std::cout).
 *
 * The default parameter is defined in the POINT class header file (i.e. point.h) and not in the 
POINT class source file (i.e. point.cpp).
 */
void POINT::print(std::ostream & output)
{
```

```
	output << "\n\n--------------------------------------------------------------------------------------------";
	output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.";
	output << "\n&X = " << &X << ". // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.";
	output << "\n&Y = " << &Y << ". // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.";
	output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(POINT) = " << sizeof(POINT) << ". // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nX = " << X << ". // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.";
	output << "\nY = " << Y << ". // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.";
	output << "\n--------------------------------------------------------------------------------------------";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (<<).
 *
 * (Overloading an operator is assigning a different function to a native operator other than the function which that operator is used to represent by default).
 *
 * Note that the default value of the leftmost function input parameter is the standard command line output stream (std::cout).
 * The default parameter is defined in the POINT class header file (i.e. point.h).
 *
 * The friend function is not a member of the POINT class,
 * but the friend function has access to the private and protected members
 * of the POINT class and not just to the public members of the POINT class.
 *
 * The friend keyword only prefaces the function prototype of this function
 * (and the prototype of this function is declared in the POINT class header file (i.e. point.h)).
 *
 * The friend keyword does not preface the definition of this function
 * (and the definition of this function is specified in the POINT class source file (i.e. point.cpp)).
```

```
 *
 * // overloaded print function example one
 * POINT point_0;
 * std::cout << point_0; // identical to point_0.print();
 *
 * // overloaded print function example two
 * std::ofstream file;
 * POINT point_1;
 * file << point_1; // identical to point_1.print(file);
 */
std::ostream & operator << (std::ostream & output, POINT & point)
{
	point.print(output);
	return output;
}

/**
 * The destructor method of the POINT class de-allocates memory which was used to
 * instantiate the POINT object which is calling this function.
 *
 * The destructor method of the POINT class is automatically called when
 * the program scope in which the caller POINT object was instantiated terminates.
 */
POINT::~POINT()
{
	std::cout << "\n\nDeleting the POINT type object whose memory address is " << this <<
"...";
}
```

TRIANGLE_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the TRIANGLE class.

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/triangle.h

---

```
/**
 * file: triangle.h
```

```
 * type: C++ (header file)
 * author: karbytes
 * date: 07_JULY_2023
 * license: PUBLIC_DOMAIN
 */

// If TRIANGLE.h has not already been linked to a source file (.cpp), then link this header file to the source file(s) which include this header file.
#ifndef TRIANGLE_H // If triangle.h has not already been linked to a source file (.cpp),
#define TRIANGLE_H // then link this header file to the source file(s) which include this header file.

// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the POINT class.
#include "point.h"

/**
 * Define a class which is used to instantiate TRIANGLE type objects.
 *
 * (An object is a variable whose data type is user defined rather than native to the C++ programming language).
 *
 * A TRIANGLE object represents an instance in which three unique POINT instances exist
 * (and such that each one of those three POINT instances represents a unique coordinate pair within the tuple of three objects
 *  which each represent exactle one two-dimensional point, POINT(X,Y), on a Cartesian grid).
 *
 * A POINT object represents a whole number coordinate pair in the form (X,Y).
 *
 * X represents a specific whole number position along the x-axis (i.e. horizontal dimension) of a two-dimensional Cartesian grid.
 * Y represents a specific whole number position along the y-axis (i.e. vertical dimension) of the same two-dimensional Cartesian grid.
 *
 * X stores one integer value at a time which is no smaller than MINIMUM_X and which is no larger than MAXIMUM_X.
 * Y stores one integer value at a time which is no smaller than MINIMUM_Y and which is no larger than MAXIMUM_Y.
 */
class TRIANGLE
{
private:
	POINT A, B, C; // data attributes
```

```
	bool points_represent_unique_coordinate_pairs(POINT point_0, POINT point_1, POINT point_2); // helper method
	bool points_form_nondegenerate_triangle(POINT point_0, POINT point_1, POINT point_2); // helper method
public:
	TRIANGLE(); // default constructor
	TRIANGLE(POINT A, POINT B, POINT C); // normal constructor
	TRIANGLE(int A_X, int A_Y, int B_X, int B_Y, int C_X, int C_Y); // normal constructor
	TRIANGLE(const TRIANGLE & triangle); // copy constructor
	POINT get_A(); // getter method
	POINT get_B(); // getter method
	POINT get_C(); // getter method
	double get_side_length_AB(); // getter method
	double get_side_length_BC(); // getter method
	double get_side_length_CA(); // getter method
	double get_interior_angle_ABC(); // getter method
	double get_interior_angle_BCA(); // getter method
	double get_interior_angle_CAB(); // getter method
	double get_perimeter(); // getter method
	double get_area(); // getter method
	void print(std::ostream & output = std::cout); // descriptor method
	friend std::ostream & operator << (std::ostream & output, TRIANGLE & triangle); // descriptor method
	~TRIANGLE(); // destructor
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

TRIANGLE_CLASS_SOURCE_CODE

The following source code defines the functions of the TRIANGLE class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/triangle.cpp

---

```
/**
 * file: triangle.cpp
 * type: C++ (source file)
```

```
 * author: karbytes
 * date: 07_JULY_2023
 * license: PUBLIC_DOMAIN
 */

// Include the C++ header file which contains preprocessing directives, variable declarations,
and function prototypes for the TRIANGLE class.
#include "triangle.h"

/**
 * Determine whether or not point_0, point_1, and point_2 each represent unique coordinate
pairs.
 *
 * (Assume that point_0, point_1, and point_2 each represent valid POINT instances).
 *
 * If each of the three POINT objects represent unique coordinate pairs, return true.
 *
 * Otherwise, return false.
 */
bool TRIANGLE::points_represent_unique_coordinate_pairs(POINT point_0, POINT point_1,
POINT point_2)
{
	if ((point_0.get_X() == point_1.get_X()) && (point_0.get_Y() == point_1.get_Y()))
	{
	std::cout << "\n\npoint_0 and point_1 appear to represent the same coordinate pair.";
	std::cout << "\npoint_0 := POINT(" << point_0.get_X() << ", " << point_0.get_Y() << ").";
	std::cout << "\npoint_1 := POINT(" << point_1.get_X() << ", " << point_1.get_Y() << ").";
	return false;
	}
	if ((point_0.get_X() == point_2.get_X()) && (point_0.get_Y() == point_2.get_Y()))
	{
	std::cout << "\n\npoint_0 and point_2 appear to represent the same coordinate pair.";
	std::cout << "\npoint_0 := POINT(" << point_0.get_X() << ", " << point_0.get_Y() << ").";
	std::cout << "\npoint_2 := POINT(" << point_2.get_X() << ", " << point_2.get_Y() << ").";
	return false;
	}
	if ((point_1.get_X() == point_2.get_X()) && (point_1.get_Y() == point_2.get_Y()))
	{
	std::cout << "\n\npoint_1 and point_2 appear to represent the same coordinate pair.";
	std::cout << "\npoint_1 := POINT(" << point_1.get_X() << ", " << point_1.get_Y() << ").";
	std::cout << "\npoint_2 := POINT(" << point_2.get_X() << ", " << point_2.get_Y() << ").";
	return false;
	}
	if ((point_2.get_X() == point_0.get_X()) && (point_2.get_Y() == point_0.get_Y()))
```

```
	{
	std::cout << "\n\npoint_2 and point_0 appear to represent the same coordinate pair.";
	std::cout << "\npoint_2 := POINT(" << point_2.get_X() << ", " << point_2.get_Y() << ").";
	std::cout << "\npoint_0 := POINT(" << point_0.get_X() << ", " << point_0.get_Y() << ").";
	return false;
	}
	return true;
}

/**
 * Determine whether or not point_0, point_1, and point_2 form a non-degenerate triangle.
 *
 * (Assume that point_0, point_1, and point_2 each represent valid POINT instances).
 *
 * A non-degenerate triangle is a triangle whose area is some positive real number quantity.
 *
 * A degenerate triangle is a triangle whose area is zero (due to the fact that one line intersects
each of the three points).
 *
 * If point_0, point_1, and point_2 form a non-degenerate triangle, return true.
 *
 * Otherwise, return false.
 */
bool TRIANGLE::points_form_nondegenerate_triangle(POINT point_0, POINT point_1, POINT
point_2)
{
	if (!points_represent_unique_coordinate_pairs(point_0, point_1, point_2))
	{
	std::cout << "\n\npoint_0, point_1, and point_2 do not each represent unique coordinate
pairs.";
	std::cout << "\nHence, points_form_degenerate_triangle(point_0, point_1, point_2) is
returning false.";
	return false;
	}
	A = point_0;
	B = point_1;
	C = point_2;
	if (get_area() <= 0)
	{
	std::cout << "\n\nWhen setting the POINT values of the caller TRIANGLE object using
the given inputs, get_area() returned a non-positive number result.";
	std::cout << "\nHence, points_form_nondegenerate_triangle(POINT point_0, POINT
point_1, POINT point_2) is returning false.";
	return false;
```

```
	}
	return true;
}

/**
 * The default constructor method of the TRIANGLE class returns a TRIANGLE object
 * whose POINT property named A represents the coordinate pair (0, 0),
 * whose POINT property named B represents the coordinate pair (0, 1), and
 * whose POINT property named C represents the coordinate pair (1, 0).
 */
TRIANGLE::TRIANGLE()
{
	std::cout << "\n\nCreating the TRIANGLE type object whose memory address is " << this
<< "...";
	A = POINT(0, 0);
	B = POINT(0, 1);
	C = POINT(1, 0);
}

/**
 * The normal constructor method of the TRIANGLE class (which takes six int type values as
function inputs) returns a TRIANGLE object
 * whose POINT property named A represents the coordinate pair (A_X, A_Y),
 * whose POINT property named B represents the coordinate pair (B_X, B_Y), and
 * whose POINT property named C represents the coordinate pair (C_X, C_Y)
 * if POINT(A_X, A_Y), POINT(B_X, B_Y), and POINT(C_X, C_Y) represent a non-degenerate
triangle.
 *
 * If POINT(A_X, A_Y, POINT(B_X, B_Y), and POINT(C_X, C_Y) do not represent a
non-degenerate triangle,
 * this function will return a TRIANGLE object
 * whose POINT property named A represents the coordinate pair (0, 0),
 * whose POINT property named B represents the coordinate pair (0, 1), and
 * whose POINT property named C represents the coordinate pair (1, 0).
 */
TRIANGLE::TRIANGLE(int A_X, int A_Y, int B_X, int B_Y, int C_X, int C_Y)
{
	std::cout << "\n\nCreating the TRIANGLE type object whose memory address is " << this
<< "...";
	POINT input_A = POINT(A_X, A_Y);
	POINT input_B = POINT(B_X, B_Y);
	POINT input_C = POINT(C_X, C_Y);
	if (points_form_nondegenerate_triangle(input_A, input_B, input_C))
	{
```

```
		A = input_A;
		B = input_B;
		C = input_C;
		}
		else
		{
		A = POINT(0, 0);
		B = POINT(0, 1);
		C = POINT(1, 0);
		}
}

/**
 * The normal constructor method of the TRIANGLE class (which takes three POINT objects as function inputs) returns a TRIANGLE object
 * whose POINT property named A represents the same coordinate pair as the parameter named A,
 * whose POINT property named B represents the same coordinate pair as the parameter named B, and
 * whose POINT property named C represents the same coordinate pair as the parameter named C
 * if parameter A, parameter B, and parameter C represent a non-degenerate triangle.
 *
 * If parameter A, parameter B, and parameter C do not represent a non-degenerate triangle,
 * this function will return a TRIANGLE object
 * whose POINT property named A represents the coordinate pair (0, 0),
 * whose POINT property named B represents the coordinate pair (0, 1), and
 * whose POINT property named C represents the coordinate pair (1, 0).
 *
 * (The keyword this refers to the TRIANGLE object which is returned by the TRIANGLE(POINT A, POINT B, POINT C) method of the TRIANGLE class).
 */
TRIANGLE::TRIANGLE(POINT A, POINT B, POINT C)
{
		std::cout << "\n\nCreating the TRIANGLE type object whose memory address is " << this
< A = A;
		this -> B = B;
		this -> C = C;
		}
		else
		{
		this -> A = POINT(0, 0);
		this -> B = POINT(0, 1);
		this -> C = POINT(1, 0);
```

```
	}
}

/**
 * The copy constructor method of the TRIANGLE class returns a TRIANGLE object
 * whose POINT property named A represents the same coordinate pair as the POINT property named A which belongs to the input TRIANGLE object,
 * whose POINT property named B represents the same coordinate pair as the POINT property named B which belongs to the input TRIANGLE object, and
 * whose POINT property named C represents the same coordinate pair as the POINT property named C which belongs to the input TRIANGLE object.
 *
 * (The keyword this refers to the TRIANGLE object which is returned by the copy constructor method of the TRIANGLE class).
 */
TRIANGLE::TRIANGLE(const TRIANGLE & triangle)
{
	std::cout << "\n\nCreating the TRIANGLE type object whose memory address is " << this
< A = triangle.A;
	this -> B = triangle.B;
	this -> C = triangle.C;
}

/**
 * The getter method of the TRIANGLE class named get_A() returns the POINT type value of the caller TRIANGLE object's A property.
 */
POINT TRIANGLE::get_A()
{
	return A;
}

/**
 * The getter method of the TRIANGLE class named get_B() returns the POINT type value of the caller TRIANGLE object's B property.
 */
POINT TRIANGLE::get_B()
{
	return B;
}

/**
 * The getter method of the TRIANGLE class named get_C() returns the POINT type value of the caller TRIANGLE object's C property.
```

```
 */
POINT TRIANGLE::get_C()
{
	return C;
}

/**
 * The getter method of the TRIANGLE class named get_side_length_AB() returns the approximate length of the shortest path between points A and B.
 */
double TRIANGLE::get_side_length_AB()
{
	return A.get_distance_from(B);
}

/**
 * The getter method of the TRIANGLE class named get_side_length_BC() returns the approximate length of the shortest path between points B and C.
 */
double TRIANGLE::get_side_length_BC()
{
	return B.get_distance_from(C);
}

/**
 * The getter method of the TRIANGLE class named get_side_length_CA() returns the approximate length of the shortest path between points C and A.
 */
double TRIANGLE::get_side_length_CA()
{
	return C.get_distance_from(A);
}

/**
 * The getter method of the TRIANGLE class named get_interior_angle_ABC() returns the approximate angle measurement in degrees of the angle
 * formed by connecting points A, B, anc C in the order specified by this sentence.
 *
 * The function below uses the Law of Cosines to compute the measurement of an interior angle of a triangle
 * using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
 */
double TRIANGLE::get_interior_angle_ABC()
```

{
	double a = 0.0, b = 0.0, c = 0.0, angle_opposite_of_a = 0.0, angle_opposite_of_b = 0.0, angle_opposite_of_c = 0.0;
	a = get_side_length_BC(); // a represents the length of the line segment whose endpoints are B and C.
	b = get_side_length_CA(); // b represents the length of the line segment whose endpoints are C and A.
	c = get_side_length_AB(); // c represents the length of the line segment whose endpoints are A and B.
	angle_opposite_of_a = acos(((b * b) + (c * c) - (a * a)) / (2 * b * c)) * (180 / PI);
	angle_opposite_of_b = acos(((a * a) + (c * c) - (b * b)) / (2 * a * c)) * (180 / PI);
	angle_opposite_of_c = acos(((a * a) + (b * b) - (c * c)) / (2 * a * b)) * (180 / PI);
	return angle_opposite_of_b;
}

/**
 * The getter method of the TRIANGLE class named get_interior_angle_BCA() returns the approximate angle measurement in degrees of the angle
 * formed by connecting points B, C, and A in the order specified by this sentence.
 *
 * The function below uses the Law of Cosines to compute the measurement of an interior angle of a triangle
 * using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
 */
double TRIANGLE::get_interior_angle_BCA()
{
	double a = 0.0, b = 0.0, c = 0.0, angle_opposite_of_a = 0.0, angle_opposite_of_b = 0.0, angle_opposite_of_c = 0.0;
	a = get_side_length_BC(); // a represents the length of the line segment whose endpoints are B and C.
	b = get_side_length_CA(); // b represents the length of the line segment whose endpoints are C and A.
	c = get_side_length_AB(); // c represents the length of the line segment whose endpoints are A and B.
	angle_opposite_of_a = acos(((b * b) + (c * c) - (a * a)) / (2 * b * c)) * (180 / PI);
	angle_opposite_of_b = acos(((a * a) + (c * c) - (b * b)) / (2 * a * c)) * (180 / PI);
	angle_opposite_of_c = acos(((a * a) + (b * b) - (c * c)) / (2 * a * b)) * (180 / PI);
	return angle_opposite_of_c;
}

/**
 * The getter method of the TRIANGLE class named get_interior_angle_CAB() returns the approximate angle measurement in degrees of the angle

```
 * formed by connecting points C, A, and B in the order specified by this sentence.
 *
 * The function below uses Law of Cosines to compute the measurement of an interior angle of a triangle
 * using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
 */
double TRIANGLE::get_interior_angle_CAB()
{
	double a = 0.0, b = 0.0, c = 0.0, angle_opposite_of_a = 0.0, angle_opposite_of_b = 0.0, angle_opposite_of_c = 0.0;
	a = get_side_length_BC(); // a represents the length of the line segment whose endpoints are B and C (and which are points of the caller TRIANGLE object of this function represents).
	b = get_side_length_CA(); // b represents the length of the line segment whose endpoints are C and A (and which are points of the caller TRIANGLE object of this function represents).
	c = get_side_length_AB(); // c represents the length of the line segment whose endpoints are A and B (and which are points of the caller TRIANGLE object of this function represents).
	angle_opposite_of_a = acos(((b * b) + (c * c) - (a * a)) / (2 * b * c)) * (180 / PI);
	angle_opposite_of_b = acos(((a * a) + (c * c) - (b * b)) / (2 * a * c)) * (180 / PI);
	angle_opposite_of_c = acos(((a * a) + (b * b) - (c * c)) / (2 * a * b)) * (180 / PI);
	return angle_opposite_of_a;
}

/**
 * The getter method of the TRIANGLE class named get_perimeter() returns the approximate sum of the three side lengths
 * of the triangle which the caller TRIANGLE object represents.
 */
double TRIANGLE::get_perimeter()
{
	return get_side_length_AB() + get_side_length_BC() + get_side_length_CA();
}

/**
 * The getter method of the TRIANGLE class named get_area() returns the approximate area of the two-dimensional space whose bounds are
 * the shortest paths between points A, B, and C of the triangle which the caller TRIANGLE object represents.
 *
 * This function uses Heron's Formula to compute the area of a triangle using that triangle's side lengths as formula inputs.
```

```
 */
double TRIANGLE::get_area()
{
	double s = 0.0, a = 0.0, b = 0.0, c = 0.0;
	s = get_perimeter() / 2; // s is technically referred to as the semiperimter of the triangle
which the caller TRIANGLE object of this function represents.
	a = get_side_length_BC(); // a represents the length of the line segment whose
endpoints are B and C (and which are points of the caller TRIANGLE object of this function
represents).
	b = get_side_length_CA(); // b represents the length of the line segment whose
endpoints are C and A (and which are points of the caller TRIANGLE object of this function
represents).
	c = get_side_length_AB(); // c represents the length of the line segment whose
endpoints are A and B (and which are points of the caller TRIANGLE object of this function
represents).
	return sqrt(s * (s - a) * (s - b) * (s - c)); // Use Heron's Formula to compute the area of the
triangle whose points are A, B, and C (and which are points of the caller TRIANGLE object of
this function represents).
}

/**
 * The print method of the TRIANGLE class prints a description of the caller TRIANGLE object to
the output stream.
 *
 * Note that the default value of the function input parameter is the standard command line
output stream (std::cout).
 *
 * The default parameter is defined in the TRIANGLE class header file (i.e. triangle.h) and not in
the TRIANGLE class source file (i.e. triangle.cpp).
 */
void TRIANGLE::print(std::ostream & output)
{
	output <<
"\n\n----------------------------------------------------------------------------------------------------";
	output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the
memory address of the first memory cell of a TRIANGLE sized chunk of contiguous memory
cells which are allocated to the caller TRIANGLE object.";
	output << "\n&A = " << &A << ". // The reference operation returns the memory address
of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated
to the POINT data attribute named A.";
	output << "\n&B = " << &B << ". // The reference operation returns the memory address
of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated
to the POINT data attribute named B.";
```

output << "\n&C = " << &C << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.";

output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT) = " << sizeof(POINT) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(TRIANGLE) = " << sizeof(TRIANGLE) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRIANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nA = POINT(" << A.get_X() << "," << A.get_Y() << "). // A represents a point (which is neither B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nB = POINT(" << B.get_X() << "," << B.get_Y() << "). // B represents a point (which is neither A nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nC = POINT(" << C.get_X() << "," << C.get_Y() << "). // C represents a point (which is neither A nor B) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\na = get_side_length_BC() = " << get_side_length_BC() << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.";

output << "\nb = get_side_length_CA() = " << get_side_length_CA() << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.";

output << "\nc = get_side_length_AB() = " << get_side_length_AB() << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.";

output << "\nA.get_slope_of_line_to(B) = " << A.get_slope_of_line_to(B) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.";

output << "\nB.get_slope_of_line_to(C) = " << B.get_slope_of_line_to(C) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\nC.get_slope_of_line_to(A) = " << C.get_slope_of_line_to(A) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.";

output << "\nget_interior_angle_CAB() = " << get_interior_angle_CAB() << ". // The method returns the approximate nonnegative real number angle measurement of the acute or

else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).";
    output << "\nget_interior_angle_ABC() = " << get_interior_angle_ABC() << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).";
    output << "\nget_interior_angle_BCA() = " << get_interior_angle_BCA() << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).";
    output << "\nget_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() = " << get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() << ". // sum of all three approximate interior angle measurements of the triangle represented by the caller TRIANGLE object (in degrees and not in radians)";
    output << "\nget_perimeter() = a + b + c = " << get_perimeter() << ". // The method returns the sum of the three approximated side lengths of the triangle which the caller TRIANGLE object represents.";
    output << "\nget_area() = " << get_area() << ". // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A, B, and C.";
    output << "\n--";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (<<).
 *
 * (Overloading an operator is assigning a different function to a native operator other than the function which that operator is used to represent by default).
 *
 * Note that the default value of the leftmost function input parameter is the standard command line output stream (std::cout).
 * The default parameter is defined in the TRIANGLE class header file (i.e. triangle.h).
 *
 * The friend function is not a member of the TRIANGLE class,
 * but the friend function has access to the private and protected members
 * of the TRIANGLE class and not just to the public members of the TRIANGLE class.
 *
 * The friend keyword only prefaces the function prototype of this function
 * (and the prototype of this function is declared in the TRIANGLE class header file (i.e. triangle.h)).

```
 *
 * The friend keyword does not preface the definition of this function
 * (and the definition of this function is specified in the TRIANGLE class source file (i.e.
triangle.cpp)).
 *
 * // overloaded print function example one
 * TRIANGLE triangle_0;
 * std::cout << triangle_0; // identical to triangle_0.print();
 *
 * // overloaded print function example two
 * std::ofstream file;
 * TRIANGLE triangle_1;
 * file << triangle_1; // identical to triangle_1.print(file);
 */
std::ostream & operator << (std::ostream & output, TRIANGLE & triangle)
{
	triangle.print(output);
	return output;
}

/**
 * The destructor method of the TRIANGLE class de-allocates memory which was used to
 * instantiate the TRIANGLE object which is calling this function.
 *
 * The destructor method of the TRIANGLE class is automatically called when
 * the program scope in which the caller TRIANGLE object was instantiated terminates.
 */
TRIANGLE::~TRIANGLE()
{
	std::cout << "\n\nDeleting the TRIANGLE type object whose memory address is " << this
<< "...";
}
```

PROGRAM_SOURCE_CODE

The following source code defines the client which implements the TRIANGLE class. The client executes a series of unit tests which demonstrate how the TRIANGLE class methods work.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/triangle_class_tester.cpp

```
/**
 * file: triangle_class_tester.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

#include "triangle.h" // Include the C++ header file which contains preprocessing directives,
variable declarations, and function prototypes for the TRIANGLE class.

/* function prototypes */
void unit_test_0(std::ostream & output);
void unit_test_1(std::ostream & output);
void unit_test_2(std::ostream & output);
void unit_test_3(std::ostream & output);
void unit_test_4(std::ostream & output);

// Unit Test # 0: TRIANGLE class default constructor, TRIANGLE class print method, and
TRIANGLE class destructor.
void unit_test_0(std::ostream & output)
{
    output << "\n\n-----------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 0: TRIANGLE class default constructor, TRIANGLE class print
method, and TRIANGLE class destructor.";
    output << "\n-----------------------------------------------------------------------------------------------";
    output << "\nTRIANGLE point;";
    output << "\ntriangle.print(output);";
    TRIANGLE triangle;
    triangle.print(output);
}

// Unit Test # 1: TRIANGLE class default constructor, TRIANGLE class overloaded ostream
operator method, TRIANGLE getter methods, and TRIANGLE class destructor.
void unit_test_1(std::ostream & output)
{
    output << "\n\n-----------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 1: TRIANGLE class default constructor, TRIANGLE class overloaded
ostream operator method, TRIANGLE getter methods, and TRIANGLE class destructor.";
    output << "\n-----------------------------------------------------------------------------------------------";
    output << "\nTRIANGLE triangle;";
```

```
    output << "\nPOINT copy_of_point_A = triangle.get_A();";
    output << "\nPOINT copy_of_point_B = triangle.get_B();";
    output << "\nPOINT copy_of_point_C = triangle.get_C();";
    output << "\noutput << triangle;";
    TRIANGLE triangle;
    POINT copy_of_point_A = triangle.get_A();
    POINT copy_of_point_B = triangle.get_B();
    POINT copy_of_point_C = triangle.get_C();
    output << triangle;
    output << "\n\ncopy_of_point_A.print(output);";
    copy_of_point_A.print(output);
    output << "\n\noutput << copy_of_point_A;";
    output << copy_of_point_A;
    output << "\n\ncopy_of_point_B.print(output);";
    copy_of_point_B.print(output);
    output << "\n\noutput << copy_of_point_B;";
    output << copy_of_point_B;
    output << "\n\ncopy_of_point_C.print(output);";
    copy_of_point_C.print(output);
    output << "\n\noutput << copy_of_point_C;";
    output << copy_of_point_C;
    output << "\n\ntriangle.get_side_length_AB() = " << triangle.get_side_length_AB() << ".";
    output << "\ntriangle.get_side_length_BC() = " << triangle.get_side_length_BC() << ".";
    output << "\ntriangle.get_side_length_CA() = " << triangle.get_side_length_CA() << ".";
    output << "\ntriangle.get_interior_angle_ABC() = " << triangle.get_interior_angle_ABC() <<
".";
    output << "\ntriangle.get_interior_angle_BCA() := " << triangle.get_interior_angle_BCA() <<
".";
    output << "\ntriangle.get_interior_angle_CAB() = " << triangle.get_interior_angle_CAB() <<
".";
    output << "\ntriangle.get_perimeter() = " << triangle.get_perimeter() << ".";
    output << "\ntriangle.get_area() = " << triangle.get_area() << ".";
}


// Unit Test # 2: TRIANGLE class normal constructors, TRIANGLE class copy constructor,
TRIANGLE class print method, and TRIANGLE class destructor.
void unit_test_2(std::ostream & output)
{
    output << "\n\n----------------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 2: TRIANGLE class normal constructors, TRIANGLE class copy
constructor, TRIANGLE class print method, and TRIANGLE class destructor.";
    output << "\n----------------------------------------------------------------------------------------------------";
```

```
    output << "\nTRIANGLE triangle_0 = TRIANGLE(-1, -1, 0, 5, 2, -5); // normal constructor
which takes exactly 6 int type values as function inputs";
    output << "\nTRIANGLE triangle_1 = TRIANGLE( POINT(-3,-3), POINT(-4,-8), POINT(0,1) );
// normal constructor which takes 3 POINT type values as function inputs";
    output << "\nTRIANGLE triangle_2 = TRIANGLE(triangle_0); // copy constructor which takes
1 TRIANGLE type value as function input";
    output << "\ntriangle_0.print(output);";
    output << "\ntriangle_1.print(output);";
    output << "\ntriangle_2.print(output);";
    TRIANGLE triangle_0 = TRIANGLE(-1, -1, 0, 5, 2, -5); // normal constructor which takes
exactly 6 int type values as function inputs
    TRIANGLE triangle_1 = TRIANGLE( POINT(-3,-3), POINT(-4,-8), POINT(0,1) ); // normal
constructor which takes 3 POINT type values as function inputs
    TRIANGLE triangle_2 = TRIANGLE(triangle_0); // copy constructor which takes 1 TRIANGLE
type value as function input
    triangle_0.print(output);
    triangle_1.print(output);
    triangle_2.print(output);
}

// Unit Test # 3: degenerate triangle examples.
void unit_test_3(std::ostream & output)
{
    output << "\n\n--------------------------------------------------------------------------------------------------";
    output << "\nUnit Test # 3: degenerate triangle examples.";
    output << "\n--------------------------------------------------------------------------------------------------";
    output << "\nTRIANGLE triangle_0 = TRIANGLE( POINT(-1,-1), POINT(0,0), POINT(1,1) ); //
Because these inputs would generate a degenerate triangle, default coordinate values are used
for A, B, and C instead of the input values.";
    output << "\nTRIANGLE triangle_1 = TRIANGLE( POINT(-1,-1), POINT(0,0), POINT(-1,-1) );
// Because these inputs represent only 2 unique points instead of 3 unique points, default
coordinate values are used for A, B, and C instead of the input values.";
    output << "\ntriangle_0.print(output);";
    output << "\ntriangle_1.print(output);";
    TRIANGLE triangle_0 = TRIANGLE( POINT(-1,-1), POINT(0,0), POINT(1,1) ); // Because
these inputs would generate a degenerate triangle, default coordinate values are used for A, B,
and C instead of the input values.
    TRIANGLE triangle_1 = TRIANGLE(-1, -1, 0, 0, -1, -1); // Because these inputs represent
only 2 unique points instead of 3 unique points, default coordinate values are used for A, B, and
C instead of the input values.
    triangle_0.print(output);
    triangle_1.print(output);
}
```

```
// Unit Test # 4: Demonstrate how the methods of the POINT class cannot be called by a TRIANGLE object due to the fact that the methods of the POINT class each have uniquely corresponding function prototypes which are prefaced with the private access specifier in the POINT class header file (i.e. POINT.h).
void unit_test_4(std::ostream & output)
{
    output << "\n\n--------------------------------------------------------------------------------------------------";
    output << "\n// Unit Test # 4: Demonstrate how the methods of the POINT class cannot be called by a TRIANGLE object due to the fact that the methods of the POINT class each have uniquely corresponding function prototypes which are prefaced with the private access specifier in the POINT class header file (i.e. POINT.h).";
    output << "\n--------------------------------------------------------------------------------------------------";
    output << "\nTRIANGLE triangle;";
    output << "\ntriangle.print(output);";
    TRIANGLE triangle;
    triangle.print(output);
    output << "\nPOINT copy_A = triangle.get_A();";
    output << "\ncopy_A.print(output);";
    POINT copy_A = triangle.get_A();
    copy_A.print(output);
    copy_A.set_X(33); // The setter method of the POINT class is public. Therefore, that method can be invoked from the program scope in which the POINT type variable copy_A is instantiated.
    output << "\ncopy_A.set_X(33); // The setter method of the POINT class is public. Therefore, that method can be invoked from the program scope in which the POINT type variable copy_A is instantiated.";
    output << "\ncopy_A.print(output); // The print method of the POINT class is public. Therefore, that method can be invoked from the program scope in which the POINT type variable copy_A is insstantiated.";
    /*
    output << "\ntriangle.A.get_X() = " << triangle.A.get_X() << ". // Note that this command can only be executed if the POINT type data member named A of a TRIANGLE instance is public.";
    output << "\ntriangle.A.get_Y() = " << triangle.A.get_Y() << ". // Note that this command can only be executed if the POINT type data member named A of a TRIANGLE instance is public.";
    output << "\ntriangle.A.set_X(25) = " << triangle.A.set_X(25) << ". // Note that this command can only be executed if the POINT type data member named A of a TRIANGLE instance is public.";
    output << "\ntriangle.A.get_Y(666) = " << triangle.A.set_Y(666) << ". // Note that this command can only be executed if the POINT type data member named A of a TRIANGLE instance is public.";
    triangle.print(output);
    */
    output << "\n// COMMENTED OUT: triangle.A.get_X(); // Note that this command can only be executed if the POINT type data member named A of a TRIANGLE instance is public.";
```

```
    output << "\n// COMMENTED OUT: triangle.A.get_Y(); // Note that this command can only be
executed if the POINT type data member named A of a TRIANGLE instance is public.";
    output << "\n// COMMENTED OUT: triangle.A.set_X(25); // Note that this command can only
be executed if the POINT type data member named A of a TRIANGLE instance is public.";
    output << "\n// COMMENTED OUT: triangle.A.get_Y(666); // Note that this command can only
be executed if the POINT type data member named A of a TRIANGLE instance is public.";
}

/* program entry point */
int main()
{
    // Declare a file output stream object.
    std::ofstream file;

    // Set the number of digits of floating-point numbers which are printed to the command line
terminal to 100 digits.
    std::cout.precision(100);

    // Set the number of digits of floating-point numbers which are printed to the file output stream
to 100 digits.
    file.precision(100);

    /**
     * If triangle_class_tester_output.txt does not already exist in the same directory as
triangle_class_tester.cpp,
     * create a new file named triangle_class_tester_output.txt.
     *
     * Open the plain-text file named triangle_class_tester_output.txt
     * and set that file to be overwritten with program data.
     */
    file.open("triangle_class_tester_output.txt");

    // Print an opening message to the command line terminal.
    std::cout << "\n\n--------------------------------";
    std::cout << "\nStart Of Program";
    std::cout << "\n--------------------------------";

    // Print an opening message to the file output stream.
    file << "--------------------------------";
    file << "\nStart Of Program";
    file << "\n--------------------------------";

    // Implement a series of unit tests which demonstrate the functionality of TRIANGLE class
variables.
```

```
    unit_test_0(std::cout);
    unit_test_0(file);
    unit_test_1(std::cout);
    unit_test_1(file);
    unit_test_2(std::cout);
    unit_test_2(file);
    unit_test_3(std::cout);
    unit_test_3(file);
    unit_test_4(std::cout);
    unit_test_4(file);

    // Print a closing message to the command line terminal.
    std::cout << "\n\n--------------------------------";
    std::cout << "\nEnd Of Program";
    std::cout << "\n--------------------------------\n\n";

    // Print a closing message to the file output stream.
    file << "\n\n--------------------------------";
    file << "\nEnd Of Program";
    file << "\n--------------------------------";

    // Close the file output stream.
    file.close();

    // Exit the program.
    return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/triangle_class_tester_output.txt

---

```
--------------------------------
Start Of Program
--------------------------------
```

Unit Test # 0: TRIANGLE class default constructor, TRIANGLE class print method, and TRIANGLE class destructor.

TRIANGLE point;
triangle.print(output);

----

this = 0x7ffcc3aa1a50. // The keyword named this is a pointer which stores the memory address of the first memory cell of a TRIANGLE sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&A = 0x7ffcc3aa1a50. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffcc3aa1a58. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffcc3aa1a60. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRIANGLE) = 24. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRIANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
A = POINT(0,0). // A represents a point (which is neither B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = get_side_length_BC() = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.

b = get_side_length_CA() = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = get_side_length_AB() = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
get_interior_angle_CAB() = 90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
get_interior_angle_ABC() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
get_interior_angle_BCA() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the triangle represented by the caller TRIANGLE object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.41421356237309492343001693370752036571502685546875. // The method returns the sum of the three approximated side lengths of the triangle which the caller TRIANGLE object represents.
get_area() = 0.4999999999999997779553950749686919152736663818359375. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A, B, and C.

Unit Test # 1: TRIANGLE class default constructor, TRIANGLE class overloaded ostream operator method, TRIANGLE getter methods, and TRIANGLE class destructor.

TRIANGLE triangle;
POINT copy_of_point_A = triangle.get_A();
POINT copy_of_point_B = triangle.get_B();
POINT copy_of_point_C = triangle.get_C();
output << triangle;

----

this = 0x7ffcc3aa1a50. // The keyword named this is a pointer which stores the memory address of the first memory cell of a TRIANGLE sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&A = 0x7ffcc3aa1a50. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffcc3aa1a58. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffcc3aa1a60. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRIANGLE) = 24. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRIANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
A = POINT(0,0). // A represents a point (which is neither B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = get_side_length_BC() = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = get_side_length_CA() = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.

c = get_side_length_AB() = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
get_interior_angle_CAB() = 90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
get_interior_angle_ABC() = 45.00003800990602087495062733069062232971191406250. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
get_interior_angle_BCA() = 45.00003800990602087495062733069062232971191406250. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the triangle represented by the caller TRIANGLE object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.41421356237309492343001693370752036571502685546875. // The method returns the sum of the three approximated side lengths of the triangle which the caller TRIANGLE object represents.
get_area() = 0.499999999999999777955395074968691915273666381835937 5. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A, B, and C.
----

copy_of_point_A.print(output);

----
this = 0x7ffcc3aa1a38. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.

&X = 0x7ffcc3aa1a38. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffcc3aa1a3c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.
----

output << copy_of_point_A;

----
this = 0x7ffcc3aa1a38. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffcc3aa1a38. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffcc3aa1a3c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.
----

copy_of_point_B.print(output);

----
this = 0x7ffcc3aa1a40. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.

&X = 0x7ffcc3aa1a40. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffcc3aa1a44. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 1. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.
----

output << copy_of_point_B;

----
this = 0x7ffcc3aa1a40. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffcc3aa1a40. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffcc3aa1a44. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 1. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.
----

copy_of_point_C.print(output);

----
this = 0x7ffcc3aa1a48. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.

&X = 0x7ffcc3aa1a48. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffcc3aa1a4c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 1. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----

output << copy_of_point_C;

----

this = 0x7ffcc3aa1a48. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffcc3aa1a48. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffcc3aa1a4c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 1. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----

triangle.get_side_length_AB() = 1.
triangle.get_side_length_BC() = 1.4142135623730951454746218587388284504413604736328125.
triangle.get_side_length_CA() = 1.
triangle.get_interior_angle_ABC() = 45.0000380099060208749506273306906223297119140625.

triangle.get_interior_angle_BCA() :=
45.0000380099060208749506273306906223297119140625.
triangle.get_interior_angle_CAB() =
90.00007601981208438246540026739239692687988828125.
triangle.get_perimeter() = 3.41421356237309492343001693370752036571502685546875.
triangle.get_area() = 0.499999999999999777955395074968691915273666381835937 5.

Unit Test # 2: TRIANGLE class normal constructors, TRIANGLE class copy constructor, TRIANGLE class print method, and TRIANGLE class destructor.

TRIANGLE triangle_0 = TRIANGLE(-1, -1, 0, 5, 2, -5); // normal constructor which takes exactly 6 int type values as function inputs
TRIANGLE triangle_1 = TRIANGLE(POINT(-3,-3), POINT(-4,-8), POINT(0,1)); // normal constructor which takes 3 POINT type values as function inputs
TRIANGLE triangle_2 = TRIANGLE(triangle_0); // copy constructor which takes 1 TRIANGLE type value as function input
triangle_0.print(output);
triangle_1.print(output);
triangle_2.print(output);

----
this = 0x7ffcc3aa1a10. // The keyword named this is a pointer which stores the memory address of the first memory cell of a TRIANGLE sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&A = 0x7ffcc3aa1a10. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffcc3aa1a18. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffcc3aa1a20. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRIANGLE) = 24. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRIANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
A = POINT(-1,-1). // A represents a point (which is neither B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the

horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,5). // B represents a point (which is neither A nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(2,-5). // C represents a point (which is neither A nor B) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = get_side_length_BC() = 10.198039027185568983213670435361564159393310546875. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = get_side_length_CA() = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = get_side_length_AB() = 6.08276253029821933893117602565325796604156494140625. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = 6. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = -5. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(A) =
-1.3333333333333332593184650249895639717578887939453125. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
get_interior_angle_CAB() = 133.66789305056954617612063884735107421875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
get_interior_angle_ABC() = 20.772272227633603591812061495147645473480224609375. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
get_interior_angle_BCA() = 25.559986761421026102425457793287932872772216796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() =
180.000152039624168764930800534784793853759765625. // sum of all three approximate

interior angle measurements of the triangle represented by the caller TRIANGLE object (in degrees and not in radians)
get_perimeter() = a + b + c = 21.280801557483787433966426760889589786529541015625. // The method returns the sum of the three approximated side lengths of the triangle which the caller TRIANGLE object represents.
get_area() = 10.9999999999999946709294817992486059665679931640625. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A, B, and C.

----
this = 0x7ffcc3aa1a30. // The keyword named this is a pointer which stores the memory address of the first memory cell of a TRIANGLE sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&A = 0x7ffcc3aa1a30. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffcc3aa1a38. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffcc3aa1a40. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRIANGLE) = 24. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRIANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
A = POINT(-3,-3). // A represents a point (which is neither B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(-4,-8). // B represents a point (which is neither A nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(0,1). // C represents a point (which is neither A nor B) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).

a = get_side_length_BC() = 9.848857801796103927927106269635260105133056640625. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = get_side_length_CA() = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = get_side_length_AB() = 5.0990195135927844916068352176807820796966552734375. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = 5. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 2.25. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(A) = 1.3333333333333332593184650249895639717578887939453125. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
get_interior_angle_CAB() = 154.440165278203068055518087930977344512939453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
get_interior_angle_ABC() = 12.6525671877242711360622706706635653972625732421875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
get_interior_angle_BCA() = 12.907419573696753190006347722373902797698974609375. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() = 180.00015203962408349980250932276248931884765625. // sum of all three approximate interior angle measurements of the triangle represented by the caller TRIANGLE object (in degrees and not in radians)
get_perimeter() = a + b + c = 19.94787731538888664317710208706557750701904296875. // The method returns the sum of the three approximated side lengths of the triangle which the caller TRIANGLE object represents.
get_area() = 5.49999999999998312461002569762058556079864501953125. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A, B, and C.

----
this = 0x7ffcc3aa1a50. // The keyword named this is a pointer which stores the memory address of the first memory cell of a TRIANGLE sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&A = 0x7ffcc3aa1a50. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffcc3aa1a58. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffcc3aa1a60. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRIANGLE) = 24. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRIANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
A = POINT(-1,-1). // A represents a point (which is neither B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,5). // B represents a point (which is neither A nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(2,-5). // C represents a point (which is neither A nor B) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = get_side_length_BC() = 10.198039027185568983213670435361564159393310546875. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = get_side_length_CA() = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = get_side_length_AB() = 6.08276253029821933893117602565325796604156494140625. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = 6. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.

B.get_slope_of_line_to(C) = -5. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(A) = -1.3333333333333332593184650249895639717578887939453125. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
get_interior_angle_CAB() = 133.66789305056954617612063884735107421875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
get_interior_angle_ABC() = 20.772272227633603591812061495147645473480224609375. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
get_interior_angle_BCA() = 25.559986761421026102425457793287932872772216796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() = 180.000152039624168764930800534784793853759765625. // sum of all three approximate interior angle measurements of the triangle represented by the caller TRIANGLE object (in degrees and not in radians)
get_perimeter() = a + b + c = 21.280801557483787433966426760889589786529541015625. // The method returns the sum of the three approximated side lengths of the triangle which the caller TRIANGLE object represents.
get_area() = 10.9999999999999946709294817992486059665679931640625. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A, B, and C.

Unit Test # 3: degenerate triangle examples.

TRIANGLE triangle_0 = TRIANGLE(POINT(-1,-1), POINT(0,0), POINT(1,1)); // Because these inputs would generate a degenerate triangle, default coordinate values are used for A, B, and C instead of the input values.
TRIANGLE triangle_1 = TRIANGLE(POINT(-1,-1), POINT(0,0), POINT(-1,-1)); // Because these inputs represent only 2 unique points instead of 3 unique points, default coordinate values are used for A, B, and C instead of the input values.
triangle_0.print(output);

triangle_1.print(output);

----
this = 0x7ffcc3aa1a30. // The keyword named this is a pointer which stores the memory address of the first memory cell of a TRIANGLE sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&A = 0x7ffcc3aa1a30. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffcc3aa1a38. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffcc3aa1a40. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRIANGLE) = 24. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRIANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
A = POINT(0,0). // A represents a point (which is neither B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = get_side_length_BC() = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = get_side_length_CA() = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = get_side_length_AB() = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.

A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
get_interior_angle_CAB() = 90.00007601981208438246540026739239692687988281 25. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
get_interior_angle_ABC() = 45.000038009906020874950627330690622329711914 0625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
get_interior_angle_BCA() = 45.000038009906020874950627330690622329711914 0625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() = 180.000152039624140343221370130777359008789 0625. // sum of all three approximate interior angle measurements of the triangle represented by the caller TRIANGLE object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.41421356237309492343001693370752036571502685546875. // The method returns the sum of the three approximated side lengths of the triangle which the caller TRIANGLE object represents.
get_area() = 0.49999999999999997779553950749686919152736663818359375. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A, B, and C.

----
this = 0x7ffcc3aa1a50. // The keyword named this is a pointer which stores the memory address of the first memory cell of a TRIANGLE sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&A = 0x7ffcc3aa1a50. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffcc3aa1a58. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.

&C = 0x7ffcc3aa1a60. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRIANGLE) = 24. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRIANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
A = POINT(0,0). // A represents a point (which is neither B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = get_side_length_BC() = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = get_side_length_CA() = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = get_side_length_AB() = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
get_interior_angle_CAB() = 90.00007601981208438246540026739239692687988828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
get_interior_angle_ABC() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute

or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
get_interior_angle_BCA() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the triangle represented by the caller TRIANGLE object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.41421356237309492343001693370752036571502685546875. // The method returns the sum of the three approximated side lengths of the triangle which the caller TRIANGLE object represents.
get_area() = 0.499999999999999777955395074968691915273666381835937 5. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A, B, and C.

// Unit Test # 4: Demonstrate how the methods of the POINT class cannot be called by a TRIANGLE object due to the fact that the methods of the POINT class each have uniquely corresponding function prototypes which are prefaced with the private access specifier in the POINT class header file (i.e. POINT.h).

TRIANGLE triangle;
triangle.print(output);

----
this = 0x7ffcc3aa1a50. // The keyword named this is a pointer which stores the memory address of the first memory cell of a TRIANGLE sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&A = 0x7ffcc3aa1a50. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffcc3aa1a58. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffcc3aa1a60. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.

sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRIANGLE) = 24. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRIANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
A = POINT(0,0). // A represents a point (which is neither B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = get_side_length_BC() = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = get_side_length_CA() = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = get_side_length_AB() = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
get_interior_angle_CAB() = 90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
get_interior_angle_ABC() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).

get_interior_angle_BCA() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the triangle represented by the caller TRIANGLE object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.41421356237309492343001693370752036571502685546875. // The method returns the sum of the three approximated side lengths of the triangle which the caller TRIANGLE object represents.
get_area() = 0.49999999999999977795539507496869191527366638183359375. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A, B, and C.

POINT copy_A = triangle.get_A();
copy_A.print(output);

----

this = 0x7ffcc3aa1a48. // The keyword named this is a pointer which stores the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the caller POINT object.
&X = 0x7ffcc3aa1a48. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.
&Y = 0x7ffcc3aa1a4c. // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
X = 0. // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.
Y = 0. // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.

----

copy_A.set_X(33); // The setter method of the POINT class is public. Therefore, that method can be invoked from the program scope in which the POINT type variable copy_A is instantiated.

copy_A.print(output); // The print method of the POINT class is public. Therefore, that method can be invoked from the program scope in which the POINT type variable copy_A is insstantiated.
// COMMENTED OUT: triangle.A.get_X(); // Note that this command can only be executed if the POINT type data member named A of a TRIANGLE instance is public.
// COMMENTED OUT: triangle.A.get_Y(); // Note that this command can only be executed if the POINT type data member named A of a TRIANGLE instance is public.
// COMMENTED OUT: triangle.A.set_X(25); // Note that this command can only be executed if the POINT type data member named A of a TRIANGLE instance is public.
// COMMENTED OUT: triangle.A.get_Y(666); // Note that this command can only be executed if the POINT type data member named A of a TRIANGLE instance is public.

End Of Program

[End of abridged plain-text content from TRIANGLE]

POLYGON

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/polygon_inheritance_diagram.png

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/quadrilateral_image.png

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/points_on_cartesian_grid.png

---

The C++ program featured in this tutorial web page demonstrates the concept of Object Oriented Programming (OOP); especially the concept of polymorphism (i.e. implementing data types which inherit abstract attributes which can be made more concrete and specific).

The program implements user defined data types for instantiating POLYGON type objects (and POLYGON is an abstract class) which are instances of either the POLYGON-derived non-abstract class named QUADRILATERAL or else the QUADRILATERAL-derived non-abstract class named TRAPEZOID or else the QUADRILATERAL-derived non-abstract class named RECTANGLE or else the RECTANGLE-derived non-abstract class named SQUARE or else the POLYGON-derived non-abstract class named TRILATERAL or else the TRILATERAL-derived non-abstract class named RIGHT_TRILATERAL.

Each non-abstract POLYGON type object object represents either three or else four instances of the POINT class such that each POINT instance represents a unique whole number coordinate pair and such that the polygon represented by the points whose coordinates are data members of the non-abstract POLYGON instance has a non-zero area and, if the polygon is an instance of TRILATERAL, the sum of the interior angles of that polygon is 180 degrees and, if the polygon is an instance of QUADRILATERAL, the sum of the interior angles of that polygon is 360 degrees.

A non-abstract POLYGON object can execute various functions including the ability to compute the area of the two-dimensional region whose boundaries are the line segments which connect the points which the POINT type variables of the non-abstract POLYGON object represent and the ability to compute the perimeter (i.e. length of the one-dimensional boundary of the two-dimensional area) which the non-abstract POLYGON object represents.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

---

class : object :: data_type : variable.

SOFTWARE_APPLICATION_COMPONENTS

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.cpp

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/polygon.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/polygon.cpp

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/quadrilateral.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/quadrilateral.cpp

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/trapezoid.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/trapezoid.cpp

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/rectangle.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/rectangle.cpp

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/square.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/square.cpp

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/trilateral.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/trilateral.cpp

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/right_trilateral.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/right_trilateral.cpp

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/polygon_class_inheritance_tester.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/polygon_class_inheritance_tester_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ code from the files named point.h, point.cpp, polygon.h, polygon.cpp, quadrilateral.h, quadrilateral.cpp, trapezoid.h, trapezoid.cpp, rectangle.h, rectangle.cpp, square.h, square.cpp, trilateral.h, trilateral.cpp, right_trilateral.h, right_trilateral.cpp, and polygon_class_inheritance_tester.cpp into their own new text editor documents and save those documents using their corresponding file names:

point.h

point.cpp

polygon.h

polygon.cpp

quadrilateral.h

quadrilateral.cpp

trapezoid.h

trapezoid.cpp

rectangle.h

rectangle.cpp

square.h

square.cpp

trilateral.h

trilateral.cpp

right_trilateral.h

right_trilateral.cpp

polygon_class_inheritance_tester.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ polygon_class_inherintance_tester.cpp right_trilateral.cpp trilateral.cpp square.cpp rectangle.cpp trapezoid.cpp quadrilateral.cpp polygon.cpp point.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: Observe program results on the command line terminal and in the output file.

POINT_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the POINT class.

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_header_file: https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.h

---

```
/**
 * file: point.h
 * type: C++ (header file)
 * author: karbytes
 * date: 07_JULY_2023
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#ifndef POINT_H // If point.h has not already been linked to a source file (.cpp),
#define POINT_H // then link this header file to the source file(s) which include this header file.

/* preprocessing directives */
#include < iostream > // library for defining objects which handle command line input and command line output
#include < fstream > // library for defining objects which handle file input and file output
#include < cmath > // library which defines various math functions such as square root (sqrt()) and sine (sin())
#include < string > // library which defines a sequence of text characters (i.e. char type values) as a string type variable
#define MINIMUM_X -999 // constant which represents minimum X value
#define MAXIMUM_X 999 // constant which represents maximum X value
#define MINIMUM_Y -999 // constant which represents minimum Y value
#define MAXIMUM_Y 999 // constant which represents maximum Y value
#define PI 3.14159 // constant which represents the approximate value of a circle's circumference divided by that circle's diameter
```

```
/**
 * Define a class which is used to instantiate POINT type objects.
 *
 * (An object is a variable whose data type is user defined rather than native to the C++
programming language).
 *
 * A POINT object represents a whole number coordinate pair in the form (X,Y).
 *
 * X represents a specific whole number position along the x-axis (i.e. horizontal dimension) of a
two-dimensional Cartesian grid.
 * Y represents a specific whole number position along the y-axis (i.e. vertical dimension) of the
same two-dimensional Cartesian grid.
 *
 * X stores one integer value at a time which is no smaller than MINIMUM_X and which is no
larger than MAXIMUM_X.
 * Y stores one integer value at a time which is no smaller than MINIMUM_Y and which is no
larger than MAXIMUM_Y.
 */
class POINT
{
private:
	int X, Y; // data attributes
public:
	POINT(); // default constructor
	POINT(int X, int Y); // normal constructor
	POINT(const POINT & point); // copy constructor
	int get_X(); // getter method
	int get_Y(); // getter method
	bool set_X(int X); // setter method
	bool set_Y(int Y); // setter method
	double get_distance_from(POINT & point); // getter method
	double get_slope_of_line_to(POINT & point); // getter method
	void print(std::ostream & output = std::cout); // descriptor method
	friend std::ostream & operator << (std::ostream & output, POINT & point); // descriptor
method
	~POINT(); // destructor
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

POINT_CLASS_SOURCE_CODE

The following source code defines the functions of the POINT class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/point.cpp

---

```
/**
 * file: point.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#include "point.h" // Include the C++ header file which contains preprocessing directives,
variable declarations, and function prototypes for the POINT class.

/**
 * The default constructor method of the POINT class
 * instantiates POINT type objects
 * whose X value is initially set to 0 and
 * whose Y value is initially set to 0.
 *
 * The default constructor method of the POINT class is invoked
 * when a POINT type variable is declared as follows:
 *
 * // variable declaration example one
 * POINT point_0;
 *
 * // variable declaration example two
 * POINT point_1 = POINT();
 */
POINT::POINT()
{
	std::cout << "\n\nCreating the POINT type object whose memory address is " << this <<
"...";
	X = 0;
	Y = 0;
}
```

```
/**
 * The normal constructor method of the POINT class
 * instantiates POINT type objects
 * whose X value is set to the leftmost function input value (if that input value is no smaller than MINIMUM_X and no larger than MAXIMUM_X) and
 * whose Y value is set to the rightmost function input value (if that input value is no smaller than MINIMUM_Y and no larger than MAXIMUM_Y).
 *
 * If a function input value is out of its specified range, then set the corresponding int type property of this to 0.
 *
 * (The keyword this refers to the POINT object which is returned by this function).
 *
 * The normal constructor method of the POINT class is invoked when a POINT type variable is declared as follows:
 *
 * // variable definition example one
 * POINT point_0 = POINT(-55,84);
 *
 * // variable definition example two
 * POINT point_1 = POINT(3,-4);
 *
 * // variable definition example three
 * POINT point_2 = POINT(-1000, 999); // point_2 = POINT(0,999).
 *
 * // variable definition example four
 * POINT point_3 = POINT(1000,-999); // point_3 = POINT(0,-999).
 *
 * // variable definition example five
 * POINT point_4 = POINT(999,-1000); // point_4 = POINT(999,0).
 *
 * // variable definition example six
 * POINT point_5 = POINT(-999,1000); // point_5 = POINT(-999,0).
 */
POINT::POINT(int X, int Y)
{
	std::cout << "\n\nCreating the POINT type object whose memory address is " << this << "...";
	this -> X = ((X < MINIMUM_X) || (X > MAXIMUM_X)) ? 0 : X; // Set the X property of the POINT instance being created to 0 if the function input X value is out of range.
	this -> Y = ((Y < MINIMUM_Y) || (Y > MAXIMUM_Y)) ? 0 : Y; // Set the Y property of the POINT instance being created to 0 if the function input Y value is out of range.
}
```

```
/**
 * The copy constructor method of the POINT class
 * instantiates POINT type objects
 * whose X value is set to the X value of the input POINT object and
 * whose Y value is set to the Y value of the input POINT object.
 *
 * The copy constructor method of the POINT class is invoked when a POINT type variable is
declared as follows:
 *
 * // variable definition example one
 * POINT point_0 = POINT(33,55);
 * POINT point_1 = POINT(point_0); // point_1 = POINT(33,55).
 *
 * // variable definition example two
 * POINT point_2 = POINT(point_1); // point_2 = POINT(33,55).
 */
POINT::POINT(const POINT & point)
{
	std::cout << "\n\nCreating the POINT type object whose memory address is " << this <<
"...";
	X = point.X;
	Y = point.Y;
}

/**
 * The getter method of the POINT class returns the value of the caller POINT object's X
property.
 *
 * X is an int type variable which stores exactly one integer value at a time which is no smaller
than MINIMUM_X and which is no larger than MAXIMUM_X.
 *
 * X represents a specific whole number position along the x-axis (i.e. horizontal dimension) of a
two-dimensional Cartesian grid.
 */
int POINT::get_X()
{
	return X;
}

/**
 * The getter method of the POINT class returns the value of the caller POINT object's Y
property.
 *
```

```
 * Y is an int type variable which stores exactly one integer value at a time which is no smaller
than MINIMUM_Y and which is no larger than MAXIMUM_Y.
 *
 * Y represents a specific whole number position along the y-axis (i.e. vertical dimension) of a
two-dimensional Cartesian grid.
 */
int POINT::get_Y()
{
	return Y;
}

/**
 * The setter method of the POINT class sets the POINT object's X property to the function input
value
 * if that value is no smaller than MINIMUM_X and which is no larger than MAXIMUM_X.
 *
 * If the input value is in range, then return true.
 * Otherwise, do not change the caller POINT object's X value and return false.
 *
 * (The keyword this refers to the POINT object which calls this function).
 *
 * (X represents a specific whole number position along the x-axis (i.e. horizontal dimension) of
a two-dimensional Cartesian grid).
 */
bool POINT::set_X(int X)
{
	if ((X >= MINIMUM_X) && (X <= MAXIMUM_X))
	{
	this -> X = X;
	return true;
	}
	return false;
}

/**
 * The setter method of the POINT class sets the POINT object's Y property to the function input
value
 * if that value is no smaller than MINIMUM_Y and which is no larger than MAXIMUM_Y.
 *
 * If the input value is in range, then return true.
 * Otherwise, do not change the caller POINT object's Y value and return false.
 *
 * (The keyword this refers to the POINT object which calls this function).
 *
```

```
 * (Y represents a specific whole number position along the y-axis (i.e. vertical dimension) of a
two-dimensional Cartesian grid).
 */
bool POINT::set_Y(int Y)
{
	if ((Y >= MINIMUM_Y) && (Y <= MAXIMUM_Y))
	{
	this -> Y = Y;
	return true;
	}
	return false;
}

/**
 * The getter method of the POINT class returns the length of the shortest path
 * between the two-dimensional point represented by the the caller POINT object (i.e. this)
 * and the two-dimensional point represented by the input POINT object (i.e. point).
 *
 * Use the Pythagorean Theorem to compute the length of a right triangle's hypotenuse
 * such that the two end points of that hypotenuse are represented by this and point.
 *
 * (A hypotenuse is the only side of a right triangle which does not form a right angle
 * with any other side of that triangle).
 *
 * (A hypotenuse is the longest side of a triangle (and a triangle is a three-sided polygon
 * in which three unique line segments connect three unique points)).
 *
 * // c is the length of a right triangle's hypotenuse.
 * // a is the length of that right triangle's horizontal leg.
 * // b is the length of that triangle's vertical leg.
 * (c * c) = (a * a) + (b * b).
 *
 * // sqrt() is a native C++ function defined in the cmath library.
 * c = square_root( (a * a) + (b * b)).
 */
double POINT::get_distance_from(POINT & point)
{
	int horizontal_difference = 0.0, vertical_difference = 0.0;
	horizontal_difference = X - point.X; // a
	vertical_difference = Y - point.Y; // b
	return sqrt((horizontal_difference * horizontal_difference) + (vertical_difference *
vertical_difference)); // c
}
```

```
/**
 * The getter method of the POINT class returns the slope of the line which intersects
 * the two-dimensional point represented by the caller POINT instance (i.e. this)
 * and the two-dimensional point represented by the input POINT instance (i.e. point).
 *
 * // y := f(x),
 * // b := f(0),
 * // f is a function whose input is an x-axis position and whose output is a y-axis position.
 * y := mx + b.
 *
 * // m is a constant which represents the rate at which y changes in relation to x changing.
 * m := (y - b) / x.
 *
 * // m represents the difference of the two y-values divided by the difference of the two x-values.
 * m := (point.Y - this.Y) / (point.X - this.X).
 */
double POINT::get_slope_of_line_to(POINT & point)
{
	double vertical_difference = 0.0, horizontal_difference = 0.0, result = 0.0;
	vertical_difference = point.Y - Y;
	horizontal_difference = point.X - X;
	result = vertical_difference / horizontal_difference;
	if (result == -0) result = 0; // Signed zeros sometimes occur inside of C++ program
runtime instances.
	return result;
}

/**
 * The print method of the POINT class prints a description of the caller POINT object to the
output stream.
 *
 * Note that the default value of the function input parameter is the standard command line
output stream (std::cout).
 *
 * The default parameter is defined in the POINT class header file (i.e. point.h) and not in the
POINT class source file (i.e. point.cpp).
 */
void POINT::print(std::ostream & output)
{
	output <<
"\n\n---------------------------------------------------------------------------------------------------";
	output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the
memory address of the first memory cell of a POINT sized chunk of contiguous memory cells
which are allocated to the caller POINT object.";
```

```
	output << "\n&X = " << &X << ". // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named X.";
	output << "\n&Y = " << &Y << ". // The reference operation returns the memory address of the first memory cell of an int sized chunk of contiguous memory cells which are allocated to the caller POINT data attribute named Y.";
	output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(POINT) = " << sizeof(POINT) << ". // The sizeof() operation returns the number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nX = " << X << ". // X stores one int type value at a time which represents the horizontal position of a two-dimensional point plotted on a Cartesian grid.";
	output << "\nY = " << Y << ". // Y stores one int type value at a time which represents the vertical position of a two-dimensional point plotted on a Cartesian grid.";
	output << "\n--------------------------------------------------------------------------------";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (<<).
 *
 * (Overloading an operator is assigning a different function to a native operator other than the function which that operator is used to represent by default).
 *
 * Note that the default value of the leftmost function input parameter is the standard command line output stream (std::cout).
 * The default parameter is defined in the POINT class header file (i.e. point.h).
 *
 * The friend function is not a member of the POINT class,
 * but the friend function has access to the private and protected members
 * of the POINT class and not just to the public members of the POINT class.
 *
 * The friend keyword only prefaces the function prototype of this function
 * (and the prototype of this function is declared in the POINT class header file (i.e. point.h)).
 *
 * The friend keyword does not preface the definition of this function
 * (and the definition of this function is specified in the POINT class source file (i.e. point.cpp)).
 *
 * // overloaded print function example one
 * POINT point_0;
 * std::cout << point_0; // identical to point_0.print();
 *
```

```
 * // overloaded print function example two
 * std::ofstream file;
 * POINT point_1;
 * file << point_1; // identical to point_1.print(file);
 */
std::ostream & operator << (std::ostream & output, POINT & point)
{
	point.print(output);
	return output;
}

/**
 * The destructor method of the POINT class de-allocates memory which was used to
 * instantiate the POINT object which is calling this function.
 *
 * The destructor method of the POINT class is automatically called when
 * the program scope in which the caller POINT object was instantiated terminates.
 */
POINT::~POINT()
{
	std::cout << "\n\nDeleting the POINT type object whose memory address is " << this <<
"...";
}
```

POLYGON_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the POLYGON class.

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/polygon.h

---

```
/**
 * file: polygon.h
 * type: C++ (header file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */
```

```
// If polygon.h has not already been linked to a source file (.cpp), then link this header file to the source file(s) which include this header file.
#ifndef POLYGON_H
#define POLYGON_H

// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the POINT class.
#include "point.h"

/**
 * POLYGON is an abstract class whose members are the essential components of objects
 * whose classes are TRILATERAL, RIGHT_TRILATERAL, QUADRILATERAL, TRAPEZOID, RECTANGLE, and SQUARE.
 *
 * (An abstract class has at least one virtual function).
 *
 * The POLYGON class includes the POINT class via composition and not via inheritance.
 *
 * Class members which are set to the protected access specifier
 * are accessible to the base class and to derived classes.
 *
 * Class members which are set to the private access specifier
 * are only accessible to the base class.
 *
 * Class members which are set to the public access specifier
 * are accessible to any scope within the program where
 * the base class and its derived classes are implemented.
 */
class POLYGON
{
protected:

    /**
    * category is a description of the POLYGON instance.
    * category is set to a constant (i.e. immutable) string type value.
    */
    const std::string category = "POLYGON";

    /**
    * color is an arbitrary string type value.
    * color is used to demonstrate how abstract constructors work.
    */
    std::string color;
```

```
public:

	/**
	* The default POLYGON constructor sets the color value to "orange".
	*
	* Note that POLYGON type objects cannot be instantiated (i.e. occupy space in memory)
	* because the POLYGON class is abstract.
	*
	* (pointer-to-POLYGON type variables can be instantiated, however, and used to store the memory addresses
	* of objects whose classes are derived from the POLYGON).
	*
	* POLYGON polygon; // This command does not work because POLYGON is an abstract class.
	* POLYGON * pointer_to_polygon; // The pointer-to-polygon type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as QUADRILATERAL.
	* pointer_to_polygon = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of during program compile time).
	* pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method and not the QUADRILATERAL print method.
	*
	* The POLYGON constructor is implemented only by classes which are
	* descendents of the POLYGON class.
	*/
	POLYGON();

	/**
	* The virtual methods get_area() and get_perimeter()
	* must be defined by classes which are derived from the POLYGON class.
	*/
	virtual double get_area() = 0;
	virtual double get_perimeter() = 0;

	/**
	* The descriptor method prints a description of the caller POLYGON instance to the output stream.
	* If no function input is supplied, output is set to the command line terminal.
	*/
	void print(std::ostream & output = std::cout);

	/**
```

```
	* The friend function is an alternative to the print method.
	* The friend function overloads the ostream operator (i.e. <<).
	*
	* The friend function is not a member of the POLYGON class,
	* but that friend function does have access to the private and protected members of the
POLYGON class
	* as though that friend function was a member of the POLYGON class.
	*/
	friend std::ostream & operator << (std::ostream & output, POLYGON & polygon);

	/**
	* The destructor method of the POLYGON class de-allocates memory which was used to
	* instantiate the POLYGON object which is calling this function.
	*
	* The destructor method of the POLYGON class is automatically called when
	* the program scope in which the caller POLYGON object was instantiated terminates.
	*/
	~POLYGON();
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

POLYGON_CLASS_SOURCE_CODE

The following source code defines the functions of the POLYGON class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/polygon.cpp

---

```
/**
 * file: polygon.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */
```

```
// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the POLYGON class.
#include "polygon.h"

/**
 * The default POLYGON constructor sets the color value to "orange".
 *
 * Note that POLYGON type objects cannot be instantiated (i.e. occupy space in memory)
 * because the POLYGON class is abstract.
 *
 * (pointer-to-POLYGON type variables can be instantiated, however, and used to store the memory addresses
 * of objects whose classes are derived from the POLYGON).
 *
 * POLYGON polygon; // This command does not work because POLYGON is an abstract class.
 * POLYGON * pointer_to_polygon; // The pointer-to-polygon type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as QUADRILATERAL.
 * pointer_to_polygon = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of during program compile time).
 * pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method and not the QUADRILATERAL print method.
 *
 * The POLYGON constructor is implemented only by classes which are
 * descendents of the POLYGON class.
 */
POLYGON::POLYGON()
{
	std::cout << "\n\nCreating the POLYGON type object whose memory address is " << this << "...";
	color = "orange";
}

/**
 * The virtual methods get_area() and get_perimeter() must be defined by
 * classes which are derived from the POLYGON class.
 */
double POLYGON::get_area() { return 0.0; }
double POLYGON::get_perimeter() { return 0.0; }

/**
 * The descriptor method prints a description of the caller POLYGON instance to the output stream.
```

```
 * If no function input is supplied, output is set to the command line terminal.
 */
void POLYGON::print(std::ostream & output)
{
	output << 
"\n\n----------------------------------------------------------------------------------------------------";
	output << "\nmemory_address = " << this << ".";
	output << "\ncategory = " << category << ".";
	output << "\ncolor = " << color << ".";
	output << "\n&category = " << &category << ".";
	output << "\n&color = " << &color << ".";
	output << "\n----------------------------------------------------------------------------------------------------";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (i.e. <<).
 *
 * The friend function is not a member of the POLYGON class,
 * but that friend function does have access to the private and protected members of the 
POLYGON class
 * as though that friend function was a member of the POLYGON class.
 */
std::ostream & operator << (std::ostream & output, POLYGON & polygon)
{
	polygon.print(output);
	return output;
}

/**
 * The destructor method of the POLYGON class de-allocates memory which was used to
 * instantiate the POLYGON object which is calling this function.
 *
 * The destructor method of the POLYGON class is automatically called when
 * the program scope in which the caller POLYGON object was instantiated terminates.
 */
POLYGON::~POLYGON()
{
	std::cout << "\n\nDeleting the POLYGON type object whose memory address is " << this 
<< "...";
}
```

QUADRILATERAL_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the QUADRILATERAL class.

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/quadrilateral.h

---

```
/**
 * file: quadrilateral.h
 * type: C++ (header file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

// If quadrilateral.h has not already been linked to a source file (.cpp), then link this header file to the source file(s) which include this header file.
#ifndef QUADRILATERAL_H
#define QUADRILATERAL_H

// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the POLYGON class.
#include "polygon.h"

/**
 * QUADRILATERAL is a class which inherits the protected and public data
 * attributes and methods of POLYGON (and POLYGON is an abstract class).
 *
 * A QUADRILATERAL object represents an instance in which four unique POINT instances exist
 * such that each one of those four POINT instances represents a unique coordinate pair within the tuple of four POINT instances
 * (such that each coordinate pair represents exactly one two-dimensional point, POINT(X,Y), on a Cartesian grid).
 *
 * Each QUADRILATERAL object represents a specific four-sided polygon whose area is a positive real number.
 *
 * Class members which are set to the protected access specifier
```

```
 * are accessible to the base class and to derived classes.
 *
 * Class members which are set to the private access specifier
 * are only accessible to the base class.
 *
 * Class members which are set to the public access specifier
 * are accessible to any scope within the program where
 * the base class and its derived classes are implemented.
 */
class QUADRILATERAL: public POLYGON
{
protected:

	/**
	* category is a description of the POLYGON instance.
	* category is set to a constant (i.e. immutable) string type value.
	*/
	const std::string category = "POLYGON/QUADRILATERAL";

	/**
	* POINT type objects A, B, C, and D represent points on a Cartesian plane.
	* Each POINT type object has two int type variables for representing a two-dimensional
whole number coordinate pair.
	* The X data attribute of a POINT object represents a whole number position on the
horizontal axis (i.e. x-axis) of a Cartesian plane.
	* The Y data attribute of a POINT object represents a whole number position on the
vertical axis (i.e. y-axis) of the same Cartesian plane.
	*/
	POINT A, B, C, D;

	/**
	* If each of the four whole number coordinate pairs represented by the POINT type input
values named _A, _B, _C, and _D are unique whole number coordinate pairs,
	* return true.
	* Otherwise, return false.
	*/
	bool points_represent_unique_coordinate_pairs(POINT _A, POINT _B, POINT _C,
POINT _D);

	/**
	* If sum of the interior angle measurements of the quadrilateral which the caller
QUADRILATERAL object represents add up to approximately 360 degrees,
	* return true.
	* Otherwise, return false.
```

```
	*/
	bool interior_angles_add_up_to_360_degrees();

public:

	/**
	* The test function helps to illustrate how pointers work.
	*/
	int quadrilateral_test(); // return 555

	/**
	* The default constructor of the QUADRILATERAL class calls the constructor of the POLYGON class and
	* sets the POINT type data member of the QUADRILATERAL object returned by this function named A to POINT(0,0),
	* sets the POINT type data member of the QUADRILATERAL object returned by this function named B to POINT(0,5),
	* sets the POINT type data member of the QUADRILATERAL object returned by this function named C to POINT(4,5), and
	* sets the POINT type data member of the QUADRILATERAL object returned by this function named D to POINT(4,0).
	*/
	QUADRILATERAL();

	/**
	* The normal constructor of QUADRILATERAL attempts to set
	* the string type data member of this to the input string type value named color and
	* the POINT type data member of this named A to the input POINT type value named A and
	* the POINT type data member of this named B to the input POINT type value named B and
	* the POINT type data member of this named C to the input POINT type value named C and
	* the POINT type data member of this named D to the input POINT type value named D.
	*
	* (The keyword this refers to the QUADRILATERAL object which is returned by this function).
	*
	* If A, B, C, and D represent unique points on a Cartesian plane and
	* if the interior angles of the quadrilateral which those points would represent add up to 360 degrees and
	* if the area of the quadrilateral which those points represents is larger than zero,
	* use the input POINT values as the POINT values for the QUADRILATERAL object which is returned by this function.
```

```
*/
QUADRILATERAL(std::string color, POINT A, POINT B, POINT C, POINT D);

/**
* The copy constructor method of the QUADRILATERAL class
* instantiates QUADRILATERAL type objects
* whose A value is set to the A value of the input QUADRILATERAL object,
* whose B value is set to the B value of the input QUADRILATERAL object,
* whose C value is set to the C value of the input QUADRILATERAL object, and
* whose D value is set to the D value of the input QUADRILATERAL object.
*/
QUADRILATERAL(QUADRILATERAL & quadrilateral);

/**
* The QUADRILATERAL class implements the virtual get_area() method of the
POLYGON class.
*
* The getter method returns the area of the quadrilateral represented by the caller
QUADRILATERAL object
* using using Heron's Formula to
* compute the area of each of the two triangles which comprise that quadrilateral.
*
* Let AB be the length of the line segment whose endpoints are A and B.
* Let BC be the length of the line segment whose endpoints are B and C.
* Let CA be the length of the line segment whose endpoints are C and A.
* Let CD be the length of the line segment whose endpoints are C and D.
* Let DA be the length of the line segment whose endpoints are D and A.
*
* Let the first triangle be the area which is enclosed inside line segments represented by
AB, BC, and CA.
* Let the second triangle be the area which is enclosed inside line segments represented
by AC, CD, and DA.
*
* Then compute the area of each triangle using Heron's Formula as follows:
*
* Let s be the semiperimeter of a triangle (i.e. the perimeter divided by 2).
* Let a, b, and c be the side lengths of a triangle.
*
* Then
*
* area = square_root( s * (s - a) * (s - b) * (s - c) ).
*
* Finally, return the sum of the two triangle areas.
*/
```

```
double get_area();

/**
* The QUADRILATERAL class implements the virtual get_perimeter() method of the
POLYGON class.
*
* The getter method returns the perimeter of the quadrilateral represented by the caller
QUADRILATERAL object
* by adding up the four side lengths of that quadrilateral.
*
* Let AB be the length of the line segment whose endpoints are A and B.
* Let BC be the length of the line segment whose endpoints are B and C.
* Let CD be the length of the line segment whose endpoints are C and D.
* Let DA be the length of the line segment whose endpoints are D and A.
*
* Then return the sum of AB, BC, CD, and DA.
*/
double get_perimeter();

/**
* This method overrides the POLYGON class's print method.
*
* The descriptor method prints a description of the caller QUADRILATERAL instance to
the output stream.
*
* If no function input is supplied, output is set to the command line terminal.
*/
void print(std::ostream & output = std::cout);

/**
* The friend function is an alternative to the print method.
* The friend function overloads the ostream operator (i.e. <<).
*
* The friend function is not a member of the QUADRILATERAL class,
* but the friend function does have access to the private and protected members of the
QUADRILATERAL class as though
* the friend function was a member of the QUADRILATERAL class.
*/
friend std::ostream & operator << (std::ostream & output, QUADRILATERAL &
quadrilateral);

/**
* The destructor method of the QUADRILATERAL class de-allocates memory which was
used to
```

* instantiate the QUADRILATERAL object which is calling this function.
*
* The destructor method of the QUADRILATERAL class is automatically called when
* the program scope in which the caller QUADRILATERAL object was instantiated terminates.
*/
~QUADRILATERAL();
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.

QUADRILATERAL_CLASS_SOURCE_CODE

The following source code defines the functions of the QUADRILATERAL class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/quadrilateral.cpp

---

/**
 * file: quadrilateral.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the QUADRILATERAL class.
#include "quadrilateral.h"

/**
 * If each of the four whole number coordinate pairs represented by the POINT type input values named _A, _B, _C, and _D are unique whole number coordinate pairs,
 * return true.
 * Otherwise, return false.
 */
bool QUADRILATERAL::points_represent_unique_coordinate_pairs(POINT _A, POINT _B, POINT _C, POINT _D)

```
{
	if ((_A.get_X() == _B.get_X()) && (_A.get_Y() == _B.get_Y())) return false;
	if ((_A.get_X() == _C.get_X()) && (_A.get_Y() == _C.get_Y())) return false;
	if ((_A.get_X() == _D.get_X()) && (_A.get_Y() == _D.get_Y())) return false;
	if ((_B.get_X() == _C.get_X()) && (_B.get_Y() == _C.get_Y())) return false;
	if ((_B.get_X() == _D.get_X()) && (_B.get_Y() == _D.get_Y())) return false;
	if ((_C.get_X() == _D.get_X()) && (_C.get_Y() == _D.get_Y())) return false;
	return true;
}

/**
 * If sum of the interior angle measurements of the quadrilateral which the caller
QUADRILATERAL object represents add up to approximately 360 degrees,
 * return true.
 * Otherwise, return false.
 */
bool QUADRILATERAL::interior_angles_add_up_to_360_degrees()
{
	double a0 = 0.0, b0 = 0.0, c0 = 0.0;
	double a1 = 0.0, b1 = 0.0, c1 = 0.0;
	double angle_opposite_of_a0 = 0.0, angle_opposite_of_b0 = 0.0, angle_opposite_of_c0
= 0.0;
	double angle_opposite_of_a1 = 0.0, angle_opposite_of_b1 = 0.0, angle_opposite_of_c1
= 0.0;
	double interior_angle_of_A = 0.0, interior_angle_of_B = 0.0, interior_angle_of_C = 0.0,
interior_angle_of_D = 0.0;
	double sum_of_interior_angles = 0.0;

	// first triangle
	a0 = A.get_distance_from(B);
	b0 = B.get_distance_from(D);
	c0 = D.get_distance_from(A);
	angle_opposite_of_a0 = acos(((b0 * b0) + (c0 * c0) - (a0 * a0)) / (2 * b0 * c0)) * (180 / PI);
	angle_opposite_of_b0 = acos(((a0 * a0) + (c0 * c0) - (b0 * b0)) / (2 * a0 * c0)) * (180 / PI);
	angle_opposite_of_c0 = acos(((a0 * a0) + (b0 * b0) - (c0 * c0)) / (2 * a0 * b0)) * (180 / PI);

	// second triangle
	a1 = D.get_distance_from(B);
	b1 = B.get_distance_from(C);
	c1 = C.get_distance_from(D);
	angle_opposite_of_a1 = acos(((b1 * b1) + (c1 * c1) - (a1 * a1)) / (2 * b1 * c1)) * (180 / PI);
	angle_opposite_of_b1 = acos(((a1 * a1) + (c1 * c1) - (b1 * b1)) / (2 * a1 * c1)) * (180 / PI);
	angle_opposite_of_c1 = acos(((a1 * a1) + (b1 * b1) - (c1 * c1)) / (2 * a1 * b1)) * (180 / PI);
```

```
	interior_angle_of_A = angle_opposite_of_b0;
	interior_angle_of_B = angle_opposite_of_c0 + angle_opposite_of_c1;
	interior_angle_of_C = angle_opposite_of_a1;
	interior_angle_of_D = angle_opposite_of_b1 + angle_opposite_of_a0;

	sum_of_interior_angles = interior_angle_of_A + interior_angle_of_B +
interior_angle_of_C + interior_angle_of_D;

	// Allow for there to be a +/- 2 margin of error for the value stored in
sum_of_interior_angles with the ideal value being 360.
	if ((sum_of_interior_angles >= 358) && (sum_of_interior_angles <= 362)) return true;
	return false;
}

/**
 * The test function helps to illustrate how pointers work.
 */
int QUADRILATERAL::quadrilateral_test()
{
	return 555;
}

/**
 * The default constructor of the QUADRILATERAL class calls the constructor of the POLYGON
class and
 * sets the POINT type data member of the QUADRILATERAL object returned by this function
named A to POINT(0,0),
 * sets the POINT type data member of the QUADRILATERAL object returned by this function
named B to POINT(0,5),
 * sets the POINT type data member of the QUADRILATERAL object returned by this function
named C to POINT(4,5), and
 * sets the POINT type data member of the QUADRILATERAL object returned by this function
named D to POINT(4,0).
 */
QUADRILATERAL::QUADRILATERAL()
{
	std::cout << "\n\nCreating the QUADRILATERAL type object whose memory address is "
<< this << "...";
	A = POINT(0,0);
	B = POINT(0,5);
	C = POINT(4,5);
	D = POINT(4,0);
}
```

```
/**
 * The normal constructor of QUADRILATERAL attempts to set
 * the string type data member of this to the input string type value named color and
 * the POINT type data member of this named A to the input POINT type value named A and
 * the POINT type data member of this named B to the input POINT type value named B and
 * the POINT type data member of this named C to the input POINT type value named C and
 * the POINT type data member of this named D to the input POINT type value named D.
 *
 * (The keyword this refers to the QUADRILATERAL object which is returned by this function).
 *
 * If A, B, C, and D represent unique points on a Cartesian plane and
 * if the interior angles of the quadrilateral which those points would represent add up to 360
degrees and
 * if the area of the quadrilateral which those points represents is larger than zero,
 * use the input POINT values as the POINT values for the QUADRILATERAL object which is
returned by this function.
 */
QUADRILATERAL::QUADRILATERAL(std::string color, POINT A, POINT B, POINT C, POINT
D)
{
	std::cout << "\n\nCreating the QUADRILATERAL type object whose memory address is "
<< this << "...";
	QUADRILATERAL test_quadrilateral;
	test_quadrilateral.A.set_X(A.get_X());
	test_quadrilateral.A.set_Y(A.get_Y());
	test_quadrilateral.B.set_X(B.get_X());
	test_quadrilateral.B.set_Y(B.get_Y());
	test_quadrilateral.C.set_X(C.get_X());
	test_quadrilateral.C.set_Y(C.get_Y());
	test_quadrilateral.D.set_X(D.get_X());
	test_quadrilateral.D.set_Y(D.get_Y());
	if (points_represent_unique_coordinate_pairs(A, B, C, D) &&
test_quadrilateral.interior_angles_add_up_to_360_degrees() && (test_quadrilateral.get_area() >
0))
	{
	this -> A = A;
	this -> B = B;
	this -> C = C;
	this -> D = D;
	}
	else
	{
	this -> A = POINT(0,0);
	this -> B = POINT(0,5);
```

```
        this -> C = POINT(4,5);
        this -> D = POINT(4,0);
        }
        this -> color = color;
}

/**
 * The copy constructor method of the QUADRILATERAL class
 * instantiates QUADRILATERAL type objects
 * whose A value is set to the A value of the input QUADRILATERAL object,
 * whose B value is set to the B value of the input QUADRILATERAL object,
 * whose C value is set to the C value of the input QUADRILATERAL object, and
 * whose D value is set to the D value of the input QUADRILATERAL object.
 */
QUADRILATERAL::QUADRILATERAL(QUADRILATERAL & quadrilateral)
{
        std::cout << "\n\nCreating the QUADRILATERAL type object whose memory address is "
<< this << "...";
        A = quadrilateral.A;
        B = quadrilateral.B;
        C = quadrilateral.C;
        D = quadrilateral.D;
        color = quadrilateral.color;
}

/**
 * The QUADRILATERAL class implements the virtual get_area() method of the POLYGON
class.
 *
 * The getter method returns the area of the quadrilateral represented by the caller
QUADRILATERAL object
 * using using Heron's Formula to
 * compute the area of each of the two triangles which comprise that quadrilateral.
 *
 * Let AB be the length of the line segment whose endpoints are A and B.
 * Let BC be the length of the line segment whose endpoints are B and C.
 * Let CA be the length of the line segment whose endpoints are C and A.
 * Let CD be the length of the line segment whose endpoints are C and D.
 * Let DA be the length of the line segment whose endpoints are D and A.
 *
 * Let the first triangle be the area which is enclosed inside line segments represented by AB,
BC, and CA.
 * Let the second triangle be the area which is enclosed inside line segments represented by
AC, CD, and DA.
```

```
 *
 * Then compute the area of each triangle using Heron's Formula as follows:
 *
 * Let s be the semiperimeter of a triangle (i.e. the perimeter divided by 2).
 * Let a, b, and c be the side lengths of a triangle.
 *
 * Then
 *
 * area = square_root( s * (s - a) * (s - b) * (s - c) ).
 *
 * Finally, return the sum of the two triangle areas.
 */
double QUADRILATERAL::get_area()
{
	double a0 = 0.0, b0 = 0.0, c0= 0.0, s0 = 0.0, area_0 = 0.0;
	double a1 = 0.0, b1 = 0.0, c1 = 0.0, s1 = 0.0, area_1 = 0.0;

	// first triangle
	a0 = A.get_distance_from(B);
	b0 = B.get_distance_from(C);
	c0 = C.get_distance_from(A);
	s0 = (a0 + b0 + c0) / 2;
	area_0 = sqrt(s0 * (s0 - a0) * (s0 - b0) * (s0 - c0));

	// second triangle
	a1 = A.get_distance_from(C);
	b1 = C.get_distance_from(D);
	c1 = D.get_distance_from(A);
	s1 = (a1 + b1 + c1) / 2;
	area_1 = sqrt(s1 * (s1 - a1) * (s1 - b1) * (s1 - c1));

	// Return the sum of the two triangle areas.
	return area_0 + area_1;
}

/**
 * The QUADRILATERAL class implements the virtual get_perimeter() method of the POLYGON
class.
 *
 * The getter method returns the perimeter of the quadrilateral represented by the caller
QUADRILATERAL object
 * by adding up the four side lengths of that quadrilateral.
 *
 * Let AB be the length of the line segment whose endpoints are A and B.
```

```
 * Let BC be the length of the line segment whose endpoints are B and C.
 * Let CD be the length of the line segment whose endpoints are C and D.
 * Let DA be the length of the line segment whose endpoints are D and A.
 *
 * Then return the sum of AB, BC, CD, and DA.
 */
double QUADRILATERAL::get_perimeter()
{
	double AB = 0.0, BC = 0.0, CD = 0.0, DA = 0.0;
	AB = A.get_distance_from(B);
	BC = B.get_distance_from(C);
	CD = C.get_distance_from(D);
	DA = D.get_distance_from(A);
	return AB + BC + CD + DA;
}

/**
 * This method overrides the POLYGON class's print method.
 *
 * The descriptor method prints a description of the caller QUADRILATERAL instance to the
output stream.
 *
 * If no function input is supplied, output is set to the command line terminal.
 */
void QUADRILATERAL::print(std::ostream & output)
{
	double a0 = 0.0, b0 = 0.0, c0 = 0.0;
	double a1 = 0.0, b1 = 0.0, c1 = 0.0;
	double angle_opposite_of_a0 = 0.0, angle_opposite_of_b0 = 0.0, angle_opposite_of_c0
= 0.0;
	double angle_opposite_of_a1 = 0.0, angle_opposite_of_b1 = 0.0, angle_opposite_of_c1
= 0.0;
	double interior_angle_of_A = 0.0, interior_angle_of_B = 0.0, interior_angle_of_C = 0.0,
interior_angle_of_D = 0.0;

	// first triangle
	a0 = A.get_distance_from(B);
	b0 = B.get_distance_from(D);
	c0 = D.get_distance_from(A);
	angle_opposite_of_a0 = acos(((b0 * b0) + (c0 * c0) - (a0 * a0)) / (2 * b0 * c0)) * (180 / PI);
	angle_opposite_of_b0 = acos(((a0 * a0) + (c0 * c0) - (b0 * b0)) / (2 * a0 * c0)) * (180 / PI);
	angle_opposite_of_c0 = acos(((a0 * a0) + (b0 * b0) - (c0 * c0)) / (2 * a0 * b0)) * (180 / PI);

	// second triangle
```

```
	a1 = D.get_distance_from(B);
	b1 = B.get_distance_from(C);
	c1 = C.get_distance_from(D);
	angle_opposite_of_a1 = acos(((b1 * b1) + (c1 * c1) - (a1 * a1)) / (2 * b1 * c1)) * (180 / PI);
	angle_opposite_of_b1 = acos(((a1 * a1) + (c1 * c1) - (b1 * b1)) / (2 * a1 * c1)) * (180 / PI);
	angle_opposite_of_c1 = acos(((a1 * a1) + (b1 * b1) - (c1 * c1)) / (2 * a1 * b1)) * (180 / PI);

	interior_angle_of_A = angle_opposite_of_b0;
	interior_angle_of_B = angle_opposite_of_c0 + angle_opposite_of_c1;
	interior_angle_of_C = angle_opposite_of_a1;
	interior_angle_of_D = angle_opposite_of_b1 + angle_opposite_of_a0;

	output << "\n\n----------------------------------------------------------------------------------------------------";
	output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.";
	output << "\n&category = " << &category << ". // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.";
	output << "\n&color = " << &color << ". // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..";
	output << "\n&A = " << &A << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.";
	output << "\n&B = " << &B << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.";
	output << "\n&C = " << &C << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.";
	output << "\n&D = " << &D << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.";
	output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(int *) = " << sizeof(int *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(int **) = " << sizeof(int **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
```

output << "\nsizeof(std::string) = " << sizeof(std::string) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string *) = " << sizeof(std::string *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string **) = " << sizeof(std::string **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT) = " << sizeof(POINT) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT *) = " << sizeof(POINT *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT **) = " << sizeof(POINT **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON) = " << sizeof(POLYGON) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON *) = " << sizeof(POLYGON *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON **) = " << sizeof(POLYGON **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL) = " << sizeof(QUADRILATERAL) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL *) = " << sizeof(QUADRILATERAL *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL **) = " << sizeof(QUADRILATERAL **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\ncategory = " << category << ". // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.";

output << "\ncolor = " << color << ". // This is a string type value which is a data member of the caller QUADRILATERAL object.";

output << "\nA = POINT(" << A.get_X() << "," << A.get_Y() << "). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nB = POINT(" << B.get_X() << "," << B.get_Y() << "). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nC = POINT(" << C.get_X() << "," << C.get_Y() << "). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nD = POINT(" << D.get_X() << "," << D.get_Y() << "). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\na = B.get_distance_from(C) = " << B.get_distance_from(C) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.";

output << "\nb = C.get_distance_from(D) = " << C.get_distance_from(D) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.";

output << "\nc = D.get_distance_from(A) = " << D.get_distance_from(A) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.";

output << "\nd = A.get_distance_from(B) = " << A.get_distance_from(B) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.";

output << "\nA.get_slope_of_line_to(B) = " << A.get_slope_of_line_to(B) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.";

output << "\nB.get_slope_of_line_to(C) = " << B.get_slope_of_line_to(C) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\nC.get_slope_of_line_to(D) = " << C.get_slope_of_line_to(D) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.";

output << "\nD.get_slope_of_line_to(A) = " << D.get_slope_of_line_to(A) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\ninterior_angle_DAB = interior_angle_of_A = " << interior_angle_of_A << ". // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_ABC = interior_angle_of_B = " << interior_angle_of_B << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_BCD = interior_angle_of_C = " << interior_angle_of_C << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_CDA = interior_angle_of_D = " << interior_angle_of_D << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = " << interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D << ". // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)";

output << "\nget_perimeter() = a + b + c + d = " << get_perimeter() << ". // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.";

output << "\nget_area() = " << get_area() << ". // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.";

output << "\n--";

}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (i.e. <<).
 *
 * The friend function is not a member of the QUADRILATERAL class,
 * but it does have access to the members of QUADRILATERAL as though

```
 * it were a member of that class.
 */
std::ostream & operator << (std::ostream & output, QUADRILATERAL & quadrilateral)
{
	quadrilateral.print(output);
	return output;
}

/**
 * The destructor method of the QUADRILATERAL class de-allocates memory which was used
to
 * instantiate the QUADRILATERAL object which is calling this function.
 *
 * The destructor method of the QUADRILATERAL class is automatically called when
 * the program scope in which the caller QUADRILATERAL object was instantiated terminates.
 */
QUADRILATERAL::~QUADRILATERAL()
{
	std::cout << "\n\nDeleting the QUADRILATERAL type object whose memory address is "
<< this << "...";
}
```

TRAPEZOID_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the TRAPEZOID class.

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/trapezoid.h

---

```
/**
 * file: trapezoid.h
 * type: C++ (header file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */
```

```
// If trapezoid.h has not already been linked to a source file (.cpp), then link this header file to
the source file(s) which include this header file.
#ifndef TRAPEZOID_H
#define TRAPEZOID_H

// Include the C++ header file which contains preprocessing directives, variable declarations,
and function prototypes for the QUADRILATERAL class.
#include "quadrilateral.h"

/**
 * TRAPEZOID is a class which inherits the protected and public data
 * attributes and methods of QUADRILATERAL.
 *
 * A TRAPEZOID object represents a four-sided polygon such that exactly two opposite sides
 * are parallel to each other and not the same length.
 *
 * Class members which are set to the protected access specifier
 * are accessible to the base class and to derived classes.
 *
 * Class members which are set to the private access specifier
 * are only accessible to the base class.
 *
 * Class members which are set to the public access specifier
 * are accessible to any scope within the program where
 * the base class and its derived classes are implemented.
 */
class TRAPEZOID: public QUADRILATERAL
{
protected:

	/**
	* category is a description of the POLYGON instance.
	* category is set to a const (i.e. const (i.e. immutable)) value.
	*/
	const std::string category = "POLYGON/QUADRILATERAL/TRAPEZOID";

	/**
	* The helper method determines whether or not the caller TRAPEZOID instance
represents a
	* quadrilateral with exactly two parallel opposite sides whose lengths are not identical.
	* Return true if the caller TRAPEZOID satisfies those conditions. Otherwise, return false.
	*/
	bool is_trapezoid();
```

public:

/**
* The default constructor of the TRAPEZOID class calls the constructor of the QUADRILATERAL class and
* sets the POINT type data member of the TRAPEZOID object returned by this function named A to POINT(0,0),
* sets the POINT type data member of the TRAPEZOID object returned by this function named B to POINT(1,1),
* sets the POINT type data member of the TRAPEZOID object returned by this function named C to POINT(2,1), and
* sets the POINT type data member of the TRAPEZOID object returned by this function named D to POINT(3,0).
*/
TRAPEZOID();

/**
* The normal constructor of TRAPEZOID attempts to set
* the string type data member of this to the input string type value named color and
* the POINT type data member of this named A to the input POINT type value named A and
* the POINT type data member of this named B to the input POINT type value named B and
* the POINT type data member of this named C to the input POINT type value named C and
* the POINT type data member of this named D to the input POINT type value named D.
*
* (The keyword this refers to the TRAPEZOID object which is returned by this function).
*
* If A, B, C, and D represent unique points on a Cartesian plane and
* if the interior angles of the quadrilateral which those points would represent add up to 360 degrees and
* if the area of the quadrilateral which those points represents is larger than zero, and
* if exactly two sides of the quadrilateral are parallel to each other,
* use the input POINT values as the POINT values for the TRAPEZOID object which is returned by this function.
*/
TRAPEZOID(std::string color, POINT A, POINT B, POINT C, POINT D);

/**
* The copy constructor of TRAPEZOID creates a clone of
* the input TRAPEZOID instance.
*/
TRAPEZOID(TRAPEZOID & trapezoid);

```
	/**
	* This method overrides the QUADRILATERAL class's print method.
	*
	* The descriptor method prints a description of the caller TRAPEZOID instance to the
output stream.
	*
	* If no function input is supplied, output is set to the command line terminal.
	*/
	void print(std::ostream & output = std::cout);

	/**
	* The friend function is an alternative to the print method.
	* The friend function overloads the ostream operator (i.e. <<).
	*
	* The friend function is not a member of the TRAPEZOID class,
	* but the friend function does have access to the private and protected members of the
TRAPEZOID class as though
	* the friend function was a member of the TRAPEZOID class.
	*/
	friend std::ostream & operator << (std::ostream & output, TRAPEZOID & trapezoid);

	/**
	* The destructor method of the TRAPEZOID class de-allocates memory which was used
to
	* instantiate the TRAPEZOID object which is calling this function.
	*
	* The destructor method of the TRAPEZOID class is automatically called when
	* the program scope in which the caller TRAPEZOID object was instantiated terminates.
	*/
	~TRAPEZOID();
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

TRAPEZOID_CLASS_SOURCE_CODE

The following source code defines the functions of the TRAPEZOID class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/trapezoid.cpp

---

```
/**
 * file: trapezoid.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

// Include the C++ header file which contains preprocessing directives, variable declarations,
and function prototypes for the TRAPEZOID class.
#include "trapezoid.h"

/**
 * The helper method determines whether or not the caller TRAPEZOID instance represents a
 * quadrilateral with exactly two parallel opposite sides whose lengths are not identical.
 * Return true if the caller TRAPEZOID satisfies those conditions. Otherwise, return false.
 */
bool TRAPEZOID::is_trapezoid()
{
	double a = 0.0, b = 0.0, c = 0.0, d = 0.0;
	double slope_of_a = 0.0, slope_of_b = 0.0, slope_of_c = 0.0, slope_of_d = 0.0;
	double a0 = 0.0, b0 = 0.0, c0 = 0.0;
	double a1 = 0.0, b1 = 0.0, c1 = 0.0;
	double angle_opposite_of_a0 = 0.0, angle_opposite_of_b0 = 0.0, angle_opposite_of_c0
= 0.0;
	double angle_opposite_of_a1 = 0.0, angle_opposite_of_b1 = 0.0, angle_opposite_of_c1
= 0.0;
	double interior_angle_of_A = 0.0, interior_angle_of_B = 0.0, interior_angle_of_C = 0.0,
interior_angle_of_D = 0.0;
	double sum_of_interior_angles = 0.0;

	// first triangle
	a0 = floor(A.get_distance_from(B));
	b0 = floor(B.get_distance_from(D));
	c0 = floor(D.get_distance_from(A));
	angle_opposite_of_a0 = floor(acos(((b0 * b0) + (c0 * c0) - (a0 * a0)) / (2 * b0 * c0)) * (180
/ PI));
```

```
	angle_opposite_of_b0 = floor(acos(((a0 * a0) + (c0 * c0) - (b0 * b0)) / (2 * a0 * c0)) * (180
/ PI));
	angle_opposite_of_c0 = floor(acos(((a0 * a0) + (b0 * b0) - (c0 * c0)) / (2 * a0 * b0)) * (180
/ PI));

	// second triangle
	a1 = floor(D.get_distance_from(B));
	b1 = floor(B.get_distance_from(C));
	c1 = floor(C.get_distance_from(D));
	angle_opposite_of_a1 = floor(acos(((b1 * b1) + (c1 * c1) - (a1 * a1)) / (2 * b1 * c1)) * (180
/ PI));
	angle_opposite_of_b1 = floor(acos(((a1 * a1) + (c1 * c1) - (b1 * b1)) / (2 * a1 * c1)) * (180
/ PI));
	angle_opposite_of_c1 = floor(acos(((a1 * a1) + (b1 * b1) - (c1 * c1)) / (2 * a1 * b1)) * (180
/ PI));

	interior_angle_of_A = angle_opposite_of_b0;
	interior_angle_of_B = angle_opposite_of_c0 + angle_opposite_of_c1;
	interior_angle_of_C = angle_opposite_of_a1;
	interior_angle_of_D = angle_opposite_of_b1 + angle_opposite_of_a0;

	/* sides of quadrilateral */
	a = B.get_distance_from(C);
	b = C.get_distance_from(D);
	c = D.get_distance_from(A);
	d = A.get_distance_from(B);

	/* slope of sides of quadrilateral */
	slope_of_a = B.get_slope_of_line_to(C);
	slope_of_b = C.get_slope_of_line_to(D);
	slope_of_c = D.get_slope_of_line_to(A);
	slope_of_d = A.get_slope_of_line_to(B);

	if (!points_represent_unique_coordinate_pairs(A,B,C,D) ||
!interior_angles_add_up_to_360_degrees()) return false;
	if ((slope_of_a == slope_of_c) && (slope_of_b != slope_of_d)) return true;
	if ((slope_of_a != slope_of_c) && (slope_of_b == slope_of_d)) return true;
	return false;
}

/**
 * The default constructor of the TRAPEZOID class calls the constructor of the
QUADRILATERAL class and
```

```
 * sets the POINT type data member of the TRAPEZOID object returned by this function named A to POINT(0,0),
 * sets the POINT type data member of the TRAPEZOID object returned by this function named B to POINT(1,1),
 * sets the POINT type data member of the TRAPEZOID object returned by this function named C to POINT(2,1), and
 * sets the POINT type data member of the TRAPEZOID object returned by this function named D to POINT(3,0).
 */
TRAPEZOID::TRAPEZOID()
{
	std::cout << "\n\nCreating the TRAPEZOID type object whose memory address is " << this << "...";
	A = POINT(0,0);
	B = POINT(1,1);
	C = POINT(2,1);
	D = POINT(3,0);
}

/**
 * The normal constructor of TRAPEZOID attempts to set
 * the string type data member of this to the input string type value named color and
 * the POINT type data member of this named A to the input POINT type value named A and
 * the POINT type data member of this named B to the input POINT type value named B and
 * the POINT type data member of this named C to the input POINT type value named C and
 * the POINT type data member of this named D to the input POINT type value named D.
 *
 * (The keyword this refers to the TRAPEZID object which is returned by this function).
 *
 * If A, B, C, and D represent unique points on a Cartesian plane and
 * if the interior angles of the quadrilateral which those points would represent add up to 360 degrees and
 * if the area of the quadrilateral which those points represents is larger than zero, and
 * if exactly two sides of the quadrilateral are parallel to each other,
 * use the input POINT values as the POINT values for the TRAPEZOID object which is returned by this function.
 */
TRAPEZOID::TRAPEZOID(std::string color, POINT A, POINT B, POINT C, POINT D)
{
	std::cout << "\n\nCreating the TRAPEZOID type object whose memory address is " << this << "...";
	TRAPEZOID test_trapezoid;
	test_trapezoid.A.set_X(A.get_X());
	test_trapezoid.A.set_Y(A.get_Y());
```

```
        test_trapezoid.B.set_X(B.get_X());
        test_trapezoid.B.set_Y(B.get_Y());
        test_trapezoid.C.set_X(C.get_X());
        test_trapezoid.C.set_Y(C.get_Y());
        test_trapezoid.D.set_X(D.get_X());
        test_trapezoid.D.set_Y(D.get_Y());
        if (test_trapezoid.is_trapezoid())
        {
        this -> A = A;
        this -> B = B;
        this -> C = C;
        this -> D = D;
        }
        else
        {
        this -> A = POINT(0,0);
        this -> B = POINT(0,5);
        this -> C = POINT(4,5);
        this -> D = POINT(4,0);
        }
        this -> color = color;
}

/**
 * The copy constructor of TRAPEZOID creates a clone of
 * the input TRAPEZOID instance.
 */
TRAPEZOID::TRAPEZOID(TRAPEZOID & trapezoid)
{
        std::cout << "\n\nCreating the TRAPEZOID type object whose memory address is " <<
this << "...";
        A = trapezoid.A;
        B = trapezoid.B;
        C = trapezoid.C;
        D = trapezoid.D;
        color = trapezoid.color;
}

/**
 * This method overrides the QUADRILATERAL class's print method.
 *
 * The descriptor method prints a description of the caller TRAPEZOID instance to the output
stream.
 *
```

```
 * If no function input is supplied, output is set to the command line terminal.
 */
void TRAPEZOID::print(std::ostream & output)
{
	double a0 = 0.0, b0 = 0.0, c0 = 0.0;
	double a1 = 0.0, b1 = 0.0, c1 = 0.0;
	double angle_opposite_of_a0 = 0.0, angle_opposite_of_b0 = 0.0, angle_opposite_of_c0 
= 0.0;
	double angle_opposite_of_a1 = 0.0, angle_opposite_of_b1 = 0.0, angle_opposite_of_c1 
= 0.0;
	double interior_angle_of_A = 0.0, interior_angle_of_B = 0.0, interior_angle_of_C = 0.0, 
interior_angle_of_D = 0.0;

	// first triangle
	a0 = A.get_distance_from(B);
	b0 = B.get_distance_from(D);
	c0 = D.get_distance_from(A);
	angle_opposite_of_a0 = acos(((b0 * b0) + (c0 * c0) - (a0 * a0)) / (2 * b0 * c0)) * (180 / PI);
	angle_opposite_of_b0 = acos(((a0 * a0) + (c0 * c0) - (b0 * b0)) / (2 * a0 * c0)) * (180 / PI);
	angle_opposite_of_c0 = acos(((a0 * a0) + (b0 * b0) - (c0 * c0)) / (2 * a0 * b0)) * (180 / PI);

	// second triangle
	a1 = D.get_distance_from(B);
	b1 = B.get_distance_from(C);
	c1 = C.get_distance_from(D);
	angle_opposite_of_a1 = acos(((b1 * b1) + (c1 * c1) - (a1 * a1)) / (2 * b1 * c1)) * (180 / PI);
	angle_opposite_of_b1 = acos(((a1 * a1) + (c1 * c1) - (b1 * b1)) / (2 * a1 * c1)) * (180 / PI);
	angle_opposite_of_c1 = acos(((a1 * a1) + (b1 * b1) - (c1 * c1)) / (2 * a1 * b1)) * (180 / PI);

	interior_angle_of_A = angle_opposite_of_b0;
	interior_angle_of_B = angle_opposite_of_c0 + angle_opposite_of_c1;
	interior_angle_of_C = angle_opposite_of_a1;
	interior_angle_of_D = angle_opposite_of_b1 + angle_opposite_of_a0;

	output << 
"\n\n--------------------------------------------------------------------------------------------------";
	output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the 
memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous 
memory cells which are allocated to the caller TRIANGLE object.";
	output << "\n&category = " << &category << ". // The reference operation returns the 
memory address of the first memory cell of a string sized chunk of contiguous memory cells 
which are allocated to the string data attribute named category.";
```

output << "\n&color = " << &color << ". // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..";

output << "\n&A = " << &A << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.";

output << "\n&B = " << &B << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.";

output << "\n&C = " << &C << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.";

output << "\n&D = " << &D << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.";

output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(int *) = " << sizeof(int *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(int **) = " << sizeof(int **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string) = " << sizeof(std::string) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string *) = " << sizeof(std::string *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string **) = " << sizeof(std::string **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT) = " << sizeof(POINT) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT *) = " << sizeof(POINT *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT **) = " << sizeof(POINT **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON) = " << sizeof(POLYGON) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON *) = " << sizeof(POLYGON *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON **) = " << sizeof(POLYGON **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL) = " << sizeof(QUADRILATERAL) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL *) = " << sizeof(QUADRILATERAL *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL **) = " << sizeof(QUADRILATERAL **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(TRAPEZOID) = " << sizeof(TRAPEZOID) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRAPEZOID type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(TRAPEZOID *) = " << sizeof(TRAPEZOID *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRAPEZOID type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(TRAPEZOID **) = " << sizeof(TRAPEZOID **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRAPEZOID type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\ncategory = " << category << ". // This is an immutable string type value which is a data member of the caller TRAPEZOID object.";

output << "\ncolor = " << color << ". // This is a string type value which is a data member of the caller TRAPEZOID object.";

output << "\nA = POINT(" << A.get_X() << "," << A.get_Y() << "). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nB = POINT(" << B.get_X() << "," << B.get_Y() << "). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nC = POINT(" << C.get_X() << "," << C.get_Y() << "). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nD = POINT(" << D.get_X() << "," << D.get_Y() << "). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\na = B.get_distance_from(C) = " << B.get_distance_from(C) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.";

output << "\nb = C.get_distance_from(D) = " << C.get_distance_from(D) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.";

output << "\nc = D.get_distance_from(A) = " << D.get_distance_from(A) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.";

output << "\nd = A.get_distance_from(B) = " << A.get_distance_from(B) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.";

output << "\nA.get_slope_of_line_to(B) = " << A.get_slope_of_line_to(B) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.";

output << "\nB.get_slope_of_line_to(C) = " << B.get_slope_of_line_to(C) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\nC.get_slope_of_line_to(D) = " << C.get_slope_of_line_to(D) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.";

output << "\nD.get_slope_of_line_to(A) = " << D.get_slope_of_line_to(A) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\ninterior_angle_DAB = interior_angle_of_A = " << interior_angle_of_A << ". // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_ABC = interior_angle_of_B = " << interior_angle_of_B << ". // The method returns the approximate nonnegative real number angle measurement of the acute

or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).";
	output << "\ninterior_angle_BCD = interior_angle_of_C = " << interior_angle_of_C << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).";
	output << "\ninterior_angle_CDA = interior_angle_of_D = " << interior_angle_of_D << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).";
	output << "\ninterior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = " << interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D << ". // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)";
	output << "\nget_perimeter() = a + b + c + d = " << get_perimeter() << ". // The method returns the sum of the four approximated side lengths of the trapezoid which the caller TRAPEZOID object represents.";
	output << "\nget_area() = " << get_area() << ". // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.";
	output << "\n--";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (i.e. <<).
 *
 * The friend function is not a member of the TRAPEZOID class,
 * but the friend function does have access to the private and protected members of the TRAPEZOID class as though
 * the friend function was a member of the TRAPEZOID class.
 */
std::ostream & operator << (std::ostream & output, TRAPEZOID & trapezoid)
{
	trapezoid.print(output);
	return output;
}

/**

* The destructor method of the TRAPEZOID class de-allocates memory which was used to
* instantiate the TRAPEZOID object which is calling this function.
*
* The destructor method of the TRAPEZOID class is automatically called when
* the program scope in which the caller TRAPEZOID object was instantiated terminates.
*/
TRAPEZOID::~TRAPEZOID()
{
std::cout << "\n\nDeleting the TRAPEZOID type object whose memory address is " << this << "...";
}

RECTANGLE_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the RECTANGLE class.

C++_header_file: https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/rectangle.h

---

/**
* file: rectangle.h
* type: C++ (header file)
* date: 07_JULY_2023
* author: karbytes
* license: PUBLIC_DOMAIN
*/

// If rectangle.h has not already been linked to a source file (.cpp), then link this header file to the source file(s) which include this header file.
#ifndef RECTANGLE_H
#define RECTANGLE_H

// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the QUADRILATERAL class.
#include "quadrilateral.h"

/**
* RECTANGLE is a class which inherits the protected and public data

```
 * attributes and methods of QUADRILATERAL.
 *
 * A RECTANGLE object represents a four-sided polygon such that opposite sides
 * are the same length and each interior angle measurement of that quadrilateral
 * is 90 degrees.
 *
 * Class members which are set to the protected access specifier
 * are accessible to the base class and to derived classes.
 *
 * Class members which are set to the private access specifier
 * are only accessible to the base class.
 *
 * Class members which are set to the public access specifier
 * are accessible to any scope within the program where
 * the base class and its derived classes are implemented.
 */
class RECTANGLE: public QUADRILATERAL
{
protected:

	/**
	* category is a description of the POLYGON instance.
	* category is set to a const (i.e. const (i.e. immutable)) value.
	*/
	const std::string category = "POLYGON/QUADRILATERAL/RECTANGLE";

	/**
	* The helper method determines whether or not the caller RECTANGLE instance
represents a
	* quadrilateral such that opposite sides are the same length and each interior angle
	* measurement of that quadrilateral is 90 degrees.
	* Return true if the caller RECTANGLE satisfies those conditions. Otherwise, return
false.
	*/
	bool is_rectangle();

public:

	/**
	* The test function helps to illustrate how pointers work.
	*/
	int rectangle_test(); // return 666

	/**
```

* The default constructor of the RECTANGLE class calls the constructor of the QUADRILATERAL class and
* sets the POINT type data member of the RECTANGLE object returned by this function named A to POINT(0,0),
* sets the POINT type data member of the RECTANGLE object returned by this function named B to POINT(0,3),
* sets the POINT type data member of the RECTANGLE object returned by this function named C to POINT(4,3), and
* sets the POINT type data member of the RECTANGLE object returned by this function named D to POINT(4,0).
*/
RECTANGLE();

/**
* The normal constructor of RECTANGLE attempts to set
* the string type data member of this to the input string type value named color and
* the POINT type data member of this named A to the input POINT type value named A and
* the POINT type data member of this named B to the input POINT type value named B and
* the POINT type data member of this named C to the input POINT type value named C and
* the POINT type data member of this named D to the input POINT type value named D.
*
* (The keyword this refers to the RECTANGLE object which is returned by this function).
*
* If A, B, C, and D represent unique points on a Cartesian plane and
* if the interior angles of the quadrilateral which those points would represent add up to 360 degrees and
* if the area of the quadrilateral which those points represents is larger than zero, and
* if each interior angle of that quadrilateral has an angle measurement of 90 degrees,
* use the input POINT values as the POINT values for the RECTANGLE object which is returned by this function.
*/
RECTANGLE(std::string color, POINT A, POINT B, POINT C, POINT D);

/**
* The copy constructor of RECTANGLE creates a clone of
* the input RECTANGLE instance.
*/
RECTANGLE(RECTANGLE & rectangle);

/**
* This method overrides the QUADRILATERAL class's print method.

```
	*
	* The descriptor method prints a description of the caller RECTANGLE instance to the
output stream.
	*
	* If no function input is supplied, output is set to the command line terminal.
	*/
	void print(std::ostream & output = std::cout);

	/**
	* The friend function is an alternative to the print method.
	* The friend function overloads the ostream operator (i.e. <<).
	*
	* The friend function is not a member of the RECTANGLE class,
	* but the friend function does have access to the private and protected members of the
RECTANGLE class as though
	* the friend function was a member of the RECTANGLE class.
	*/
	friend std::ostream & operator << (std::ostream & output, RECTANGLE & rectangle);

	/**
	* The destructor method of the RECTANGLE class de-allocates memory which was used
to
	* instantiate the RECTANGLE object which is calling this function.
	*
	* The destructor method of the RECTANGLE class is automatically called when
	* the program scope in which the caller RECTANGLE object was instantiated terminates.
	*/
	~RECTANGLE();
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

RECTANGLE_CLASS_SOURCE_CODE

The following source code defines the functions of the RECTANGLE class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/rectangle.cpp

---

```
/**
 * file: rectangle.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

// Include the C++ header file which contains preprocessing directives, variable declarations,
and function prototypes for the RECTANGLE class.
#include "rectangle.h"

/**
 * The helper method determines whether or not the caller RECTANGLE instance represents a
 * quadrilateral such that opposite sides are the same length and each interior angle
 * measurement of that quadrilateral is 90 degrees.
 * Return true if the caller RECTANGLE satisfies those conditions. Otherwise, return false.
 */
bool RECTANGLE::is_rectangle()
{
	double a = 0.0, b = 0.0, c = 0.0, d = 0.0;
	double a0 = 0.0, b0 = 0.0, c0 = 0.0;
	double a1 = 0.0, b1 = 0.0, c1 = 0.0;
	double angle_opposite_of_a0 = 0.0, angle_opposite_of_b0 = 0.0, angle_opposite_of_c0
= 0.0;
	double angle_opposite_of_a1 = 0.0, angle_opposite_of_b1 = 0.0, angle_opposite_of_c1
= 0.0;
	double interior_angle_of_A = 0.0, interior_angle_of_B = 0.0, interior_angle_of_C = 0.0,
interior_angle_of_D = 0.0;

	// first triangle
	a0 = A.get_distance_from(B);
	b0 = B.get_distance_from(D);
	c0 = D.get_distance_from(A);
	angle_opposite_of_a0 = acos(((b0 * b0) + (c0 * c0) - (a0 * a0)) / (2 * b0 * c0)) * (180 / PI);
	angle_opposite_of_b0 = acos(((a0 * a0) + (c0 * c0) - (b0 * b0)) / (2 * a0 * c0)) * (180 / PI);
	angle_opposite_of_c0 = acos(((a0 * a0) + (b0 * b0) - (c0 * c0)) / (2 * a0 * b0)) * (180 / PI);

	// second triangle
	a1 = D.get_distance_from(B);
	b1 = B.get_distance_from(C);
	c1 = C.get_distance_from(D);
	angle_opposite_of_a1 = acos(((b1 * b1) + (c1 * c1) - (a1 * a1)) / (2 * b1 * c1)) * (180 / PI);
```

```
	angle_opposite_of_b1 = acos(((a1 * a1) + (c1 * c1) - (b1 * b1)) / (2 * a1 * c1)) * (180 / PI);
	angle_opposite_of_c1 = acos(((a1 * a1) + (b1 * b1) - (c1 * c1)) / (2 * a1 * b1)) * (180 / PI);

	interior_angle_of_A = angle_opposite_of_b0;
	interior_angle_of_B = angle_opposite_of_c0 + angle_opposite_of_c1;
	interior_angle_of_C = angle_opposite_of_a1;
	interior_angle_of_D = angle_opposite_of_b1 + angle_opposite_of_a0;

	/* sides of quadrilateral */
	a = B.get_distance_from(C);
	b = C.get_distance_from(D);
	c = D.get_distance_from(A);
	d = A.get_distance_from(B);

	if (!points_represent_unique_coordinate_pairs(A,B,C,D) ||
!interior_angles_add_up_to_360_degrees()) return false;

	// Determine whether or not exactly one pair of opposite sides of the quadrilateral are
parallel to each other.
	if ((a == c) && (b == d))
	{
	if (!floor(interior_angle_of_A) == 90) return false;
	if (!floor(interior_angle_of_B) == 90) return false;
	if (!floor(interior_angle_of_C) == 90) return false;
	if (!floor(interior_angle_of_C) == 90) return false;
	return true;
	}
	return false;
}

/**
 * The test function helps to illustrate how pointers work.
 */
int RECTANGLE::rectangle_test()
{
	return 666;
}

/**
 * The default constructor of the RECTANGLE class calls the constructor of the
QUADRILATERAL class and
 * sets the POINT type data member of the RECTANGLE object returned by this function named
A to POINT(0,0),
```

```
 * sets the POINT type data member of the RECTANGLE object returned by this function named
B to POINT(0,3),
 * sets the POINT type data member of the RECTANGLE object returned by this function named
C to POINT(4,3), and
 * sets the POINT type data member of the RECTANGLE object returned by this function named
D to POINT(4,0).
 */
RECTANGLE::RECTANGLE()
{
	std::cout << "\n\nCreating the RECTANGLE type object whose memory address is " <<
this << "...";
	A = POINT(0,0);
	B = POINT(0,3);
	C = POINT(4,3);
	D = POINT(4,0);
}

/**
 * The normal constructor of RECTANGLE attempts to set
 * the string type data member of this to the input string type value named color and
 * the POINT type data member of this named A to the input POINT type value named A and
 * the POINT type data member of this named B to the input POINT type value named B and
 * the POINT type data member of this named C to the input POINT type value named C and
 * the POINT type data member of this named D to the input POINT type value named D.
 *
 * (The keyword this refers to the RECTANGLE object which is returned by this function).
 *
 * If A, B, C, and D represent unique points on a Cartesian plane and
 * if the interior angles of the quadrilateral which those points would represent add up to 360
degrees and
 * if the area of the quadrilateral which those points represents is larger than zero, and
 * if each interior angle of that quadrilateral has an angle measurement of 90 degrees,
 * use the input POINT values as the POINT values for the RECTANGLE object which is
returned by this function.
 */
RECTANGLE::RECTANGLE(std::string color, POINT A, POINT B, POINT C, POINT D)
{
	std::cout << "\n\nCreating the RECTANGLE type object whose memory address is " <<
this << "...";
	RECTANGLE test_rectangle;
	test_rectangle.A.set_X(A.get_X());
	test_rectangle.A.set_Y(A.get_Y());
	test_rectangle.B.set_X(B.get_X());
	test_rectangle.B.set_Y(B.get_Y());
```

```
	test_rectangle.C.set_X(C.get_X());
	test_rectangle.C.set_Y(C.get_Y());
	test_rectangle.D.set_X(D.get_X());
	test_rectangle.D.set_Y(D.get_Y());
	if (test_rectangle.is_rectangle())
	{
	this -> A = A;
	this -> B = B;
	this -> C = C;
	this -> D = D;
	}
	else
	{
	this -> A = POINT(0,0);
	this -> B = POINT(0,3);
	this -> C = POINT(4,3);
	this -> D = POINT(4,0);
	}
	this -> color = color;
}

/**
 * The copy constructor of RECTANGLE creates a clone of
 * the input RECTANGLE instance.
 */
RECTANGLE::RECTANGLE(RECTANGLE & rectangle)
{
	std::cout << "\n\nCreating the RECTANGLE type object whose memory address is " <<
this << "...";
	A = rectangle.A;
	B = rectangle.B;
	C = rectangle.C;
	D = rectangle.D;
	color = rectangle.color;
}

/**
 * This method overrides the QUADRILATERAL class's print method.
 *
 * The descriptor method prints a description of the caller RECTANGLE instance to the output
stream.
 *
 * If no function input is supplied, output is set to the command line terminal.
 */
```

```
void RECTANGLE::print(std::ostream & output)
{
	double a0 = 0.0, b0 = 0.0, c0 = 0.0;
	double a1 = 0.0, b1 = 0.0, c1 = 0.0;
	double angle_opposite_of_a0 = 0.0, angle_opposite_of_b0 = 0.0, angle_opposite_of_c0
= 0.0;
	double angle_opposite_of_a1 = 0.0, angle_opposite_of_b1 = 0.0, angle_opposite_of_c1
= 0.0;
	double interior_angle_of_A = 0.0, interior_angle_of_B = 0.0, interior_angle_of_C = 0.0,
interior_angle_of_D = 0.0;

	// first triangle
	a0 = A.get_distance_from(B);
	b0 = B.get_distance_from(D);
	c0 = D.get_distance_from(A);
	angle_opposite_of_a0 = acos(((b0 * b0) + (c0 * c0) - (a0 * a0)) / (2 * b0 * c0)) * (180 / PI);
	angle_opposite_of_b0 = acos(((a0 * a0) + (c0 * c0) - (b0 * b0)) / (2 * a0 * c0)) * (180 / PI);
	angle_opposite_of_c0 = acos(((a0 * a0) + (b0 * b0) - (c0 * c0)) / (2 * a0 * b0)) * (180 / PI);

	// second triangle
	a1 = D.get_distance_from(B);
	b1 = B.get_distance_from(C);
	c1 = C.get_distance_from(D);
	angle_opposite_of_a1 = acos(((b1 * b1) + (c1 * c1) - (a1 * a1)) / (2 * b1 * c1)) * (180 / PI);
	angle_opposite_of_b1 = acos(((a1 * a1) + (c1 * c1) - (b1 * b1)) / (2 * a1 * c1)) * (180 / PI);
	angle_opposite_of_c1 = acos(((a1 * a1) + (b1 * b1) - (c1 * c1)) / (2 * a1 * b1)) * (180 / PI);

	interior_angle_of_A = angle_opposite_of_b0;
	interior_angle_of_B = angle_opposite_of_c0 + angle_opposite_of_c1;
	interior_angle_of_C = angle_opposite_of_a1;
	interior_angle_of_D = angle_opposite_of_b1 + angle_opposite_of_a0;

	output <<
"\n\n----------------------------------------------------------------------------------------------";
	output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the
memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous
memory cells which are allocated to the caller TRIANGLE object.";
	output << "\n&category = " << &category << ". // The reference operation returns the
memory address of the first memory cell of a string sized chunk of contiguous memory cells
which are allocated to the string data attribute named category.";
	output << "\n&color = " << &color << ". // The reference operation returns the memory
address of the first memory cell of a string sized chunk of contiguous memory cells which are
allocated to the string data attribute named color..";
```

output << "\n&A = " << &A << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.";

output << "\n&B = " << &B << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.";

output << "\n&C = " << &C << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.";

output << "\n&D = " << &D << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.";

output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(int *) = " << sizeof(int *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(int **) = " << sizeof(int **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string) = " << sizeof(std::string) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string *) = " << sizeof(std::string *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string **) = " << sizeof(std::string **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT) = " << sizeof(POINT) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT *) = " << sizeof(POINT *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT **) = " << sizeof(POINT **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON) = " << sizeof(POLYGON) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON *) = " << sizeof(POLYGON *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON **) = " << sizeof(POLYGON **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL) = " << sizeof(QUADRILATERAL) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL *) = " << sizeof(QUADRILATERAL *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL **) = " << sizeof(QUADRILATERAL **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(RECTANGLE) = " << sizeof(RECTANGLE) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(RECTANGLE *) = " << sizeof(RECTANGLE *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(RECTANGLE **) = " << sizeof(RECTANGLE **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\ncategory = " << category << ". // This is an immutable string type value which is a data member of the caller RECTANGLE object.";

output << "\ncolor = " << color << ". // This is a string type value which is a data member of the caller RECTANGLE object.";

output << "\nA = POINT(" << A.get_X() << "," << A.get_Y() << "). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nB = POINT(" << B.get_X() << "," << B.get_Y() << "). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nC = POINT(" << C.get_X() << "," << C.get_Y() << "). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nD = POINT(" << D.get_X() << "," << D.get_Y() << "). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\na = B.get_distance_from(C) = " << B.get_distance_from(C) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.";

output << "\nb = C.get_distance_from(D) = " << C.get_distance_from(D) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.";

output << "\nc = D.get_distance_from(A) = " << D.get_distance_from(A) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.";

output << "\nd = A.get_distance_from(B) = " << A.get_distance_from(B) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.";

output << "\nA.get_slope_of_line_to(B) = " << A.get_slope_of_line_to(B) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.";

output << "\nB.get_slope_of_line_to(C) = " << B.get_slope_of_line_to(C) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\nC.get_slope_of_line_to(D) = " << C.get_slope_of_line_to(D) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.";

output << "\nD.get_slope_of_line_to(A) = " << D.get_slope_of_line_to(A) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\ninterior_angle_DAB = interior_angle_of_A = " << interior_angle_of_A << ". // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_ABC = interior_angle_of_B = " << interior_angle_of_B << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).";

```
	output << "\ninterior_angle_BCD = interior_angle_of_C = " << interior_angle_of_C << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).";
	output << "\ninterior_angle_CDA = interior_angle_of_D = " << interior_angle_of_D << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).";
	output << "\ninterior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = " << interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D << ". // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)";
	output << "\nget_perimeter() = a + b + c + d = " << get_perimeter() << ". // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.";
	output << "\nget_area() = " << get_area() << ". // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.";
	output << "\n--------------------------------------------------------------------------------";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (i.e. <<).
 *
 * The friend function is not a member of the RECTANGLE class,
 * but the friend function does have access to the private and protected members of the RECTANGLE class as though
 * the friend function was a member of the RECTANGLE class.
 */
std::ostream & operator << (std::ostream & output, RECTANGLE & rectangle)
{
	rectangle.print(output);
	return output;
}

/**
 * The destructor method of the RECTANGLE class de-allocates memory which was used to
 * instantiate the RECTANGLE object which is calling this function.
 *
```

```
 * The destructor method of the RECTANGLE class is automatically called when
 * the program scope in which the caller RECTANGLE object was instantiated terminates.
 */
RECTANGLE::~RECTANGLE()
{
	std::cout << "\n\nDeleting the RECTANGLE type object whose memory address is " << this << "...";
}
```

SQUARE_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the SQUARE class.

C++_header_file: https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/square.h

---

```
/**
 * file: square.h
 * type: C++ (header file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

// If square.h has not already been linked to a source file (.cpp), then link this header file to the source file(s) which include this header file.
#ifndef SQUARE_H
#define SQUARE_H

// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the RECTANGLE class.
#include "rectangle.h"

/**
 * SQUARE is a class which inherits the protected and public data
 * attributes and methods of RECTANGLE.
 *
 * A SQUARE object represents a four-sided polygon such that each side of that
```

```
 * quadrilateral is the same length and each interior angle measurement of that
 * quadrilateral is 90 degrees.
 *
 * Class members which are set to the protected access specifier
 * are accessible to the base class and to derived classes.
 *
 * Class members which are set to the private access specifier
 * are only accessible to the base class.
 *
 * Class members which are set to the public access specifier
 * are accessible to any scope within the program where
 * the base class and its derived classes are implemented.
 */
class SQUARE: public RECTANGLE
{
protected:

	/**
	* category is a description of the POLYGON instance.
	* category is set to a const (i.e. const (i.e. immutable)) value.
	*/
	const std::string category = "POLYGON/QUADRILATERAL/RECTANGLE/SQUARE";

	/**
	* The helper method determines whether or not the caller SQUARE instance represents
a
	* quadrilateral such that each side is the same length and each interior angle
measurement of that is 90 degrees.
	* Return true if the caller SQUARE satisfies those conditions. Otherwise, return false.
	*/
	bool is_square();

public:

	/**
	* The default constructor of the SQUARE class calls the constructor of the RECTANGLE
class and
	* sets the POINT type data member of the SQUARE object returned by this function
named A to POINT(0,0),
	* sets the POINT type data member of the SQUARE object returned by this function
named B to POINT(0,5),
	* sets the POINT type data member of the SQUARE object returned by this function
named C to POINT(5,5), and
```

* sets the POINT type data member of the SQUARE object returned by this function named D to POINT(5,0).
*/
SQUARE();

/**
* The normal constructor of SQUARE attempts to set
* the string type data member of this to the input string type value named color and
* the POINT type data member of this named A to the input POINT type value named A and
* the POINT type data member of this named B to the input POINT type value named B and
* the POINT type data member of this named C to the input POINT type value named C and
* the POINT type data member of this named D to the input POINT type value named D.
*
* (The keyword this refers to the SQUARE object which is returned by this function).
*
* If A, B, C, and D represent unique points on a Cartesian plane and
* if the interior angles of the quadrilateral which those points would represent add up to 360 degrees and
* if the area of the quadrilateral which those points represents is larger than zero,
* if each interior angle of that quadrilateral has an angle measurement of 90 degrees, and
* if each side of that quadrilateral has the same length,
* use the input POINT values as the POINT values for the SQUARE object which is returned by this function.
*/
SQUARE(std::string color, POINT A, POINT B, POINT C, POINT D);

/**
* The copy constructor of SQUARE creates a clone of
* the input SQUARE instance.
*/
SQUARE(SQUARE & square);

/**
* This method overrides the RECTANGLE class's print method.
*
* The descriptor method prints a description of the caller SQUARE instance to the output stream.
*
* If no function input is supplied, output is set to the command line terminal.
*/

```
	void print(std::ostream & output = std::cout);

	/**
	* The friend function is an alternative to the print method.
	* The friend function overloads the ostream operator (i.e. <<).
	*
	* The friend function is not a member of the SQUARE class,
	* but the friend function does have access to the private and protected members of the 
SQUARE class as though
	* the friend function was a member of the SQUARE class.
	*/
	friend std::ostream & operator << (std::ostream & output, SQUARE & square);

	/**
	* The destructor method of the SQUARE class de-allocates memory which was used to
	* instantiate the SQUARE object which is calling this function.
	*
	* The destructor method of the SQUARE class is automatically called when
	* the program scope in which the caller SQUARE object was instantiated terminates.
	*/
	~SQUARE();
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

SQUARE_CLASS_SOURCE_CODE

The following source code defines the functions of the SQUARE class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/square.cpp

---

```
/**
 * file: square.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
```

```
 */

// Include the C++ header file which contains preprocessing directives, variable declarations,
and function prototypes for the SQUARE class.
#include "square.h"

/**
 * The helper method determines whether or not the caller SQUARE instance represents a
 * quadrilateral such that each side is the same length and each interior angle measurement of
that is 90 degrees.
 * Return true if the caller SQUARE satisfies those conditions. Otherwise, return false.
 */
bool SQUARE::is_square()
{
	double a = 0.0, b = 0.0, c = 0.0, d = 0.0;
	double a0 = 0.0, b0 = 0.0, c0 = 0.0;
	double a1 = 0.0, b1 = 0.0, c1 = 0.0;
	double angle_opposite_of_a0 = 0.0, angle_opposite_of_b0 = 0.0, angle_opposite_of_c0
= 0.0;
	double angle_opposite_of_a1 = 0.0, angle_opposite_of_b1 = 0.0, angle_opposite_of_c1
= 0.0;
	double interior_angle_of_A = 0.0, interior_angle_of_B = 0.0, interior_angle_of_C = 0.0,
interior_angle_of_D = 0.0;

	// first triangle
	a0 = A.get_distance_from(B);
	b0 = B.get_distance_from(D);
	c0 = D.get_distance_from(A);
	angle_opposite_of_a0 = acos(((b0 * b0) + (c0 * c0) - (a0 * a0)) / (2 * b0 * c0)) * (180 / PI);
	angle_opposite_of_b0 = acos(((a0 * a0) + (c0 * c0) - (b0 * b0)) / (2 * a0 * c0)) * (180 / PI);
	angle_opposite_of_c0 = acos(((a0 * a0) + (b0 * b0) - (c0 * c0)) / (2 * a0 * b0)) * (180 / PI);

	// second triangle
	a1 = D.get_distance_from(B);
	b1 = B.get_distance_from(C);
	c1 = C.get_distance_from(D);
	angle_opposite_of_a1 = acos(((b1 * b1) + (c1 * c1) - (a1 * a1)) / (2 * b1 * c1)) * (180 / PI);
	angle_opposite_of_b1 = acos(((a1 * a1) + (c1 * c1) - (b1 * b1)) / (2 * a1 * c1)) * (180 / PI);
	angle_opposite_of_c1 = acos(((a1 * a1) + (b1 * b1) - (c1 * c1)) / (2 * a1 * b1)) * (180 / PI);

	interior_angle_of_A = angle_opposite_of_b0;
	interior_angle_of_B = angle_opposite_of_c0 + angle_opposite_of_c1;
	interior_angle_of_C = angle_opposite_of_a1;
	interior_angle_of_D = angle_opposite_of_b1 + angle_opposite_of_a0;
```

```
        /* sides of quadrilateral */
        a = B.get_distance_from(C);
        b = C.get_distance_from(D);
        c = D.get_distance_from(A);
        d = A.get_distance_from(B);

        if (!is_rectangle()) return false;

        // Determine whether each side of the quadrilateral has the same length.
        if ((a == b) && (b == c) && (c == d) && (d == a)) return true;
        return false;
}

/**
 * The default constructor of the SQUARE class calls the constructor of the RECTANGLE class
and
 * sets the POINT type data member of the SQUARE object returned by this function named A
to POINT(0,0),
 * sets the POINT type data member of the SQUARE object returned by this function named B
to POINT(0,5),
 * sets the POINT type data member of the SQUARE object returned by this function named C
to POINT(5,5), and
 * sets the POINT type data member of the SQUARE object returned by this function named D
to POINT(5,0).
 */
SQUARE::SQUARE()
{
        std::cout << "\n\nCreating the SQUARE type object whose memory address is " << this
<< "...";
        A = POINT(0,0);
        B = POINT(0,5);
        C = POINT(5,5);
        D = POINT(5,0);
}

/**
 * The normal constructor of SQUARE attempts to set
 * the string type data member of this to the input string type value named color and
 * the POINT type data member of this named A to the input POINT type value named A and
 * the POINT type data member of this named B to the input POINT type value named B and
 * the POINT type data member of this named C to the input POINT type value named C and
 * the POINT type data member of this named D to the input POINT type value named D.
 *
```

```
 * (The keyword this refers to the SQUARE object which is returned by this function).
 *
 * If A, B, C, and D represent unique points on a Cartesian plane and
 * if the interior angles of the quadrilateral which those points would represent add up to 360
degrees and
 * if the area of the quadrilateral which those points represents is larger than zero,
 * if each interior angle of that quadrilateral has an angle measurement of 90 degrees, and
 * if each side of that quadrilateral has the same length,
 * use the input POINT values as the POINT values for the SQUARE object which is returned by
this function.
 */
SQUARE::SQUARE(std::string color, POINT A, POINT B, POINT C, POINT D)
{
	std::cout << "\n\nCreating the SQUARE type object whose memory address is " << this
<< "...";
	SQUARE test_square;
	test_square.A.set_X(A.get_X());
	test_square.A.set_Y(A.get_Y());
	test_square.B.set_X(B.get_X());
	test_square.B.set_Y(B.get_Y());
	test_square.C.set_X(C.get_X());
	test_square.C.set_Y(C.get_Y());
	test_square.D.set_X(D.get_X());
	test_square.D.set_Y(D.get_Y());
	if (test_square.is_square())
	{
	this -> A = A;
	this -> B = B;
	this -> C = C;
	this -> D = D;
	}
	else
	{
	this -> A = POINT(0,0);
	this -> B = POINT(0,3);
	this -> C = POINT(4,3);
	this -> D = POINT(4,0);
	}
	this -> color = color;
}

/**
 * The copy constructor of SQUARE creates a clone of
 * the input SQUARE instance.
```

```
 */
SQUARE::SQUARE(SQUARE & square)
{
	std::cout << "\n\nCreating the SQUARE type object whose memory address is " << this
<< "...";
	A = square.A;
	B = square.B;
	C = square.C;
	D = square.D;
	color = square.color;
}

/**
 * This method overrides the RECTANGLE class's print method.
 *
 * The descriptor method prints a description of the caller SQUARE instance to the output
stream.
 *
 * If no function input is supplied, output is set to the command line terminal.
 */
void SQUARE::print(std::ostream & output)
{
	double a0 = 0.0, b0 = 0.0, c0 = 0.0;
	double a1 = 0.0, b1 = 0.0, c1 = 0.0;
	double angle_opposite_of_a0 = 0.0, angle_opposite_of_b0 = 0.0, angle_opposite_of_c0
= 0.0;
	double angle_opposite_of_a1 = 0.0, angle_opposite_of_b1 = 0.0, angle_opposite_of_c1
= 0.0;
	double interior_angle_of_A = 0.0, interior_angle_of_B = 0.0, interior_angle_of_C = 0.0,
interior_angle_of_D = 0.0;

	// first triangle
	a0 = A.get_distance_from(B);
	b0 = B.get_distance_from(D);
	c0 = D.get_distance_from(A);
	angle_opposite_of_a0 = acos(((b0 * b0) + (c0 * c0) - (a0 * a0)) / (2 * b0 * c0)) * (180 / PI);
	angle_opposite_of_b0 = acos(((a0 * a0) + (c0 * c0) - (b0 * b0)) / (2 * a0 * c0)) * (180 / PI);
	angle_opposite_of_c0 = acos(((a0 * a0) + (b0 * b0) - (c0 * c0)) / (2 * a0 * b0)) * (180 / PI);

	// second triangle
	a1 = D.get_distance_from(B);
	b1 = B.get_distance_from(C);
	c1 = C.get_distance_from(D);
	angle_opposite_of_a1 = acos(((b1 * b1) + (c1 * c1) - (a1 * a1)) / (2 * b1 * c1)) * (180 / PI);
```

```
	angle_opposite_of_b1 = acos(((a1 * a1) + (c1 * c1) - (b1 * b1)) / (2 * a1 * c1)) * (180 / PI);
	angle_opposite_of_c1 = acos(((a1 * a1) + (b1 * b1) - (c1 * c1)) / (2 * a1 * b1)) * (180 / PI);

	interior_angle_of_A = angle_opposite_of_b0;
	interior_angle_of_B = angle_opposite_of_c0 + angle_opposite_of_c1;
	interior_angle_of_C = angle_opposite_of_a1;
	interior_angle_of_D = angle_opposite_of_b1 + angle_opposite_of_a0;

	output << 
"\n\n----------------------------------------------------------------------------------------------------";
	output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.";
	output << "\n&category = " << &category << ". // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.";
	output << "\n&color = " << &color << ". // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..";
	output << "\n&A = " << &A << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.";
	output << "\n&B = " << &B << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.";
	output << "\n&C = " << &C << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.";
	output << "\n&D = " << &D << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.";
	output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(int *) = " << sizeof(int *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(int **) = " << sizeof(int **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(std::string) = " << sizeof(std::string) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).";
```

output << "\nsizeof(std::string *) = " << sizeof(std::string *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string **) = " << sizeof(std::string **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT) = " << sizeof(POINT) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT *) = " << sizeof(POINT *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT **) = " << sizeof(POINT **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON) = " << sizeof(POLYGON) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON *) = " << sizeof(POLYGON *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON **) = " << sizeof(POLYGON **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL) = " << sizeof(QUADRILATERAL) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL *) = " << sizeof(QUADRILATERAL *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(QUADRILATERAL **) = " << sizeof(QUADRILATERAL **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(RECTANGLE) = " << sizeof(RECTANGLE) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(RECTANGLE *) = " << sizeof(RECTANGLE *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(RECTANGLE **) = " << sizeof(RECTANGLE **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(SQUARE) = " << sizeof(SQUARE) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a SQUARE type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(SQUARE *) = " << sizeof(SQUARE *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(SQUARE **) = " << sizeof(SQUARE **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\ncategory = " << category << ". // This is an immutable string type value which is a data member of the caller RECTANGLE object.";

output << "\ncolor = " << color << ". // This is a string type value which is a data member of the caller RECTANGLE object.";

output << "\nA = POINT(" << A.get_X() << "," << A.get_Y() << "). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nB = POINT(" << B.get_X() << "," << B.get_Y() << "). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nC = POINT(" << C.get_X() << "," << C.get_Y() << "). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nD = POINT(" << D.get_X() << "," << D.get_Y() << "). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\na = B.get_distance_from(C) = " << B.get_distance_from(C) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.";

output << "\nb = C.get_distance_from(D) = " << C.get_distance_from(D) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.";

output << "\nc = D.get_distance_from(A) = " << D.get_distance_from(A) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.";

output << "\nd = A.get_distance_from(B) = " << A.get_distance_from(B) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.";

output << "\nA.get_slope_of_line_to(B) = " << A.get_slope_of_line_to(B) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.";

output << "\nB.get_slope_of_line_to(C) = " << B.get_slope_of_line_to(C) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\nC.get_slope_of_line_to(D) = " << C.get_slope_of_line_to(D) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.";

output << "\nD.get_slope_of_line_to(A) = " << D.get_slope_of_line_to(A) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\ninterior_angle_DAB = interior_angle_of_A = " << interior_angle_of_A << ". // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_ABC = interior_angle_of_B = " << interior_angle_of_B << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_BCD = interior_angle_of_C = " << interior_angle_of_C << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_CDA = interior_angle_of_D = " << interior_angle_of_D << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = " << interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D << ". // sum of all four approximate interior angle measurements of the

quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)";
	output << "\nget_perimeter() = a + b + c + d = " << get_perimeter() << ". // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.";
	output << "\nget_area() = " << get_area() << ". // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.";
	output << "\n--";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (i.e. <<).
 *
 * The friend function is not a member of the SQUARE class,
 * but the friend function does have access to the private and protected members of the SQUARE class as though
 * the friend function was a member of the SQUARE class.
 */
std::ostream & operator << (std::ostream & output, SQUARE & square)
{
	square.print(output);
	return output;
}

/**
 * The destructor method of the SQUARE class de-allocates memory which was used to
 * instantiate the SQUARE object which is calling this function.
 *
 * The destructor method of the SQUARE class is automatically called when
 * the program scope in which the caller SQUARE object was instantiated terminates.
 */
SQUARE::~SQUARE()
{
	std::cout << "\n\nDeleting the SQUARE type object whose memory address is " << this << "...";
}

TRILATERAL_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the TRILATERAL class.

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/trilateral.h

---

/**
 * file: trilateral.h
 * type: C++ (header file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

// If trilateral.h has not already been linked to a source file (.cpp), then link this header file to the source file(s) which include this header file.
#ifndef TRILATERAL_H
#define TRILATERAL_H

// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the POLYGON class.
#include "polygon.h"

/**
 * TRILATERAL is a class which inherits the protected and public data
 * attributes and methods of POLYGON (and POLYGON is an abstract class).
 *
 * A TRILATERAL object represents an instance in which three unique POINT instances exist
 * such that each one of those three POINT instances represents a unique coordinate pair within the tuple of three POINT instances
 * (such that each coordinate pair represents exactly one two-dimensional point, POINT(X,Y), on a Cartesian grid).
 *
 * Each TRILATERAL object represents a specific three-sided polygon whose area is a positive real number.
 *
 * (A synonym for "trilateral" is "triangle").
 *
 * Class members which are set to the protected access specifier
 * are accessible to the base class and to derived classes.
 *

```
 * Class members which are set to the private access specifier
 * are only accessible to the base class.
 *
 * Class members which are set to the public access specifier
 * are accessible to any scope within the program where
 * the base class and its derived classes are implemented.
 */
class TRILATERAL: public POLYGON
{
protected:

	/**
	* category is a description of the POLYGON instance.
	* category is set to a constant (i.e. immutable) string type value.
	*/
	const std::string category = "POLYGON/TRILATERAL";

	/**
	* POINT type objects A, B, and C represent points on a Cartesian plane.
	* Each POINT type object has two int type variables for representing a two-dimensional
whole number coordinate pair.
	* The X data attribute of a POINT object represents a whole number position on the
horizontal axis (i.e. x-axis) of a Cartesian plane.
	* The Y data attribute of a POINT object represents a whole number position on the
vertical axis (i.e. y-axis) of the same Cartesian plane.
	*/
	POINT A, B, C;

	/**
	* If each of the three whole number coordinate pairs represented by the POINT type
input values named _A, _B, and _C are unique whole number coordinate pairs,
	* return true.
	* Otherwise, return false.
	*/
	bool points_represent_unique_coordinate_pairs(POINT _A, POINT _B, POINT _C);

	/**
	* The getter method of the TRILATERAL class named get_interior_angle_ABC() returns
the approximate angle measurement in degrees of the angle
	* formed by connecting points A, B, anc C in the order specified by this sentence.
	*
	* The function below uses the Law of Cosines to compute the measurement of an
interior angle of a triangle
```

* using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
*/
double get_interior_angle_ABC();

/**
* The getter method of the TRILATERAL class named get_interior_angle_BCA() returns the approximate angle measurement in degrees of the angle
* formed by connecting points B, C, and A in the order specified by this sentence.
*
* The function below uses the Law of Cosines to compute the measurement of an interior angle of a triangle
* using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
*/
double get_interior_angle_BCA();

/**
* The getter method of the TRILATERAL class named get_interior_angle_CAB() returns the approximate angle measurement in degrees of the angle
* formed by connecting points C, A, and B in the order specified by this sentence.
*
* The function below uses Law of Cosines to compute the measurement of an interior angle of a triangle
* using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
*/
double get_interior_angle_CAB();

/**
* If sum of the interior angle measurements of the quadrilateral which the caller TRILATERAL object represents add up to approximately 180 degrees,
* return true.
* Otherwise, return false.
*/
bool interior_angles_add_up_to_180_degrees();

public:

/**
* The default constructor of the TRILATERAL class calls the constructor of the POLYGON class and
* sets the POINT type data member of the TRILATERAL object returned by this function named A to POINT(0,0),

* sets the POINT type data member of the TRILATERAL object returned by this function named B to POINT(4,3), and
* sets the POINT type data member of the TRILATERAL object returned by this function named C to POINT(4,0).
*/
TRILATERAL();

/**
* The normal constructor of TRILATERAL attempts to set
* the string type data member of this to the input string type value named color and
* the POINT type data member of this named A to the input POINT type value named A and
* the POINT type data member of this named B to the input POINT type value named B and
* the POINT type data member of this named C to the input POINT type value named C.
*
* (The keyword this refers to the TRILATERAL object which is returned by this function).
*
* If A, B, and C represent unique points on a Cartesian plane,
* if the interior angles of the trilatreal which those points would represent add up to 180 degrees, and
* if the area of the trilateral which those points represents is larger than zero,
*/
TRILATERAL(std::string color, POINT A, POINT B, POINT C);

/**
* The copy constructor method of the TRILATERAL class
* instantiates TRILATERAL type objects
* whose A value is set to the A value of the input TRILATERAL object,
* whose B value is set to the B value of the input TRILATERAL object, and
* whose C value is set to the C value of the input TRILATERAL object.
*/
TRILATERAL(TRILATERAL & trilateral);

/**
* The TRILATERAL class implements the virtual get_area() method of the POLYGON class.
*
* The getter method returns the approximate area of the two-dimensional space whose bounds are
* the shortest paths between points A, B, and C of the triangle which the caller TRILATERAL object represents.
* This function uses Heron's Formula to compute the area of a triangle using that triangle's side lengths as formula inputs.

*/
double get_area();

/**
* The TRILATERAL class implements the virtual get_perimeter() method of the POLYGON class.
*
* The getter method returns the perimeter of the trilateral represented by the caller TRILATERAL object
* by adding up the three side lengths of that trilateral.
*
* Let AB be the length of the line segment whose endpoints are A and B.
* Let BC be the length of the line segment whose endpoints are B and C.
* Let CA be the length of the line segment whose endpoints are C and A.
*
* Then return the sum of AB, BC, and CA.
*/
double get_perimeter();

/**
* This method overrides the POLYGON class's print method.
*
* The descriptor method prints a description of the caller TRILATERAL instance to the output stream.
*
* If no function input is supplied, output is set to the command line terminal.
*/
void print(std::ostream & output = std::cout);

/**
* The friend function is an alternative to the print method.
* The friend function overloads the ostream operator (i.e. <<).
*
* The friend function is not a member of the TRILATERAL class,
* but the friend function does have access to the private and protected members of the TRILATERAL class as though
* the friend function was a member of the TRILATERAL class.
*/
friend std::ostream & operator << (std::ostream & output, TRILATERAL & trilateral);

/**
* The destructor method of the TRILATERAL class de-allocates memory which was used to
* instantiate the TRILATERAL object which is calling this function.

```
	*
	* The destructor method of the TRILATERAL class is automatically called when
	* the program scope in which the caller TRILATERAL object was instantiated terminates.
	*/
	~TRILATERAL();
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

```
/**
 * file: trilateral.h
 * type: C++ (header file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

// If trilateral.h has not already been linked to a source file (.cpp), then link this header file to the source file(s) which include this header file.
#ifndef TRILATERAL_H
#define TRILATERAL_H

// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the POLYGON class.
#include "polygon.h"

/**
 * TRILATERAL is a class which inherits the protected and public data
 * attributes and methods of POLYGON (and POLYGON is an abstract class).
 *
 * A TRILATERAL object represents an instance in which three unique POINT instances exist
 * such that each one of those three POINT instances represents a unique coordinate pair within the tuple of three POINT instances
 * (such that each coordinate pair represents exactly one two-dimensional point, POINT(X,Y), on a Cartesian grid).
 *
 * Each TRILATERAL object represents a specific three-sided polygon whose area is a positive real number.
 *
 * (A synonym for "trilateral" is "triangle").
 *
```

```
 * Class members which are set to the protected access specifier
 * are accessible to the base class and to derived classes.
 *
 * Class members which are set to the private access specifier
 * are only accessible to the base class.
 *
 * Class members which are set to the public access specifier
 * are accessible to any scope within the program where
 * the base class and its derived classes are implemented.
 */
class TRILATERAL: public POLYGON
{
protected:

	/**
	* category is a description of the POLYGON instance.
	* category is set to a constant (i.e. immutable) string type value.
	*/
	const std::string category = "POLYGON/TRILATERAL";

	/**
	* POINT type objects A, B, and C represent points on a Cartesian plane.
	* Each POINT type object has two int type variables for representing a two-dimensional whole number coordinate pair.
	* The X data attribute of a POINT object represents a whole number position on the horizontal axis (i.e. x-axis) of a Cartesian plane.
	* The Y data attribute of a POINT object represents a whole number position on the vertical axis (i.e. y-axis) of the same Cartesian plane.
	*/
	POINT A, B, C;

	/**
	* If each of the three whole number coordinate pairs represented by the POINT type input values named _A, _B, and _C are unique whole number coordinate pairs,
	* return true.
	* Otherwise, return false.
	*/
	bool points_represent_unique_coordinate_pairs(POINT _A, POINT _B, POINT _C);

	/**
	* The getter method of the TRILATERAL class named get_interior_angle_ABC() returns the approximate angle measurement in degrees of the angle
	* formed by connecting points A, B, anc C in the order specified by this sentence.
	*
```

* The function below uses the Law of Cosines to compute the measurement of an interior angle of a triangle
* using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
*/
double get_interior_angle_ABC();

/**
* The getter method of the TRILATERAL class named get_interior_angle_BCA() returns the approximate angle measurement in degrees of the angle
* formed by connecting points B, C, and A in the order specified by this sentence.
*
* The function below uses the Law of Cosines to compute the measurement of an interior angle of a triangle
* using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
*/
double get_interior_angle_BCA();

/**
* The getter method of the TRILATERAL class named get_interior_angle_CAB() returns the approximate angle measurement in degrees of the angle
* formed by connecting points C, A, and B in the order specified by this sentence.
*
* The function below uses Law of Cosines to compute the measurement of an interior angle of a triangle
* using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
*/
double get_interior_angle_CAB();

/**
* If sum of the interior angle measurements of the quadrilateral which the caller TRILATERAL object represents add up to approximately 180 degrees,
* return true.
* Otherwise, return false.
*/
bool interior_angles_add_up_to_180_degrees();

public:

/**
* The default constructor of the TRILATERAL class calls the constructor of the POLYGON class and

* sets the POINT type data member of the TRILATERAL object returned by this function named A to POINT(0,0),
* sets the POINT type data member of the TRILATERAL object returned by this function named B to POINT(4,3), and
* sets the POINT type data member of the TRILATERAL object returned by this function named C to POINT(4,0).
*/
TRILATERAL();

/**
* The normal constructor of TRILATERAL attempts to set
* the string type data member of this to the input string type value named color and
* the POINT type data member of this named A to the input POINT type value named A and
* the POINT type data member of this named B to the input POINT type value named B and
* the POINT type data member of this named C to the input POINT type value named C.
*
* (The keyword this refers to the TRILATERAL object which is returned by this function).
*
* If A, B, and C represent unique points on a Cartesian plane,
* if the interior angles of the trilatreal which those points would represent add up to 180 degrees, and
* if the area of the trilateral which those points represents is larger than zero,
*/
TRILATERAL(std::string color, POINT A, POINT B, POINT C);

/**
* The copy constructor method of the TRILATERAL class
* instantiates TRILATERAL type objects
* whose A value is set to the A value of the input TRILATERAL object,
* whose B value is set to the B value of the input TRILATERAL object, and
* whose C value is set to the C value of the input TRILATERAL object.
*/
TRILATERAL(TRILATERAL & trilateral);

/**
* The TRILATERAL class implements the virtual get_area() method of the POLYGON class.
*
* The getter method returns the approximate area of the two-dimensional space whose bounds are
* the shortest paths between points A, B, and C of the triangle which the caller TRILATERAL object represents.

```
* This function uses Heron's Formula to compute the area of a triangle using that triangle's side lengths as formula inputs.
*/
double get_area();

/**
* The TRILATERAL class implements the virtual get_perimeter() method of the POLYGON class.
*
* The getter method returns the perimeter of the trilateral represented by the caller TRILATERAL object
* by adding up the three side lengths of that trilateral.
*
* Let AB be the length of the line segment whose endpoints are A and B.
* Let BC be the length of the line segment whose endpoints are B and C.
* Let CA be the length of the line segment whose endpoints are C and A.
*
* Then return the sum of AB, BC, and CA.
*/
double get_perimeter();

/**
* This method overrides the POLYGON class's print method.
*
* The descriptor method prints a description of the caller TRILATERAL instance to the output stream.
*
* If no function input is supplied, output is set to the command line terminal.
*/
void print(std::ostream & output = std::cout);

/**
* The friend function is an alternative to the print method.
* The friend function overloads the ostream operator (i.e. <<).
*
* The friend function is not a member of the TRILATERAL class,
* but the friend function does have access to the private and protected members of the TRILATERAL class as though
* the friend function was a member of the TRILATERAL class.
*/
friend std::ostream & operator << (std::ostream & output, TRILATERAL & trilateral);

/**
```

```
        * The destructor method of the TRILATERAL class de-allocates memory which was used
to
        * instantiate the TRILATERAL object which is calling this function.
        *
        * The destructor method of the TRILATERAL class is automatically called when
        * the program scope in which the caller TRILATERAL object was instantiated terminates.
        */
        ~TRILATERAL();
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

```
/**
 * file: trilateral.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

// Include the C++ header file which contains preprocessing directives, variable declarations,
and function prototypes for the TRILATERAL class.
#include "trilateral.h"

/**
 * If each of the three whole number coordinate pairs represented by the POINT type input
values named _A, _B, and _C are unique whole number coordinate pairs,
 * return true.
 * Otherwise, return false.
 */
bool TRILATERAL::points_represent_unique_coordinate_pairs(POINT _A, POINT _B, POINT
_C)
{
   if ((_A.get_X() == _B.get_X()) && (_A.get_Y() == _B.get_Y())) return false;
   if ((_A.get_X() == _C.get_X()) && (_A.get_Y() == _C.get_Y())) return false;
   if ((_B.get_X() == _C.get_X()) && (_B.get_Y() == _C.get_Y())) return false;
   return true;
}

/**
```

```
 * The getter method of the TRILATERAL class named get_interior_angle_ABC() returns the approximate angle measurement in degrees of the angle
 * formed by connecting points A, B, anc C in the order specified by this sentence.
 *
 * The function below uses the Law of Cosines to compute the measurement of an interior angle of a triangle
 * using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
 */
double TRILATERAL::get_interior_angle_ABC()
{
    double a = 0.0, b = 0.0, c = 0.0, angle_opposite_of_a = 0.0, angle_opposite_of_b = 0.0, angle_opposite_of_c = 0.0;
    a = B.get_distance_from(C); // a represents the length of the line segment whose endpoints are B and C.
    b = C.get_distance_from(A); // b represents the length of the line segment whose endpoints are C and A.
    c = A.get_distance_from(B); // c represents the length of the line segment whose endpoints are A and B.
    angle_opposite_of_a = acos(((b * b) + (c * c) - (a * a)) / (2 * b * c)) * (180 / PI);
    angle_opposite_of_b = acos(((a * a) + (c * c) - (b * b)) / (2 * a * c)) * (180 / PI);
    angle_opposite_of_c = acos(((a * a) + (b * b) - (c * c)) / (2 * a * b)) * (180 / PI);
    return angle_opposite_of_b;
}

/**
 * The getter method of the TRILATERAL class named get_interior_angle_BCA() returns the approximate angle measurement in degrees of the angle
 * formed by connecting points B, C, and A in the order specified by this sentence.
 *
 * The function below uses the Law of Cosines to compute the measurement of an interior angle of a triangle
 * using that triangle's three side lengths as function inputs to output some nonnegative real number of degrees.
 */
double TRILATERAL::get_interior_angle_BCA()
{
    double a = 0.0, b = 0.0, c = 0.0, angle_opposite_of_a = 0.0, angle_opposite_of_b = 0.0, angle_opposite_of_c = 0.0;
    a = B.get_distance_from(C); // a represents the length of the line segment whose endpoints are B and C.
    b = C.get_distance_from(A); // b represents the length of the line segment whose endpoints are C and A.
```

```
    c = A.get_distance_from(B); // c represents the length of the line segment whose endpoints
are A and B.
    angle_opposite_of_a = acos(((b * b) + (c * c) - (a * a)) / (2 * b * c)) * (180 / PI);
    angle_opposite_of_b = acos(((a * a) + (c * c) - (b * b)) / (2 * a * c)) * (180 / PI);
    angle_opposite_of_c = acos(((a * a) + (b * b) - (c * c)) / (2 * a * b)) * (180 / PI);
    return angle_opposite_of_c;
}

/**
 * The getter method of the TRILATERAL class named get_interior_angle_CAB() returns the
approximate angle measurement in degrees of the angle
 * formed by connecting points C, A, and B in the order specified by this sentence.
 *
 * The function below uses Law of Cosines to compute the measurement of an interior angle of
a triangle
 * using that triangle's three side lengths as function inputs to output some nonnegative real
number of degrees.
 */
double TRILATERAL::get_interior_angle_CAB()
{
    double a = 0.0, b = 0.0, c = 0.0, angle_opposite_of_a = 0.0, angle_opposite_of_b = 0.0,
angle_opposite_of_c = 0.0;
    a = B.get_distance_from(C); // a represents the length of the line segment whose endpoints
are B and C (and which are points of the caller TRIANGLE object of this function represents).
    b = C.get_distance_from(A); // b represents the length of the line segment whose endpoints
are C and A (and which are points of the caller TRIANGLE object of this function represents).
    c = A.get_distance_from(B); // c represents the length of the line segment whose endpoints
are A and B (and which are points of the caller TRIANGLE object of this function represents).
    angle_opposite_of_a = acos(((b * b) + (c * c) - (a * a)) / (2 * b * c)) * (180 / PI);
    angle_opposite_of_b = acos(((a * a) + (c * c) - (b * b)) / (2 * a * c)) * (180 / PI);
    angle_opposite_of_c = acos(((a * a) + (b * b) - (c * c)) / (2 * a * b)) * (180 / PI);
    return angle_opposite_of_a;
}

/**
 * If sum of the interior angle measurements of the quadrilateral which the caller TRILATERAL
object represents add up to approximately 180 degrees,
 * return true.
 * Otherwise, return false.
 */
bool TRILATERAL::interior_angles_add_up_to_180_degrees()
{
    return floor(get_interior_angle_ABC()) + floor(get_interior_angle_BCA()) +
floor(get_interior_angle_CAB());
```

```
}

/**
 * The default constructor of the TRILATERAL class calls the constructor of the POLYGON class and
 * sets the POINT type data member of the TRILATERAL object returned by this function named A to POINT(0,0),
 * sets the POINT type data member of the TRILATERAL object returned by this function named B to POINT(4,3), and
 * sets the POINT type data member of the TRILATERAL object returned by this function named C to POINT(4,0).
 */
TRILATERAL::TRILATERAL()
{
   std::cout << "\n\nCreating the TRILATERAL type object whose memory address is " << this << "...";
   A = POINT(0,0);
   B = POINT(4,3);
   C = POINT(4,0);
}

/**
 * The normal constructor of TRILATERAL attempts to set
 * the string type data member of this to the input string type value named color and
 * the POINT type data member of this named A to the input POINT type value named A and
 * the POINT type data member of this named B to the input POINT type value named B and
 * the POINT type data member of this named C to the input POINT type value named C.
 *
 * (The keyword this refers to the TRILATERAL object which is returned by this function).
 *
 * If A, B, and C represent unique points on a Cartesian plane,
 * if the interior angles of the trilatreal which those points would represent add up to 180 degrees, and
 * if the area of the trilateral which those points represents is larger than zero,
 * use the input POINT values as the POINT values for the TRILATERAL object which is returned by this function.
 */
TRILATERAL::TRILATERAL(std::string color, POINT A, POINT B, POINT C)
{
   std::cout << "\n\nCreating the TRILATERAL type object whose memory address is " << this << "...";
   TRILATERAL test_trilateral;
   test_trilateral.A.set_X(A.get_X());
   test_trilateral.A.set_Y(A.get_Y());
```

```
    test_trilateral.B.set_X(B.get_X());
    test_trilateral.B.set_Y(B.get_Y());
    test_trilateral.C.set_X(C.get_X());
    test_trilateral.C.set_Y(C.get_Y());
    if (test_trilateral.interior_angles_add_up_to_180_degrees() && (test_trilateral.get_area() > 0))
    {
        this -> A = A;
        this -> B = B;
        this -> C = C;
    }
    else
    {
        this -> A = POINT(0,0);
        this -> B = POINT(4,3);
        this -> C = POINT(4,0);
    }
    this -> color = color;
}

/**
 * The copy constructor method of the TRILATERAL class
 * instantiates TRILATERAL type objects
 * whose A value is set to the A value of the input TRILATERAL object,
 * whose B value is set to the B value of the input TRILATERAL object, and
 * whose C value is set to the C value of the input TRILATERAL object.
 */
TRILATERAL::TRILATERAL(TRILATERAL & trilateral)
{
    std::cout << "\n\nCreating the TRILATERAL type object whose memory address is " << this
<< "...";
    A = trilateral.A;
    B = trilateral.B;
    C = trilateral.C;
    color = trilateral.color;
}

/**
 * The TRILATERAL class implements the virtual get_area() method of the POLYGON class.
 *
 * The getter method returns the approximate area of the two-dimensional space whose bounds
are
 * the shortest paths between points A, B, and C of the triangle which the caller TRILATERAL
object represents
```

```
 * This function uses Heron's Formula to compute the area of a triangle using that triangle's side lengths as formula inputs.
 */
double TRILATERAL::get_area()
{
    double s = 0.0, a = 0.0, b = 0.0, c = 0.0;
    s = get_perimeter() / 2; // s is technically referred to as the semiperimter of the triangle which the caller TRIANGLE object of this function represents.
    a = B.get_distance_from(C); // a represents the length of the line segment whose endpoints are B and C (and which are points of the caller TRIANGLE object of this function represents).
    b = C.get_distance_from(A); // b represents the length of the line segment whose endpoints are C and A (and which are points of the caller TRIANGLE object of this function represents).
    c = A.get_distance_from(B); // c represents the length of the line segment whose endpoints are A and B (and which are points of the caller TRIANGLE object of this function represents).
    return sqrt(s * (s - a) * (s - b) * (s - c)); // Use Heron's Formula to compute the area of the triangle whose points are A, B, and C (and which are points of the caller TRIANGLE object of this function represents).
}

/**
 * The TRILATERAL class implements the virtual get_perimeter() method of the POLYGON class.
 *
 * The getter method returns the perimeter of the trilateral represented by the caller TRILATERAL object
 * by adding up the three side lengths of that trilateral.
 *
 * Let AB be the length of the line segment whose endpoints are A and B.
 * Let BC be the length of the line segment whose endpoints are B and C.
 * Let CA be the length of the line segment whose endpoints are C and A.
 *
 * Then return the sum of AB, BC, and CA.
 */
double TRILATERAL::get_perimeter()
{
    return A.get_distance_from(B) + B.get_distance_from(C) + C.get_distance_from(A);
}

/**
 * This method overrides the POLYGON class's print method.
 *
 * The descriptor method prints a description of the caller TRILATERAL instance to the output stream.
 *
```

```
 * If no function input is supplied, output is set to the command line terminal.
 */
void TRILATERAL::print(std::ostream & output)
{
    output << "\n\n----------------------------------------------------------------------------------------------------";
    output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the memory address of the first memory cell of a TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRILATERAL object.";
    output << "\n&category = " << &category << ". // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.";
    output << "\n&color = " << &color << ". // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..";
    output << "\n&A = " << &A << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.";
    output << "\n&B = " << &B << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.";
    output << "\n&C = " << &C << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.";
    output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
    output << "\nsizeof(int *) = " << sizeof(int *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
    output << "\nsizeof(int **) = " << sizeof(int **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
    output << "\nsizeof(std::string) = " << sizeof(std::string) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).";
    output << "\nsizeof(std::string *) = " << sizeof(std::string *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";
    output << "\nsizeof(std::string **) = " << sizeof(std::string **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";
    output << "\nsizeof(POINT) = " << sizeof(POINT) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).";
```

output << "\nsizeof(POINT *) = " << sizeof(POINT *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT **) = " << sizeof(POINT **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON) = " << sizeof(POLYGON) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON *) = " << sizeof(POLYGON *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON **) = " << sizeof(POLYGON **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(TRILATERAL) = " << sizeof(TRILATERAL) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(TRILATERAL *) = " << sizeof(TRILATERAL *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(TRILATERAL **) = " << sizeof(TRILATERAL **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\ncategory = " << category << ". // This is an immutable string type value which is a data member of the caller TRILATERAL object.";

output << "\ncolor = " << color << ". // This is a string type value which is a data member of the caller TRILATERAL object.";

output << "\nA = POINT(" << A.get_X() << "," << A.get_Y() << "). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nB = POINT(" << B.get_X() << "," << B.get_Y() << "). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nC = POINT(" << C.get_X() << "," << C.get_Y() << "). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\na = B.get_distance_from(C) = " << B.get_distance_from(C) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.";

output << "\nb = C.get_distance_from(A) = " << C.get_distance_from(A) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.";

output << "\nc = A.get_distance_from(B) = " << A.get_distance_from(B) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.";

output << "\nslope_of_side_a = B.get_slope_of_line_to(C) = " << B.get_slope_of_line_to(C) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\nslope_of_side_b = C.get_slope_of_line_to(A) = " << C.get_slope_of_line_to(A) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.";

output << "\nslope_of_side_c = A.get_slope_of_line_to(B) = " << A.get_slope_of_line_to(B) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.";

output << "\ninterior_angle_of_A = get_interior_angle_CAB() = " << get_interior_angle_CAB() << ". // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_of_B = get_interior_angle_ABC() = " << get_interior_angle_ABC() << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_of_C = get_interior_angle_BCA() = " << get_interior_angle_BCA() << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_of_A + interior_angle_of_B + interior_angle_of_C = " << get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() << ". // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)";

output << "\nget_perimeter() = a + b + c = " << get_perimeter() << ". // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.";

output << "\nget_area() = " << get_area() << ". // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the

```
two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.";
    output << "\n--------------------------------------------------------------------------------";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (i.e. <<).
 *
 * The friend function is not a member of the TRILATERAL class,
 * but the friend function does have access to the private and protected members of the TRILATERAL class as though
 * the friend function was a member of the TRILATERAL class.
 */
std::ostream & operator << (std::ostream & output, TRILATERAL & trilateral)
{
    trilateral.print(output);
    return output;
}

/**
 * The destructor method of the TRILATERAL class de-allocates memory which was used to
 * instantiate the TRILATERAL object which is calling this function.
 *
 * The destructor method of the TRILATERAL class is automatically called when
 * the program scope in which the caller TRILATERAL object was instantiated terminates.
 */
TRILATERAL::~TRILATERAL()
{
    std::cout << "\n\nDeleting the TRILATERAL type object whose memory address is " << this << "...";
}
```

RIGHT_TRILATERAL_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the RIGHT_TRILATREAL class.

C++_header_file: https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/right_trilateral.h

---

```
/**
 * file: right_trilateral.h
 * type: C++ (header file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

// If right_trilateral.h has not already been linked to a source file (.cpp), then link this header file
to the source file(s) which include this header file.
#ifndef RIGHT_TRILATERAL_H
#define RIGHT_TRILATERAL_H

// Include the C++ header file which contains preprocessing directives, variable declarations,
and function prototypes for the TRILATERAL class.
#include "trilateral.h"

/**
 * RIGHT_TRILATERAL is a class which inherits the protected and public data
 * attributes and methods of TRILATERAL.
 *
 * A RIGHT_TRILATERAL object represents an instance in which three unique POINT instances
exist
 * such that each one of those three POINT instances represents a unique coordinate pair within
the tuple of three POINT instances
 * (such that each coordinate pair represents exactly one two-dimensional point, POINT(X,Y), on
a Cartesian grid) and
 * such that exactly one interior angle of the triangle which that RIGHT_TRILATERAL object
represents is 90 degrees.
 *
 * Each RIGHT_TRILATERAL object represents a specific three-sided polygon whose area is a
positive real number.
 *
 * (A synonym for "trilateral" is "triangle").
 *
 * Class members which are set to the protected access specifier
 * are accessible to the base class and to derived classes.
 *
 * Class members which are set to the private access specifier
 * are only accessible to the base class.
 *
 * Class members which are set to the public access specifier
```

* are accessible to any scope within the program where
 * the base class and its derived classes are implemented.
 */
class RIGHT_TRILATERAL: public TRILATERAL
{
protected:

	/**
	* category is a description of the POLYGON instance.
	* category is set to a constant (i.e. immutable) string type value.
	*/
	const std::string category = "POLYGON/TRILATERAL/RIGHT_TRILATERAL";

public:

	/**
	* The default constructor of the RIGHT_TRILATERAL class calls the constructor of the TRILATERAL class and
	* sets the POINT type data member of the RIGHT_TRILATERAL object returned by this function named A to POINT(0,0),
	* sets the POINT type data member of the RIGHT_TRILATERAL object returned by this function named B to POINT(0,1), and
	* sets the POINT type data member of the RIGHT_TRILATERAL object returned by this function named C to POINT(1,0).
	*/
	RIGHT_TRILATERAL();

	/**
	* The normal constructor of RIGHT_TRILATERAL attempts to set
	* the string type data member of this to the input string type value named color and
	* the POINT type data member of this named A to the input POINT type value named A and
	* the POINT type data member of this named B to the input POINT type value named B and
	* the POINT type data member of this named C to the input POINT type value named C.
	*
	* (The keyword this refers to the RIGHT_TRILATERAL object which is returned by this function).
	*
	* If A, B, and C represent unique points on a Cartesian plane,
	* if the interior angles of the trilatreal which those points would represent add up to 180 degrees,
	* if the area of the trilateral which those points represents is larger than zero, and
	* if one of the interior angles which the trilateral those points represnts is 90 degrees,

* use the input POINT values as the POINT values for the RIGHT_TRILATERAL object which is returned by this function.
*/
RIGHT_TRILATERAL(std::string color, POINT A, POINT B, POINT C);

/**
* The copy constructor method of the RIGHT_TRILATERAL class
* instantiates RIGHT_TRILATERAL type objects
* whose A value is set to the A value of the input RIGHT_TRILATERAL object,
* whose B value is set to the B value of the input RIGHT_TRILATERAL object, and
* whose C value is set to the C value of the input RIGHT_TRILATERAL object.
*/
RIGHT_TRILATERAL(RIGHT_TRILATERAL & right_trilateral);

/**
* This method overrides the TRILATERAL class's print method.
*
* The descriptor method prints a description of the caller RIGHT_TRILATERAL instance to the output stream.
*
* If no function input is supplied, output is set to the command line terminal.
*/
void print(std::ostream & output = std::cout);

/**
* The friend function is an alternative to the print method.
* The friend function overloads the ostream operator (i.e. <<).
*
* The friend function is not a member of the RIGHT_TRILATERAL class,
* but the friend function does have access to the private and protected members of the RIGHT_TRILATERAL class as though
* the friend function was a member of the RIGHT_TRILATERAL class.
*/
friend std::ostream & operator << (std::ostream & output, RIGHT_TRILATERAL & right_trilateral);

/**
* The destructor method of the RIGHT_TRILATERAL class de-allocates memory which was used to
* instantiate the RIGHT_TRILATERAL object which is calling this function.
*
* The destructor method of the RIGHT_TRILATERAL class is automatically called when
* the program scope in which the caller RIGHT_TRILATERAL object was instantiated terminates.

```
*/
~RIGHT_TRILATERAL();
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

RIGHT_TRILATERAL_CLASS_SOURCE_CODE

The following source code defines the functions of the RIGHT_TRILATERAL class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/right_trilateral.cpp

---

```
/**
 * file: right_trilateral.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the RIGHT_TRILATERAL class.
#include "right_trilateral.h"

/**
 * The default constructor of the RIGHT_TRILATERAL class calls the constructor of the TRILATERAL class and
 * sets the POINT type data member of the RIGHT_TRILATERAL object returned by this function named A to POINT(0,0),
 * sets the POINT type data member of the RIGHT_TRILATERAL object returned by this function named B to POINT(0,1), and
 * sets the POINT type data member of the RIGHT_TRILATERAL object returned by this function named C to POINT(1,0).
 */
RIGHT_TRILATERAL::RIGHT_TRILATERAL()
{
```

```
	std::cout << "\n\nCreating the RIGHT_TRILATERAL type object whose memory address is " << this << "...";
	A = POINT(0,0);
	B = POINT(0,1);
	C = POINT(1,0);
}

/**
 * The normal constructor of RIGHT_TRILATERAL attempts to set
 * the string type data member of this to the input string type value named color and
 * the POINT type data member of this named A to the input POINT type value named A and
 * the POINT type data member of this named B to the input POINT type value named B and
 * the POINT type data member of this named C to the input POINT type value named C.
 *
 * (The keyword this refers to the RIGHT_TRILATERAL object which is returned by this function).
 *
 * If A, B, and C represent unique points on a Cartesian plane,
 * if the interior angles of the trilatreal which those points would represent add up to 180 degrees,
 * if the area of the trilateral which those points represents is larger than zero, and
 * if one of the interior angles which the trilateral those points represnts is 90 degrees,
 * use the input POINT values as the POINT values for the RIGHT_TRILATERAL object which is returned by this function.
 */
RIGHT_TRILATERAL::RIGHT_TRILATERAL(std::string color, POINT A, POINT B, POINT C)
{
	bool is_right_triangle = false;
	int test_interior_angle_A = 0.0, test_interior_angle_B = 0.0, test_interior_angle_C = 0.0;
	std::cout << "\n\nCreating the RIGHT_TRILATERAL type object whose memory address is " << this << "...";
	RIGHT_TRILATERAL test_right_trilateral;
	test_right_trilateral.A.set_X(A.get_X());
	test_right_trilateral.A.set_Y(A.get_Y());
	test_right_trilateral.B.set_X(B.get_X());
	test_right_trilateral.B.set_Y(B.get_Y());
	test_right_trilateral.C.set_X(C.get_X());
	test_right_trilateral.C.set_Y(C.get_Y());
	test_interior_angle_A = (int) floor(test_right_trilateral.get_interior_angle_CAB()); // coerce the data type to be int
	test_interior_angle_B = (int) floor(test_right_trilateral.get_interior_angle_ABC()); // coerce the data type to be int
	test_interior_angle_C = (int) floor(test_right_trilateral.get_interior_angle_BCA()); // coerce the data type to be int
```

```
	if ((test_interior_angle_A == 90) && (test_interior_angle_B < 90) &&
(test_interior_angle_C < 90)) is_right_triangle = true;
	if ((test_interior_angle_B == 90) && (test_interior_angle_A < 90) &&
(test_interior_angle_C < 90)) is_right_triangle = true;
	if ((test_interior_angle_C == 90) && (test_interior_angle_A < 90) &&
(test_interior_angle_B < 90)) is_right_triangle = true;
	if (test_right_trilateral.interior_angles_add_up_to_180_degrees() &&
(test_right_trilateral.get_area() > 0) && (is_right_triangle))
	{
	this -> A = A;
	this -> B = B;
	this -> C = C;
	}
	else
	{
	this -> A = POINT(0,0);
	this -> B = POINT(0,1);
	this -> C = POINT(1,0);
	}
	this -> color = color;
}

/**
 * The copy constructor method of the RIGHT_TRILATERAL class
 * instantiates RIGHT_TRILATERAL type objects
 * whose A value is set to the A value of the input RIGHT_TRILATERAL object,
 * whose B value is set to the B value of the input RIGHT_TRILATERAL object, and
 * whose C value is set to the C value of the input RIGHT_TRILATERAL object.
 */
RIGHT_TRILATERAL::RIGHT_TRILATERAL(RIGHT_TRILATERAL & right_trilateral)
{
	std::cout << "\n\nCreating the RIGHT_TRILATERAL type object whose memory address
is " << this << "...";
	A = right_trilateral.A;
	B = right_trilateral.B;
	C = right_trilateral.C;
	color = right_trilateral.color;
}

/**
 * This method overrides the TRILATERAL class's print method.
 *
 * The descriptor method prints a description of the caller RIGHT_TRILATERAL instance to the
output stream.
```

```
 *
 * If no function input is supplied, output is set to the command line terminal.
 */
void RIGHT_TRILATERAL::print(std::ostream & output)
{
	output << "\n\n----------------------------------------------------------------------------------------------------";
	output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the memory address of the first memory cell of a RIGHT_TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller RIGHT_TRILATERAL object.";
	output << "\n&category = " << &category << ". // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.";
	output << "\n&color = " << &color << ". // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..";
	output << "\n&A = " << &A << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.";
	output << "\n&B = " << &B << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.";
	output << "\n&C = " << &C << ". // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.";
	output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(int *) = " << sizeof(int *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(int **) = " << sizeof(int **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(std::string) = " << sizeof(std::string) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(std::string *) = " << sizeof(std::string *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";
	output << "\nsizeof(std::string **) = " << sizeof(std::string **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";
```

output << "\nsizeof(POINT) = " << sizeof(POINT) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT *) = " << sizeof(POINT *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POINT **) = " << sizeof(POINT **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON) = " << sizeof(POLYGON) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON *) = " << sizeof(POLYGON *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(POLYGON **) = " << sizeof(POLYGON **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(TRILATERAL) = " << sizeof(TRILATERAL) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(TRILATERAL *) = " << sizeof(TRILATERAL *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(TRILATERAL **) = " << sizeof(TRILATERAL **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(RIGHT_TRILATERAL) = " << sizeof(RIGHT_TRILATERAL) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RIGHT_TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(RIGHT_TRILATERAL *) = " << sizeof(RIGHT_TRILATERAL *) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(RIGHT_TRILATERAL **) = " << sizeof(RIGHT_TRILATERAL **) << ". // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\ncategory = " << category << ". // This is an immutable string type value which is a data member of the caller RIGHT_TRILATERAL object.";

output << "\ncolor = " << color << ". // This is a string type value which is a data member of the caller RIGHT_TRILATERAL object.";

output << "\nA = POINT(" << A.get_X() << "," << A.get_Y() << "). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nB = POINT(" << B.get_X() << "," << B.get_Y() << "). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\nC = POINT(" << C.get_X() << "," << C.get_Y() << "). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).";

output << "\na = B.get_distance_from(C) = " << B.get_distance_from(C) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.";

output << "\nb = C.get_distance_from(A) = " << C.get_distance_from(A) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.";

output << "\nc = A.get_distance_from(B) = " << A.get_distance_from(B) << ". // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.";

output << "\nslope_of_side_a = B.get_slope_of_line_to(C) = " << B.get_slope_of_line_to(C) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.";

output << "\nslope_of_side_b = C.get_slope_of_line_to(A) = " << C.get_slope_of_line_to(A) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.";

output << "\nslope_of_side_c = A.get_slope_of_line_to(B) = " << A.get_slope_of_line_to(B) << ". // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.";

output << "\ninterior_angle_of_A = get_interior_angle_CAB() = " << get_interior_angle_CAB() << ". // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).";

output << "\ninterior_angle_of_B = get_interior_angle_ABC() = " << get_interior_angle_ABC() << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line

```
segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).";
	output << "\ninterior_angle_of_C = get_interior_angle_BCA() = " << get_interior_angle_BCA() << ". // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).";
	output << "\ninterior_angle_of_A + interior_angle_of_B + interior_angle_of_C = " << get_interior_angle_CAB() + get_interior_angle_ABC() + get_interior_angle_BCA() << ". // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)";
	output << "\nget_perimeter() = a + b + c = " << get_perimeter() << ". // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.";
	output << "\nget_area() = " << get_area() << ". // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.";
	output << "\n--------------------------------------------------------------------------------";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (i.e. <<).
 *
 * The friend function is not a member of the RIGHT_TRILATERAL class,
 * but the friend function does have access to the private and protected members of the RIGHT_TRILATERAL class as though
 * the friend function was a member of the RIGHT_TRILATERAL class.
 */
std::ostream & operator << (std::ostream & output, RIGHT_TRILATERAL & right_trilateral)
{
	right_trilateral.print(output);
	return output;
}

/**
 * The destructor method of the RIGHT_TRILATERAL class de-allocates memory which was used to
 * instantiate the RIGHT_TRILATERAL object which is calling this function.
 *
 * The destructor method of the RIGHT_TRILATERAL class is automatically called when
```

```
 * the program scope in which the caller RIGHT_TRILATERAL object was instantiated
terminates.
 */
RIGHT_TRILATERAL::~RIGHT_TRILATERAL()
{
	std::cout << "\n\nDeleting the RIGHT_TRILATERAL type object whose memory address
is " << this << "...";
}
```

PROGRAM_SOURCE_CODE

The following source code defines the client which implements the POINT class, the POLYGON class, the QUADRILATERAL class, the TRAPEZOID class, the RECTANGLE class, the SQUARE class, the TRILATERAL class, and the RIGHT_TRILATERAL class. The client executes a series of unit tests which demonstrate how the methods of those classes work.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/polygon_class_inheritance_tester.cpp

---

```
/**
 * file: polygon_class_inheritance_tester.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

#include "polygon.h" // Include the C++ header file which contains preprocessing directives,
variable declarations, and function prototypes for the POLYGON class.
#include "quadrilateral.h" // Include the C++ header file which contains preprocessing directives,
variable declarations, and function prototypes for the QUADRILATERAL class.
#include "trapezoid.h" // Include the C++ header file which contains preprocessing directives,
variable declarations, and function prototypes for the TRAPEZOID class.
#include "rectangle.h" // Include the C++ header file which contains preprocessing directives,
variable declarations, and function prototypes for the RECTANGLE class.
#include "square.h" // Include the C++ header file which contains preprocessing directives,
variable declarations, and function prototypes for the SQUARE class.
#include "trilateral.h" // Include the C++ header file which contains preprocessing directives,
variable declarations, and function prototypes for the TRILATERAL class.
```

```
#include "right_trilateral.h" // Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the RIGHT_TRILATERAL class.

/* function prototypes */
void unit_test_0(std::ostream & output);
void unit_test_1(std::ostream & output);
void unit_test_2(std::ostream & output);
void unit_test_3(std::ostream & output);
void unit_test_4(std::ostream & output);
void unit_test_5(std::ostream & output);
void unit_test_6(std::ostream & output);
void unit_test_7(std::ostream & output);
void unit_test_8(std::ostream & output);
void unit_test_9(std::ostream & output);
void unit_test_10(std::ostream & output);
void unit_test_11(std::ostream & output);
void unit_test_12(std::ostream & output);
void unit_test_13(std::ostream & output);
void unit_test_14(std::ostream & output);
void unit_test_15(std::ostream & output);
void unit_test_16(std::ostream & output);
void unit_test_17(std::ostream & output);
void unit_test_18(std::ostream & output);
void unit_test_19(std::ostream & output);
void unit_test_20(std::ostream & output);
void unit_test_21(std::ostream & output);
void unit_test_22(std::ostream & output);
void unit_test_23(std::ostream & output);
void unit_test_24(std::ostream & output);
void unit_test_25(std::ostream & output);
void unit_test_26(std::ostream & output);
void unit_test_27(std::ostream & output);
void unit_test_28(std::ostream & output);
void unit_test_29(std::ostream & output);

/**
 * Unit Test # 0: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated QUADRILATERAL instance.
 * Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class and the getter methods of the POLYGON class.
 */
void unit_test_0(std::ostream & output)
{
```

```
	output <<
"\n\n--------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 0: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated QUADRILATERAL instance. Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class and the getter methods of the POLYGON class.";
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\n// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.";
	output << "\nPOLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as QUADRILATERAL.";
	output << "\npointer_to_polygon = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";
	output << "\npointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.";
	// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.
	POLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as QUADRILATERAL.
	pointer_to_polygon = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
	pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.
	output << "\npointer_to_polygon -> get_area() = " << pointer_to_polygon -> get_area() << ". // Indirectly call the POLYGON get_area() method.";
	output << "\npointer_to_polygon -> get_perimeter() = " << pointer_to_polygon -> get_perimeter() << ". // Indirectly call the POLYGON get_permieter() method.";
	output << "\ndelete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated QUADRILATERAL instance.";
	output << "\n--------------------------------------------------------------------------------------------------";
	delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated QUADRILATERAL instance.
}

/**
 * Unit Test # 1: Test the default QUADRILATERAL constructor and the QUADRILATERAL print method.
 */
void unit_test_1(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------------------------";
```

```
	output << "\nUnit Test # 1: Test the default QUADRILATERAL constructor and the QUADRILATERAL print method.";
	output << "\n--------------------------------------------------------------------------------------------------------";
	output << "\nQUADRILATERAL quadrilateral;";
	output << "\nquadrilateral.print(); // Test the default argument (which is std::cout).";
	output << "\nquadrilateral.print(output);";
	output << "\noutput << quadrilateral; // overloaded ostream operator as defined in quadrilateral.cpp";
	output << "\n--------------------------------------------------------------------------------------------------------";
	QUADRILATERAL quadrilateral;
	quadrilateral.print(); // Test the default argument (which is std::cout).
	quadrilateral.print(output);
	output << quadrilateral; // overloaded ostream operator as defined in quadrilateral.cpp
}

/**
 * Unit Test # 2: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated QUADRILATERAL instance.
 * Use that pointer-to-POLYGON to call the overloaded ostream operator method of the POLYGON class (and not of the QUADRILATERAL class).
 */
void unit_test_2(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 2: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated QUADRILATERAL instance. Use that pointer-to-POLYGON to call the overloaded ostream operator method of the POLYGON class (and not of the QUADRILATERAL class).";
	output << "\n--------------------------------------------------------------------------------------------------------";
	output << "\n// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.";
	output << "\nPOLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as QUADRILATERAL.";
	output << "\npointer_to_polygon = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";
	output << "\noutput << * pointer_to_polygon; // Use the overloaded ostream operator as defined in polygon.cpp to print the data which is stored at the memory address which pointer_to_polygon stores.";
	output << "\ndelete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated QUADRILATERAL instance.";
	output << "\n--------------------------------------------------------------------------------------------------------";
```

```
	// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.";
	POLYGON * pointer_to_polygon; // The pointer-to-polygon type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as QUADRILATERAL.
	pointer_to_polygon = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
	output << * pointer_to_polygon; // Use the overloaded ostream operator as defined in polygon.cpp to print the data which is stored at the memory address which pointer_to_polygon stores.
	delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated QUADRILATERAL instance.
}

/**
 * Unit Test # 3: Unit Test # 3: Create a pointer-to-QUADRILATERAL type variable to store the memory address of a dynamically allocated QUADRILATERAL instance.
 * Use that pointer-to-QUADRILATERAL to call the overloaded ostream operator method of the QUADRILATERAL class and the public getter methods of the QUADRILATERAL class.
 */
void unit_test_3(std::ostream & output)
{
	output << "\n----------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 3: Create a pointer-to-QUADRILATERAL type variable to store the memory address of a dynamically allocated QUADRILATERAL instance. Use that pointer-to-QUADRILATERAL to call the overloaded ostream operator method of the QUADRILATERAL class and the public getter methods of the QUADRILATERAL class.";
	output << "\n----------------------------------------------------------------------------------------------------";
	output << "\nQUADRILATERAL * pointer_to_quadrilateral; // The pointer-to-QUADRILATERAL type variable can store the memory address of an object whose data type is QUADRILATERAL or else a non-abstract derived class of QUADRILATERAL such as TRAPEZOID.";
	output << "\npointer_to_quadrilateral = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";
	output << "\noutput << * pointer_to_quadrilateral; // Use the overloaded ostream operator as defined in quadrilateral.cpp to print the data which is stored at the memory address which pointer_to_quadrilateral stores.";
	output << "\ndelete pointer_to_quadrilateral; // De-allocate memory which was assigned to the dynamically allocated QUADRILATERAL instance.";
	output << "\n----------------------------------------------------------------------------------------------------";
```

```
	QUADRILATERAL * pointer_to_quadrilateral; // The pointer-to-QUADRATERAL type variable can store the memory address of an object whose data type is QUADRILATERAL or else a non-abstract derived class of QUADRILATERAL such as TRAPEZOID.
	pointer_to_quadrilateral = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
	output << * pointer_to_quadrilateral; // Use the overloaded ostream operator as defined in quadrilateral.cpp to print the data which is stored at the memory address which pointer_to_quadrilateral stores.
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\npointer_to_quadrilateral -> get_area() = " << pointer_to_quadrilateral -> get_area() << ". // Indirectly call the get_area() method of the QUADRILATERAL class.";
	output << "\npointer_to_quadrilateral -> get_perimeter() = " << pointer_to_quadrilateral -> get_perimeter() << ". // Indirectly call the get_perimeter() method of the QUADRILATERAL class.";
	delete pointer_to_quadrilateral; // De-allocate memory which was assigned to the dynamically allocated QUADRILATERAL instance.
}

/**
 * Unit Test # 4: Test the normal QUADRILATERAL constructor and QUADRILATERAL copy constructor using valid function inputs and the QUADRILATERAL print method.
 */
void unit_test_4(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 4: Test the normal QUADRILATERAL constructor and QUADRILATERAL copy constructor using valid function inputs and the QUADRILATERAL print method.";
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\nQUADRILATERAL quadrilateral_0 = QUADRILATERAL(\"green\", POINT(-2,-2), POINT(-2,2), POINT(2,2), POINT(2,-2));";
	output << "\nquadrilateral_0.print(output);";
	output << "\nQUADRILATERAL quadrilateral_1 = QUADRILATERAL(\"blue\", POINT(0,0), POINT(3,2), POINT(5,1), POINT(-1,-2));";
	output << "\nquadrilateral_1.print(output);";
	output << "\nQUADRILATERAL quadrilateral_2 = QUADRILATERAL(quadrilateral_0);";
	output << "\nquadrilateral_2.print(output);";
	output << "\n--------------------------------------------------------------------------------------------------";
	QUADRILATERAL quadrilateral_0 = QUADRILATERAL("green", POINT(-2,-2), POINT(-2,2), POINT(2,2), POINT(2,-2));
	quadrilateral_0.print(output);
	QUADRILATERAL quadrilateral_1 = QUADRILATERAL("blue", POINT(0,0), POINT(3,2), POINT(5,1), POINT(-1,-2));
```

```
	quadrilateral_1.print(output);
	QUADRILATERAL quadrilateral_2 = QUADRILATERAL(quadrilateral_0);
	quadrilateral_2.print(output);
}

/**
 * Unit Test # 5: Test the normal QUADRILATERAL constructor using invalid function inputs and
the QUADRILATERAL print method.
 */
void unit_test_5(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 5: Test the normal QUADRILATERAL constructor using invalid
function inputs and the QUADRILATERAL print method.";
	output << "\n--------------------------------------------------------------------------------------------";
	output << "\nQUADRILATERAL quadrilateral_0 = QUADRILATERAL(\"red\",
POINT(-2,-2), POINT(0,0), POINT(1,1), POINT(2,2)); // A line intersects all four points.";
	output << "\nquadrilateral_0.print(output);";
	output << "\nQUADRILATERAL quadrilateral_1 = QUADRILATERAL(\"purple\",
POINT(0,0), POINT(3,2), POINT(0,0), POINT(-1,-2)); // Not all point coordinate pairs are
unique.";
	output << "\nquadrilateral_1.print(output);";
	output << "\nQUADRILATERAL quadrilateral_2 = QUADRILATERAL(\"yellow\",
POINT(0,0), POINT(0,2), POINT(4,0), POINT(4,2)); // The points form a bow-tie shaped
polygon.";
	output << "\nquadrilateral_2.print(output);";
	output << "\n--------------------------------------------------------------------------------------------";
	QUADRILATERAL quadrilateral_0 = QUADRILATERAL("red", POINT(-2,-2),
POINT(0,0), POINT(1,1), POINT(2,2)); // A line intersects all four points.
	quadrilateral_0.print(output);
	QUADRILATERAL quadrilateral_1 = QUADRILATERAL("purple", POINT(0,0),
POINT(3,2), POINT(0,0), POINT(-1,-2)); // Not all point coordinate pairs are unique.
	quadrilateral_1.print(output);
	QUADRILATERAL quadrilateral_2 = QUADRILATERAL("yellow", POINT(0,0),
POINT(0,2), POINT(4,0), POINT(4,2)); // The points form a bow-tie shaped polygon.
	quadrilateral_2.print(output);
}

/**
 * Unit Test # 6: Test the default TRAPEZOID constructor and the TRAPEZOID print method.
 */
void unit_test_6(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------------------";
```

```
	output << "\nUnit Test # 6: Test the default TRAPEZOID constructor and the TRAPEZOID print method.";
	output << "\n--------------------------------------------------------------------------------";
	output << "\nTRAPEZOID trapezoid;";
	output << "\ntrapezoid.print(); // Test the default argument (which is std::cout).";
	output << "\ntrapezoid.print(output);";
	output << "\noutput << trapezoid; // overloaded ostream operator as defined in trapezoid.cpp";
	output << "\n--------------------------------------------------------------------------------";
	TRAPEZOID trapezoid;
	trapezoid.print(); // Test the default argument (which is std::cout).
	trapezoid.print(output);
	output << trapezoid; // overloaded ostream operator as defined in trapezoid.cpp
}

/**
 * Unit Test # 7: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated TRAPEZOID instance.
 * Use that pointer-to-POLYGON type variable to call the POLYGON print method and the POLYGON getter methods.
 */
void unit_test_7(std::ostream & output)
{
	output << "\n\n--------------------------------------------------------------------------------";
	output << "\nUnit Test # 7: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated TRAPEZOID instance. Use that pointer-to-POLYGON type variable to call the POLYGON print method and the POLYGON getter methods.";
	output << "\n--------------------------------------------------------------------------------";
	output << "\n// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.";
	output << "\nPOLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as TRAPEZOID.";
	output << "\npointer_to_polygon = new TRAPEZOID; // Assign memory to a dynamic TRAPEZOID instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";
	output << "\npointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.";
	// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.
```

```
	POLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the
memory address of an object whose data type is a non-abstract derived class of POLYGON
such as TRAPEZOID.
	pointer_to_polygon = new TRAPEZOID; // Assign memory to a dynamic TRAPEZOID
instance (i.e. and dynamic implies that the variable was created during program runtime instead
of program compile time).
	pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.
	output << "\npointer_to_polygon -> get_area() = " << pointer_to_polygon -> get_area()
<< ". // Indirectly call the POLYGON get_area() method.";
	output << "\npointer_to_polygon -> get_perimeter() = " << pointer_to_polygon ->
get_perimeter() << ". // Indirectly call the POLYGON get_permieter() method.";
	output << "\ndelete pointer_to_polygon; // De-allocate memory which was assigned to
the dynamically allocated TRAPEZOID instance.";
	output << "\n--------------------------------------------------------------------------------";
	delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically
allocated TRAPEZOID instance.
}

/**
 * Unit Test # 8: Create a pointer-to-QUADRILATERAL type variable to store the memory
address of a dynamically allocated TRAPEZOID instance.
 * Use that pointer-to-QUADRILATERAL type variable to call the QUADRILATERAL print
method.
 */
void unit_test_8(std::ostream & output)
{
	output <<
"\n\n--------------------------------------------------------------------------------";
	output << "\nUnit Test # 8: Create a pointer-to-QUADRILATERAL type variable to store
the memory address of a dynamically allocated TRAPEZOID instance. Use that
pointer-to-QUADRILATERAL type variable to call the QUADRILATERAL print method.";
	output << "\n--------------------------------------------------------------------------------";
	output << "\nQUADRILATERAL * pointer_to_quadrilateral; // The
pointer-to-QUADRILATERAL type variable can store the memory address of an object whose
data type is a non-abstract derived class of QUADRILATERAL such as TRAPEZOID.";
	output << "\npointer_to_quadrilateral = new TRAPEZOID; // Assign memory to a
dynamic TRAPEZOID instance (i.e. and dynamic implies that the variable was created during
program runtime instead of program compile time).";
	output << "\npointer_to_quadrilateral -> print(output); // Indirectly call the
QUADRILATERAL print method.";
	output << "\ndelete pointer_to_quadrilateral; // De-allocate memory which was assigned
to the dynamically allocated TRAPEZOID instance.";
	output << "\n--------------------------------------------------------------------------------";
```

```
	QUADRILATERAL * pointer_to_quadrilateral; // The pointer-to-QUADRILATERAL type variable can store the memory address of an object whose data type is a non-abstract derived class of QUADRILATERAL such as TRAPEZOID.
	pointer_to_quadrilateral = new TRAPEZOID; // Assign memory to a dynamic TRAPEZOID instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
	pointer_to_quadrilateral -> print(output); // Indirectly call the QUADRILATERAL print method.
	delete pointer_to_quadrilateral; // De-allocate memory which was assigned to the dynamically allocated TRAPEZOID instance.
}

/**
 * Unit Test # 9: Test the normal TRAPEZOID constructor and TRAPEZOID copy constructor using valid function inputs and the TRAPEZOID print method.
 */
void unit_test_9(std::ostream & output)
{
	output << "\n----------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 9: Test the normal TRAPEZOID constructor and TRAPEZOID copy constructor using valid function inputs and the TRAPEZOID print method.";
	output << "\n----------------------------------------------------------------------------------------------------";
	output << "\nTRAPEZOID trapezoid_0 = TRAPEZOID(\"pink\", POINT(-5,-10), POINT(0,5), POINT(15,5), POINT(30,-10));";
	output << "\ntrapezoid_0.print(output);";
	output << "\nTRAPEZOID trapezoid_1 = TRAPEZOID(\"brown\", POINT(0,0), POINT(4,-5), POINT(15,-5), POINT(45,0));";
	output << "\ntrapezoid_1.print(output);";
	output << "\nTRAPEZOID trapezoid_2 = TRAPEZOID(trapezoid_0);";
	output << "\ntrapezoid_2.print(output);";
	output << "\n----------------------------------------------------------------------------------------------------";
	TRAPEZOID trapezoid_0 = TRAPEZOID("pink", POINT(-5,-10), POINT(0,5), POINT(15,5), POINT(30,-10));
	trapezoid_0.print(output);
	TRAPEZOID trapezoid_1 = TRAPEZOID("brown", POINT(0,0), POINT(4,-5), POINT(15,-5), POINT(45,0));
	trapezoid_1.print(output);
	TRAPEZOID trapezoid_2 = TRAPEZOID(trapezoid_0);
	trapezoid_2.print(output);
}

/**
 * Unit Test # 10: Test the default RECTANGLE constructor and the RECTANGLE print method.
 */
```

```
void unit_test_10(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------";
	output << "\nUnit Test # 10: Test the default RECTANGLE constructor and the
RECTANGLE print method.";
	output << "\n--------------------------------------------------------------------------------";
	output << "\nRECTANGLE rectangle;";
	output << "\nrectangle.print(); // Test the default argument (which is std::cout).";
	output << "\nrectangle.print(output);";
	output << "\noutput << rectangle; // overloaded ostream operator as defined in
rectangle.cpp";
	output << "\n--------------------------------------------------------------------------------";
	RECTANGLE rectangle;
	rectangle.print(); // Test the default argument (which is std::cout).
	rectangle.print(output);
	output << rectangle; // overloaded ostream operator as defined in rectangle.cpp
}

/**
 * Unit Test # 11: Create a pointer-to-POLYGON type variable to store the memory address of a
dynamically allocated RECTANGLE instance.
 * Use that pointer-to-POLYGON type variable to call the POLYGON print method and the
POLYGON getter methods.
 */
void unit_test_11(std::ostream & output)
{
	output <<
"\n\n--------------------------------------------------------------------------------";
	output << "\nUnit Test # 11: Create a pointer-to-POLYGON type variable to store the
memory address of a dynamically allocated RECTANGLE instance. Use that
pointer-to-POLYGON type variable to call the POLYGON print method and the POLYGON getter
methods.";
	output << "\n--------------------------------------------------------------------------------";
	output << "\n// COMMENTED OUT: POLYGON polygon; // This command does not work
because POLYGON is an abstract class.";
	output << "\nPOLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable
can store the memory address of an object whose data type is a non-abstract derived class of
POLYGON such as RECTANGLE.";
	output << "\npointer_to_polygon = new RECTANGLE; // Assign memory to a dynamic
RECTANGLE instance (i.e. and dynamic implies that the variable was created during program
runtime instead of program compile time).";
	output << "\npointer_to_polygon -> print(output); // Indirectly call the POLYGON print
method.";
```

// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.

POLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as RECTANGLE.

pointer_to_polygon = new RECTANGLE; // Assign memory to a dynamic RECTANGLE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).

pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.

output << "\n// COMMENTED OUT (does not work): pointer_to_polygon -> quadrilateral_test(). // Indirectly call the QUADRILATERAL quadrilateral_test() method.";

output << "\npointer_to_polygon -> get_area() = " << pointer_to_polygon -> get_area() << ". // Indirectly call the POLYGON get_area() method.";

output << "\npointer_to_polygon -> get_perimeter() = " << pointer_to_polygon -> get_perimeter() << ". // Indirectly call the POLYGON get_permieter() method.";

output << "\ndelete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated RECTANGLE instance.";

output << "\n--";

delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated TRAPEZOID instance.

}

/**

* Unit Test # 12: Create a pointer-to-QUADRILATERAL type variable to store the memory address of a dynamically allocated RECTANGLE instance.

* Use that pointer-to-QUADRILATERAL type variable to call the QUADRILATERAL print method and the QUADRILATERAL getter methods.

*/

void unit_test_12(std::ostream & output)

{

output << "\n\n--";

output << "\nUnit Test # 12: Create a pointer-to-QUADRILATERAL type variable to store the memory address of a dynamically allocated RECTANGLE instance. Use that pointer-to-QUADRILATERAL type variable to call the POLYGON print method and the QUADRILATERAL getter methods.";

output << "\n--";

output << "\nQUADRILATERAL * pointer_to_quadrilateral; // The pointer-to-QUADRILATERAL type variable can store the memory address of an object whose data type is a non-abstract derived class of QUADRILATERAL such as RECTANGLE.";

output << "\npointer_to_quadrilateral = new RECTANGLE; // Assign memory to a dynamic RECTANGLE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";

```
	output << "\npointer_to_quadrilateral -> print(output); // Indirectly call the QUADRILATERAL print method.";
	QUADRILATERAL * pointer_to_quadrilateral; // The pointer-to-QUADRILATERAL type variable can store the memory address of an object whose data type is a non-abstract derived class of QUADRILATERAL such as RECTANGLE.
	pointer_to_quadrilateral = new RECTANGLE; // Assign memory to a dynamic RECTANGLE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
	pointer_to_quadrilateral -> print(output); // Indirectly call the POLYGON print method.
	output << "\n// COMMENTED OUT (does not work): pointer_to_quadrilateral -> rectangle_test(); // Indirectly call the RECTANGLE rectangle_test() method.";
	output << "\npointer_to_quadrilateral -> get_area() = " << pointer_to_quadrilateral -> get_area() << ". // Indirectly call the QUADRILATERAL get_area() method.";
	output << "\npointer_to_quadrilateral -> get_perimeter() = " << pointer_to_quadrilateral -> get_perimeter() << ". // Indirectly call the QUADRILATERAL get_permieter() method.";
	output << "\ndelete pointer_to_quadrilateral; // De-allocate memory which was assigned to the dynamically allocated RECTANGLE instance.";
	output << "\n--------------------------------------------------------------------------------------------------";
	delete pointer_to_quadrilateral; // De-allocate memory which was assigned to the dynamically allocated RECTANGLE instance.
}

/**
 * Unit Test # 13: Test the normal RECTANGLE constructor and RECTANGLE copy constructor using valid function inputs and the RECTANGLE print method.
 */
void unit_test_13(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 13: Test the normal RECTANGLE constructor and RECTANGLE copy constructor using valid function inputs and the RECTANGLE print method.";
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\nRECTANGLE rectangle_0 = RECTANGLE(\"gray\", POINT(9,10), POINT(9,5), POINT(3,5), POINT(3,10));";
	output << "\nrectangle_0.print(output);";
	output << "\nRECTANGLE rectangle_1 = RECTANGLE(\"black\", POINT(0,0), POINT(0,1), POINT(1,1), POINT(1,0));";
	output << "\nrectangle_1.print(output);";
	output << "\nRECTANGLE rectangle_2 = RECTANGLE(rectangle_0);";
	output << "\nrectangle_2.print(output);";
	output << "\n--------------------------------------------------------------------------------------------------";
	RECTANGLE rectangle_0 = RECTANGLE("gray", POINT(9,10), POINT(9,5), POINT(3,5), POINT(3,10));
	rectangle_0.print(output);
```

```
	RECTANGLE rectangle_1 = RECTANGLE("black", POINT(0,0), POINT(0,1),
POINT(1,1), POINT(1,0));
	rectangle_1.print(output);
	RECTANGLE rectangle_2 = RECTANGLE(rectangle_0);
	rectangle_2.print(output);
}

/**
 * Unit Test # 14: Test the normal RECTANGLE constructor using invalid function inputs and the
RECTANGLE print method.
 */
void unit_test_14(std::ostream & output)
{
	output << "\n----------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 14: Test the normal RECTANGLE constructor using invalid
function inputs and the RECTANGLE print method.";
	output << "\n----------------------------------------------------------------------------------------------------";
	output << "\nRECTANGLE rectangle_0 = RECTANGLE(\"red\", POINT(-1,-1),
POINT(0,0), POINT(1,1), POINT(2,2));";
	output << "\nrectangle_0.print(output);";
	output << "\nRECTANGLE rectangle_1 = RECTANGLE(\"green\", POINT(-5,-10),
POINT(0,5), POINT(15,5), POINT(30,-10));";
	output << "\nrectangle_1.print(output);";
	output << "\nRECTANGLE rectangle_2 = RECTANGLE(\"blue\", POINT(-5,-5),
POINT(0,0), POINT(-5,-5), POINT(0,0));";
	output << "\nrectangle_2.print(output);";
	output << "\n----------------------------------------------------------------------------------------------------";
	RECTANGLE rectangle_0 = RECTANGLE("red", POINT(-1,-1), POINT(0,0), POINT(1,1),
POINT(2,2));
	rectangle_0.print(output);
	RECTANGLE rectangle_1 = RECTANGLE("green", POINT(-5,-10), POINT(0,5),
POINT(15,5), POINT(30,-10));
	rectangle_1.print(output);
	RECTANGLE rectangle_2 = RECTANGLE("blue", POINT(-5,-5), POINT(0,0),
POINT(-5,-5), POINT(0,0));
	rectangle_2.print(output);
}

/**
 * Unit Test # 15: Create a pointer-to-POLYGON type variable to store the memory address of a
dynamically allocated SQUARE instance.
 * Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class
and the getter methods of the POLYGON class.
 */
```

```
void unit_test_15(std::ostream & output)
{
	output << 
"\n\n--------------------------------------------------------------------------------";
	output << "\nUnit Test # 15: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated SQUARE instance. Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class and the getter methods of the POLYGON class.";
	output << "\n--------------------------------------------------------------------------------";
	output << "\n// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.";
	output << "\nPOLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as SQUARE.";
	output << "\npointer_to_polygon = new SQUARE; // Assign memory to a dynamic SQUARE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";
	output << "\npointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.";
	// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.
	POLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as SQUARE.
	pointer_to_polygon = new SQUARE; // Assign memory to a dynamic SQUARE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
	pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.
	output << "\npointer_to_polygon -> get_area() = " << pointer_to_polygon -> get_area() << ". // Indirectly call the POLYGON get_area() method.";
	output << "\npointer_to_polygon -> get_perimeter() = " << pointer_to_polygon -> get_perimeter() << ". // Indirectly call the POLYGON get_permieter() method.";
	output << "\ndelete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated SQUARE instance.";
	output << "\n--------------------------------------------------------------------------------";
	delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated SQUARE instance.
}

/**
 * Unit Test # 16: Test the default SQUARE constructor and the SQUARE print method.
 */
void unit_test_16(std::ostream & output)
{
```

```
	output << "\n---------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 16: Test the default SQUARE constructor and the SQUARE print method.";
	output << "\n---------------------------------------------------------------------------------------------------";
	output << "\nSQUARE square;";
	output << "\nsquare.print(); // Test the default argument (which is std::cout).";
	output << "\nsquare.print(output);";
	output << "\noutput << square; // overloaded ostream operator as defined in square.cpp";
	output << "\n---------------------------------------------------------------------------------------------------";
	SQUARE square;
	square.print(); // Test the default argument (which is std::cout).
	square.print(output);
	output << square; // overloaded ostream operator as defined in square.cpp
}

/**
 * Unit Test # 17: Create a pointer-to-QUADRILATERAL type variable to store the memory address of a dynamically allocated SQUARE instance.
 * Use that pointer-to-QUADRILATERAL type variable to call the QUADRILATERAL print method and the QUADRILATERAL getter methods.
 */
void unit_test_17(std::ostream & output)
{
	output << "\n\n---------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 17: Create a pointer-to-QUADRILATERAL type variable to store the memory address of a dynamically allocated SQUARE instance. Use that pointer-to-QUADRILATERAL type variable to call the QUADRILATERAL print method and the QUADRILATERAL getter methods.";
	output << "\n---------------------------------------------------------------------------------------------------";
	output << "\nQUADRILATERAL * pointer_to_quadrilateral; // The pointer-to-QUADRILATERAL type variable can store the memory address of an object whose data type is a non-abstract derived class of QUADRILATERAL such as SQUARE.";
	output << "\npointer_to_quadrilateral = new SQUARE; // Assign memory to a dynamic SQUARE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";
	output << "\npointer_to_quadrilateral -> print(output); // Indirectly call the QUADRILATERAL print method.";
	QUADRILATERAL * pointer_to_quadrilateral; // The pointer-to-QUADRILATERAL type variable can store the memory address of an object whose data type is a non-abstract derived class of QUADRILATERAL such as SQUARE.
	pointer_to_quadrilateral = new SQUARE; // Assign memory to a dynamic SQUARE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
```

```
	pointer_to_quadrilateral -> print(output); // Indirectly call the POLYGON print method.
	output << "\npointer_to_quadrilateral -> get_area() = " << pointer_to_quadrilateral -> get_area() << ". // Indirectly call the QUADRILATERAL get_area() method.";
	output << "\npointer_to_quadrilateral -> get_perimeter() = " << pointer_to_quadrilateral -> get_perimeter() << ". // Indirectly call the QUADRILATERAL get_permieter() method.";
	output << "\ndelete pointer_to_quadrilateral; // De-allocate memory which was assigned to the dynamically allocated SQUARE instance.";
	output << "\n--------------------------------------------------------------------------------";
	delete pointer_to_quadrilateral; // De-allocate memory which was assigned to the dynamically allocated SQUARE instance.
}

/**
 * Unit Test # 18: Create a pointer-to-RECTANGLE type variable to store the memory address of a dynamically allocated SQUARE instance.
 * Use that pointer-to-RECTANGLE type variable to call the RECTANGLE print method and the RECTANGLE getter methods.
 */
void unit_test_18(std::ostream & output)
{
	output << "\n\n--------------------------------------------------------------------------------";
	output << "\nUnit Test # 18: Create a pointer-to-RECTANGLE type variable to store the memory address of a dynamically allocated SQUARE instance. Use that pointer-to-RECTANGLE type variable to call the RECTANGLE print method and the RECTANGLE getter methods.";
	output << "\n--------------------------------------------------------------------------------";
	output << "\nRECTANGLE * pointer_to_rectangle; // The pointer-to-RECTANGLE type variable can store the memory address of an object whose data type is a non-abstract derived class of RECTANGLE such as SQUARE.";
	output << "\npointer_to_rectangle = new SQUARE; // Assign memory to a dynamic SQUARE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";
	output << "\npointer_to_rectangle -> print(output); // Indirectly call the RECTANGLE print method.";
	RECTANGLE * pointer_to_rectangle; // The pointer-to-RECTANGLE type variable can store the memory address of an object whose data type is a non-abstract derived class of RECTANGLE such as SQUARE.
	pointer_to_rectangle = new SQUARE; // Assign memory to a dynamic SQUARE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
	pointer_to_rectangle -> print(output); // Indirectly call the RECTANGLE print method.
	output << "\npointer_to_rectangle -> get_area() = " << pointer_to_rectangle -> get_area() << ". // Indirectly call the RECTANGLE get_area() method.";
```

```
	output << "\npointer_to_rectangle -> get_perimeter() = " << pointer_to_rectangle -> get_perimeter() << ". // Indirectly call the RECTANGLE get_permieter() method.";
	output << "\ndelete pointer_to_rectangle; // De-allocate memory which was assigned to the dynamically allocated SQUARE instance.";
	output << "\n--------------------------------------------------------------------------------------------------";
	delete pointer_to_rectangle; // De-allocate memory which was assigned to the dynamically allocated SQUARE instance.
}

/**
 * Unit Test # 19: Test the normal SQUARE constructor and SQUARE copy constructor using valid function inputs and the SQUARE print method.
 */
void unit_test_19(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 19: Test the normal SQUARE constructor and SQUARE copy constructor using valid function inputs and the SQUARE print method.";
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\nSQUARE square_0 = SQUARE(\"yellow\", POINT(-3,-3), POINT(-3,0), POINT(0,0), POINT(0,-3));";
	output << "\nsquare_0.print(output);";
	output << "\nSQUARE square_1 = SQUARE(\"white\", POINT(-1,-1), POINT(-1,1), POINT(1,1), POINT(1,-1));";
	output << "\nsquare_1.print(output);";
	output << "\nSQUARE square_2 = SQUARE(square_0);";
	output << "\nsquare_2.print(output);";
	output << "\n--------------------------------------------------------------------------------------------------";
	SQUARE square_0 = SQUARE("yellow", POINT(-3,-3), POINT(-3,0), POINT(0,0), POINT(0,-3));
	square_0.print(output);
	SQUARE square_1 = SQUARE("white", POINT(-1,-1), POINT(-1,1), POINT(1,1), POINT(1,-1));
	square_1.print(output);
	SQUARE square_2 = SQUARE(square_0);
	square_2.print(output);
}

/**
 * Unit Test # 20: Test the normal SQUARE constructor using invalid function inputs and the SQUARE print method.
 */
void unit_test_20(std::ostream & output)
{
```

```
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 20: Test the normal SQUARE constructor using invalid function inputs and the SQUARE print method.";
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\nSQUARE square_0 = SQUARE(\"red\", POINT(0,0), POINT(0,1), POINT(0,2), POINT(0,3));";
	output << "\nsquare_0.print(output);";
	output << "\nSQUARE square_1 = SQUARE(\"green\", POINT(0,0), POINT(0,1), POINT(5,1), POINT(5,0));";
	output << "\nsquare_1.print(output);";
	output << "\nSQUARE square_2 = SQUARE(\"blue\", POINT(0,0), POINT(0,1), POINT(1,1), POINT(0,0));";
	output << "\nsquare_2.print(output);";
	output << "\n--------------------------------------------------------------------------------------------------";
	SQUARE square_0 = SQUARE("red", POINT(0,0), POINT(0,1), POINT(0,2), POINT(0,3));
	square_0.print(output);
	SQUARE square_1 = SQUARE("green", POINT(0,0), POINT(0,1), POINT(5,1), POINT(5,0));
	square_1.print(output);
	SQUARE square_2 = SQUARE("blue", POINT(0,0), POINT(0,1), POINT(1,1), POINT(0,0));
	square_2.print(output);
}

/**
 * Unit Test # 21: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated TRILATERAL instance.
 * Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class and the getter methods of the POLYGON class.
 */
void unit_test_21(std::ostream & output)
{
	output << "\n\n--------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 21: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated TRILATERAL instance. Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class and the getter methods of the POLYGON class.";
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\n// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.";
```

```
	output << "\nPOLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as TRILATERAL.";
	output << "\npointer_to_polygon = new TRILATERAL; // Assign memory to a dynamic TRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";
	output << "\npointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.";
	// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.
	POLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as SQUARE.
	pointer_to_polygon = new TRILATERAL; // Assign memory to a dynamic TRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
	pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.
	output << "\npointer_to_polygon -> get_area() = " << pointer_to_polygon -> get_area() << ". // Indirectly call the POLYGON get_area() method.";
	output << "\npointer_to_polygon -> get_perimeter() = " << pointer_to_polygon -> get_perimeter() << ". // Indirectly call the POLYGON get_permieter() method.";
	output << "\ndelete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated TRILATERAL instance.";
	output << "\n--------------------------------------------------------------------------------------------------";
	delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated TRILATERAL instance.
}

/**
 * Unit Test # 22: Test the default TRILATERAL constructor and the TRILATERAL print method.
 */
void unit_test_22(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 22: Test the default TRILATERAL constructor and the TRILATERAL print method.";
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\nTRILATERAL trilateral;";
	output << "\ntrilateral.print(); // Test the default argument (which is std::cout).";
	output << "\ntrilateral.print(output);";
	output << "\noutput << trilateral; // overloaded ostream operator as defined in trilateral.cpp";
	output << "\n--------------------------------------------------------------------------------------------------";
	TRILATERAL trilateral;
```

```
	trilateral.print(); // Test the default argument (which is std::cout).
	trilateral.print(output);
	output << trilateral; // overloaded ostream operator as defined in trilateral.cpp
}

/**
 * Unit Test # 23: Test the normal TRILATERAL constructor and TRILATERAL copy constructor
using valid function inputs and the TRILATERAL print method.
 */
void unit_test_23(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 23: Test the normal TRILATERAL constructor and TRILATREAL
copy constructor using valid function inputs and the TRILATERAL print method.";
	output << "\n--------------------------------------------------------------------------------------------";
	output << "\nTRILATERAL trilateral_0 = TRILATERAL(\"purple\", POINT(0,0),
POINT(10,0), POINT(10,-10));";
	output << "\ntrilateral_0.print(output);";
	output << "\nTRILATERAL trilateral_1 = TRILATERAL(\"green\", POINT(-1,-1),
POINT(4,4), POINT(7,-18);";
	output << "\ntrilateral_1.print(output);";
	output << "\nTRILATERAL trilateral_2 = TRILATERAL(trilateral_0);";
	output << "\ntrilateral_2.print(output);";
	output << "\n--------------------------------------------------------------------------------------------";
	TRILATERAL trilateral_0 = TRILATERAL("purple", POINT(0,0), POINT(10,0),
POINT(10,-10));
	trilateral_0.print(output);
	TRILATERAL trilateral_1 = TRILATERAL("green", POINT(-1,-1), POINT(4,4),
POINT(7,-18));
	trilateral_1.print(output);
	TRILATERAL trilateral_2 = TRILATERAL(trilateral_0);
	trilateral_2.print(output);
}

/**
 * Unit Test # 24: Test the normal SQUARE constructor using invalid function inputs and the
SQUARE print method.
 */
void unit_test_24(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 24: Test the normal TRILATERAL constructor using invalid
function inputs and the TRILATERAL print method.";
	output << "\n--------------------------------------------------------------------------------------------";
```

```
	output << "\nTRILATERAL trilateral_0 = TRILATERAL(\"red\", POINT(-1,-1), POINT(0,0), POINT(1,1));";
	output << "\ntrilateral_0.print(output);";
	output << "\nTRILATERAL trilateral_1 = TRILATERAL(\"green\", POINT(5,0), POINT(5,1), POINT(5,0));";
	output << "\ntrilateral_1.print(output);";
	output << "\n--------------------------------------------------------------------------------------------------";
	TRILATERAL trilateral_0 = TRILATERAL("red", POINT(-1,-1), POINT(0,0), POINT(1,1));
	trilateral_0.print(output);
	TRILATERAL trilateral_1 = TRILATERAL("green", POINT(5,0), POINT(5,1), POINT(5,0));
	trilateral_1.print(output);
}

/**
 * Unit Test # 25: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated RIGHT_TRILATERAL instance.
 * Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class and the getter methods of the POLYGON class.
 */
void unit_test_25(std::ostream & output)
{
	output << "\n\n--------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 25: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated RIGHT_TRILATERAL instance. Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class and the getter methods of the POLYGON class.";
	output << "\n--------------------------------------------------------------------------------------------------";
	output << "\n// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.";
	output << "\nPOLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as RIGHT_TRILATERAL.";
	output << "\npointer_to_polygon = new RIGHT_TRILATERAL; // Assign memory to a dynamic RIGHT_TRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";
	output << "\npointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.";
	// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.
	POLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as RIGHT_TRILATERAL.
```

pointer_to_polygon = new RIGHT_TRILATERAL; // Assign memory to a dynamic RIGHT_TRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.
output << "\npointer_to_polygon -> get_area() = " << pointer_to_polygon -> get_area() << ". // Indirectly call the POLYGON get_area() method.";
output << "\npointer_to_polygon -> get_perimeter() = " << pointer_to_polygon -> get_perimeter() << ". // Indirectly call the POLYGON get_permieter() method.";
output << "\ndelete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated RIGHT_TRILATERAL instance.";
output << "\n--";
delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated RIGHT_TRILATERAL instance.
}

/**
 * Unit Test # 26: Create a pointer-to-TRILATERAL type variable to store the memory address of a dynamically allocated RIGHT_TRILATERAL instance.
 * Use that pointer-to-TRILATERAL type variable to call the print method of the TRILATERAL class and the getter methods of the TRILATERAL class.
 */
void unit_test_26(std::ostream & output)
{
output << "\n\n--";
output << "\nUnit Test # 25: Create a pointer-to-TRILATERAL type variable to store the memory address of a dynamically allocated RIGHT_TRILATERAL instance. Use that pointer-to-TRILATERAL type variable to call the print method of the TRILATERAL class and the getter methods of the TRILATERAL class.";
output << "\n--";
output << "\nTRILATERAL * pointer_to_trilateral; // The pointer-to-TRILATERAL type variable can store the memory address of an object whose data type is a non-abstract derived class of TRILATERAL such as RIGHT_TRILATERAL.";
output << "\npointer_to_trilateral = new RIGHT_TRILATERAL; // Assign memory to a dynamic RIGHT_TRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).";
output << "\npointer_to_trilateral -> print(output); // Indirectly call the TRILATERAL print method.";

TRILATERAL * pointer_to_trilateral; // The pointer-to-TRILATERAL type variable can store the memory address of an object whose data type is a non-abstract derived class of TRILATERAL such as RIGHT_TRILATERAL.

```
	pointer_to_trilateral = new RIGHT_TRILATERAL; // Assign memory to a dynamic
RIGHT_TRILATERAL instance (i.e. and dynamic implies that the variable was created during
program runtime instead of program compile time).
	pointer_to_trilateral -> print(output); // Indirectly call the TRILATERAL print method.
	output << "\npointer_to_trilateral -> get_area() = " << pointer_to_trilateral -> get_area()
<< ". // Indirectly call the TRILATERAL get_area() method.";
	output << "\npointer_to_trilateral -> get_perimeter() = " << pointer_to_trilateral ->
get_perimeter() << ". // Indirectly call the TRILATERAL get_permieter() method.";
	output << "\ndelete pointer_to_trilateral; // De-allocate memory which was assigned to
the dynamically allocated RIGHT_TRILATERAL instance.";
	output << "\n--------------------------------------------------------------------------------";
	delete pointer_to_trilateral; // De-allocate memory which was assigned to the
dynamically allocated RIGHT_TRILATERAL instance.
}

/**
 * Unit Test # 27: Test the default TRILATERAL constructor and the TRILATERAL print method.
 */
void unit_test_27(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------";
	output << "\nUnit Test # 27: Test the default RIGHT_TRILATERAL constructor and the
RIGHT_TRILATERAL print method.";
	output << "\n--------------------------------------------------------------------------------";
	output << "\nRIGHT_TRILATERAL right_trilateral;";
	output << "\nright_trilateral.print(); // Test the default argument (which is std::cout).";
	output << "\nright_trilateral.print(output);";
	output << "\noutput << right_trilateral; // overloaded ostream operator as defined in
right_trilateral.cpp";
	output << "\n--------------------------------------------------------------------------------";
	RIGHT_TRILATERAL right_trilateral;
	right_trilateral.print(); // Test the default argument (which is std::cout).
	right_trilateral.print(output);
	output << right_trilateral; // overloaded ostream operator as defined in right_trilateral.cpp
}

/**
 * Unit Test # 28: Test the normal RIGHT_TRILATERAL constructor and RIGHT_TRILATERAL
copy constructor using valid function inputs and the RIGHT_TRILATERAL print method.
 */
void unit_test_28(std::ostream & output)
{
	output << "\n--------------------------------------------------------------------------------";
```

```
	output << "\nUnit Test # 28: Test the normal RIGHT_TRILATERAL constructor and 
RIGHT_TRILATERAL copy constructor using valid function inputs and the RIGHT_TRILATERAL 
print method.";
	output << "\n----------------------------------------------------------------------------------------------------";
	output << "\nRIGHT_TRILATERAL right_trilateral_0 = RIGHT_TRILATERAL(\"purple\", 
POINT(0,0), POINT(0,100), POINT(100,0));";
	output << "\nright_trilateral_0.print(output);";
	output << "\nRIGHT_TRILATERAL right_trilateral_1 = RIGHT_TRILATERAL(\"green\", 
POINT(-3,0), POINT(0,-4), POINT(0,0);";
	output << "\nright_trilateral_1.print(output);";
	output << "\nRIGHT_TRILATERAL right_trilateral_2 = 
RIGHT_TRILATERAL(right_trilateral_0);";
	output << "\nright_trilateral_2.print(output);";
	output << "\n----------------------------------------------------------------------------------------------------";
	RIGHT_TRILATERAL right_trilateral_0 = RIGHT_TRILATERAL("purple", POINT(0,0), 
POINT(0,100), POINT(100,0));
	right_trilateral_0.print(output);
	RIGHT_TRILATERAL right_trilateral_1 = RIGHT_TRILATERAL("green", POINT(-3,0), 
POINT(0,-4), POINT(0,0));
	right_trilateral_1.print(output);
	RIGHT_TRILATERAL right_trilateral_2 = RIGHT_TRILATERAL(right_trilateral_0);
	right_trilateral_2.print(output);
}

/**
 * Unit Test # 29: Test the normal RIGHT_TRILATERAL constructor using invalid function inputs 
and the RIGHT_TRILATERAL print method.
 */
void unit_test_29(std::ostream & output)
{
	output << "\n----------------------------------------------------------------------------------------------------";
	output << "\nUnit Test # 29: Test the normal RIGHT_TRILATERAL constructor using 
invalid function inputs and the RIGHT_TRILATERAL print method.";
	output << "\n----------------------------------------------------------------------------------------------------";
	output << "\nRIGHT_TRILATERAL right_trilateral_0 = RIGHT_TRILATERAL(\"red\", 
POINT(-2,-2), POINT(0,0), POINT(4,4));";
	output << "\nright_trilateral_0.print(output);";
	output << "\nRIGHT_TRILATERAL right_trilateral_1 = RIGHT_TRILATERAL(\"green\", 
POINT(0,0), POINT(4,5), POINT(9,-3));";
	output << "\nright_trilateral_1.print(output);";
	output << "\nRIGHT_TRILATERAL right_trilateral_2 = RIGHT_TRILATERAL(\"blue\", 
POINT(0,0), POINT(4,5), POINT(0,0));";
	output << "\nright_trilateral_2.print(output);";
	output << "\n----------------------------------------------------------------------------------------------------";
```

```
	RIGHT_TRILATERAL right_trilateral_0 = RIGHT_TRILATERAL("red", POINT(-2,-2), POINT(0,0), POINT(4,4));
	right_trilateral_0.print(output);
	RIGHT_TRILATERAL right_trilateral_1 = RIGHT_TRILATERAL("green", POINT(0,0), POINT(4,5), POINT(9,-3));
	right_trilateral_1.print(output);
	RIGHT_TRILATERAL right_trilateral_2 = RIGHT_TRILATERAL("blue", POINT(0,0), POINT(4,5), POINT(0,0));
	right_trilateral_2.print(output);
}

/* program entry point */
int main()
{
	// Declare a file output stream object.
	std::ofstream file;

	// Set the number of digits of floating-point numbers which are printed to the command line terminal to 100 digits.
	std::cout.precision(100);

	// Set the number of digits of floating-point numbers which are printed to the file output stream to 100 digits.
	file.precision(100);

	/**
	* If polygon_class_inheritance_tester_output.txt does not already exist in the same directory as polygon_class_inheritance_tester.cpp,
	* create a new file named polygon_class_inheritance_tester_output.txt.
	*
	* Open the plain-text file named polygon_class_inheritance_tester_output.txt
	* and set that file to be overwritten with program data.
	*/
	file.open("polygon_class_inheritance_tester_output.txt");

	// Print an opening message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nStart Of Program";
	std::cout << "\n--------------------------------";

	// Print an opening message to the file output stream.
	file << "--------------------------------";
	file << "\nStart Of Program";
	file << "\n--------------------------------";
```

```
	// Implement a series of unit tests which demonstrate the functionality of POLYGON class variables.
	unit_test_0(std::cout);
	unit_test_0(file);
	unit_test_1(std::cout);
	unit_test_1(file);
	unit_test_2(std::cout);
	unit_test_2(file);
	unit_test_3(std::cout);
	unit_test_3(file);
	unit_test_4(std::cout);
	unit_test_4(file);
	unit_test_5(std::cout);
	unit_test_5(file);
	unit_test_6(std::cout);
	unit_test_6(file);
	unit_test_7(std::cout);
	unit_test_7(file);
	unit_test_8(std::cout);
	unit_test_8(file);
	unit_test_9(std::cout);
	unit_test_9(file);
	unit_test_10(std::cout);
	unit_test_10(file);
	unit_test_11(std::cout);
	unit_test_11(file);
	unit_test_12(std::cout);
	unit_test_12(file);
	unit_test_13(std::cout);
	unit_test_13(file);
	unit_test_14(std::cout);
	unit_test_14(file);
	unit_test_15(std::cout);
	unit_test_15(file);
	unit_test_16(std::cout);
	unit_test_16(file);
	unit_test_17(std::cout);
	unit_test_17(file);
	unit_test_18(std::cout);
	unit_test_18(file);
	unit_test_19(std::cout);
	unit_test_19(file);
	unit_test_20(std::cout);
```

```
	unit_test_20(file);
	unit_test_21(std::cout);
	unit_test_21(file);
	unit_test_22(std::cout);
	unit_test_22(file);
	unit_test_23(std::cout);
	unit_test_23(file);
	unit_test_24(std::cout);
	unit_test_24(file);
	unit_test_25(std::cout);
	unit_test_25(file);
	unit_test_26(std::cout);
	unit_test_26(file);
	unit_test_27(std::cout);
	unit_test_27(file);
	unit_test_28(std::cout);
	unit_test_28(file);
	unit_test_29(std::cout);
	unit_test_29(file);

	// Print a closing message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nEnd Of Program";
	std::cout << "\n--------------------------------\n\n";

	// Print a closing message to the file output stream.
	file << "\n\n--------------------------------";
	file << "\nEnd Of Program";
	file << "\n--------------------------------";

	// Close the file output stream.
	file.close();

	// Exit the program.
	return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/polygon_class_inheritance_tester_output.txt

---

Start Of Program

Unit Test # 0: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated QUADRILATERAL instance. Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class and the getter methods of the POLYGON class.

// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.
POLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as QUADRILATERAL.
pointer_to_polygon = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.

----
memory_address = 0x562bfd0084b0.
category = POLYGON.
color = orange.
&category = 0x562bfd0084b8.
&color = 0x562bfd0084d8.
----
pointer_to_polygon -> get_area() = 20. // Indirectly call the POLYGON get_area() method.
pointer_to_polygon -> get_perimeter() = 18. // Indirectly call the POLYGON get_permieter() method.
delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated QUADRILATERAL instance.

Unit Test # 1: Test the default QUADRILATERAL constructor and the QUADRILATERAL print method.

QUADRILATERAL quadrilateral;
quadrilateral.print(); // Test the default argument (which is std::cout).

quadrilateral.print(output);
output << quadrilateral; // overloaded ostream operator as defined in quadrilateral.cpp
----

----
this = 0x7ffe7c17e640. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e668. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e6a8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e6b0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e6b8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e6c0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.
color = orange. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position

along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A =
90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B =
90.0000760198120559607559698633849620819091796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C =
90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D =
90.0000760198120559607559698633849620819091796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line

segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 18. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 20. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e640. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e668. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e6a8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e6b0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e6b8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e6c0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.
color = orange. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).

B = POINT(0,5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.0000760198120559607559698633849620819091796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).

interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.0000760198120559607559698633849620819091796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 18. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 20. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

Unit Test # 2: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated QUADRILATERAL instance. Use that pointer-to-POLYGON to call the overloaded ostream operator method of the POLYGON class (and not of the QUADRILATERAL class).

// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.
POLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as QUADRILATERAL.
pointer_to_polygon = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
output << * pointer_to_polygon; // Use the overloaded ostream operator as defined in polygon.cpp to print the data which is stored at the memory address which pointer_to_polygon stores.
delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated QUADRILATERAL instance.

----

memory_address = 0x562bfd0084b0.
category = POLYGON.
color = orange.
&category = 0x562bfd0084b8.
&color = 0x562bfd0084d8.

Unit Test # 3: Create a pointer-to-QUADRILATERAL type variable to store the memory address of a dynamically allocated QUADRILATERAL instance. Use that pointer-to-QUADRILATERAL to call the overloaded ostream operator method of the QUADRILATERAL class and the public getter methods of the QUADRILATERAL class.

QUADRILATERAL * pointer_to_quadrilateral; // The pointer-to-QUADRILATERAL type variable can store the memory address of an object whose data type is QUADRILATERAL or else a non-abstract derived class of QUADRILATERAL such as TRAPEZOID.
pointer_to_quadrilateral = new QUADRILATERAL; // Assign memory to a dynamic QUADRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
output < get_area() = 20. // Indirectly call the get_area() method of the QUADRILATERAL class.
pointer_to_quadrilateral -> get_perimeter() = 18. // Indirectly call the get_perimeter() method of the QUADRILATERAL class.

Unit Test # 4: Test the normal QUADRILATERAL constructor and QUADRILATERAL copy constructor using valid function inputs and the QUADRILATERAL print method.

QUADRILATERAL quadrilateral_0 = QUADRILATERAL("green", POINT(-2,-2), POINT(-2,2), POINT(2,2), POINT(2,-2));
quadrilateral_0.print(output);
QUADRILATERAL quadrilateral_1 = QUADRILATERAL("blue", POINT(0,0), POINT(3,2), POINT(5,1), POINT(-1,-2));
quadrilateral_1.print(output);
QUADRILATERAL quadrilateral_2 = QUADRILATERAL(quadrilateral_0);
quadrilateral_2.print(output);

----

this = 0x7ffe7c17e520. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e568. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.

&color = 0x7ffe7c17e548. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e588. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e590. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e598. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e5a0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.
color = green. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(-2,-2). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(-2,2). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(2,2). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(2,-2). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.

d = A.get_distance_from(B) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.0000760198120843824654002673923969268798828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.000076019812041749901254661381244659423828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.000076019812041749901254661381244659423828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 16. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 15.99999999999999289457264239899814128875732421875. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are

enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----

this = 0x7ffe7c17e5b0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e5f8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e5d8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e618. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e620. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e628. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e630. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.
color = blue. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(3,2). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(5,1). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(-1,-2). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position

along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 2.236067977499789805051477742381393909454345703125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 6.7082039324993694151544332271441817283630371093750. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 2.236067977499789805051477742381393909454345703125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 3.605551275463989124858699142350815236568450927734375. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = 0.66666666666666662965923251249478198587894439697265625. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = -0.5. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = 0.5. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 2. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 150.2552456182343689761182758957147598266601 5625. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 119.7449824412019125929873553104698657989501953125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 53.130147231271493524218385573476552963256835937 5. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).

interior_angle_CDA = interior_angle_of_D = 36.86992878854054822568286908790469169616 69921875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248337529861601069569587707519 53125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 14.785891162962938594205297704320400953292846 6796875. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 8. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e640. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e668. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e6a8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e6b0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e6b8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e6c0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.

color = green. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(-2,-2). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(-2,2). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(2,2). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(2,-2). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.0000760198120843824654002673923969268798828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).

interior_angle_ABC = interior_angle_of_B = 90.000076019812041749901254661381244659423828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.000076019812041749901254661381244659423828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 16. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 15.99999999999999289457264239899814128875732421875. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

Unit Test # 5: Test the normal QUADRILATERAL constructor using invalid function inputs and the QUADRILATERAL print method.

QUADRILATERAL quadrilateral_0 = QUADRILATERAL("red", POINT(-2,-2), POINT(0,0), POINT(1,1), POINT(2,2)); // A line intersects all four points.
quadrilateral_0.print(output);
QUADRILATERAL quadrilateral_1 = QUADRILATERAL("purple", POINT(0,0), POINT(3,2), POINT(0,0), POINT(-1,-2)); // Not all point coordinate pairs are unique.
quadrilateral_1.print(output);
QUADRILATERAL quadrilateral_2 = QUADRILATERAL("yellow", POINT(0,0), POINT(0,2), POINT(4,0), POINT(4,2)); // The points form a bow-tie shaped polygon.
quadrilateral_2.print(output);

----
this = 0x7ffe7c17e520. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e568. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e548. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e588. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e590. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e598. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e5a0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.
color = red. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.

b = C.get_distance_from(D) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.0000760198120559607559698633849620819091796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.0000760198120559607559698633849620819091796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior

angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 18. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 20. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e5b0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e5f8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e5d8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e618. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e620. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e628. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e630. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.
color = purple. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).

C = POINT(4,5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.0000760198120559607559698633849620819091796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line

segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.0000760198120559607559698633849620819091796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 18. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 20. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e640. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e668. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e6a8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e6b0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e6b8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e6c0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.

color = yellow. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,2). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,2). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4.47213595499957961010295548476278781890869140625. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 2. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4.47213595499957961010295548476278781890869140625. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 2. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = -0.5. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0.5. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 63.4350024041763163040741346776485443115234375. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).

interior_angle_ABC = interior_angle_of_B = 116.565149635447824039147235453128814697265625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 63.4350024041763163040741346776485443115234375. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 116.565149635447824039147235453128814697265625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 12.9442719099991592202059109695255756378173828125. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 8. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

Unit Test # 6: Test the default TRAPEZOID constructor and the TRAPEZOID print method.

TRAPEZOID trapezoid;
trapezoid.print(); // Test the default argument (which is std::cout).
trapezoid.print(output);
output < print(output); // Indirectly call the POLYGON print method.

----
memory_address = 0x562bfd008670.
category = POLYGON.
color = orange.
&category = 0x562bfd008678.
&color = 0x562bfd008698.
----

pointer_to_polygon -> get_area() = 1.9999999999999993338661852249060757458209991455078125. // Indirectly call the POLYGON get_area() method.
pointer_to_polygon -> get_perimeter() = 6.8284271247461898468600338674150407314300537109375. // Indirectly call the POLYGON get_permieter() method.
delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated TRAPEZOID instance.

Unit Test # 8: Create a pointer-to-QUADRILATERAL type variable to store the memory address of a dynamically allocated TRAPEZOID instance. Use that pointer-to-QUADRILATERAL type variable to call the QUADRILATERAL print method.

QUADRILATERAL * pointer_to_quadrilateral; // The pointer-to-QUADRILATERAL type variable can store the memory address of an object whose data type is a non-abstract derived class of QUADRILATERAL such as TRAPEZOID.
pointer_to_quadrilateral = new TRAPEZOID; // Assign memory to a dynamic TRAPEZOID instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
pointer_to_quadrilateral -> print(output); // Indirectly call the QUADRILATERAL print method.
delete pointer_to_quadrilateral; // De-allocate memory which was assigned to the dynamically allocated TRAPEZOID instance.

----
this = 0x562bfd008670. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x562bfd0086b8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x562bfd008698. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x562bfd0086d8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x562bfd0086e0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.

&C = 0x562bfd0086e8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x562bfd0086f0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.
color = orange. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(1,1). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(2,1). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(3,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = 1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.

C.get_slope_of_line_to(D) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 45.00003800990604219123270013369619846343994140625. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 135.00011402971807683570659719407558441162109375. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 135.0001140297181336791254580020904541015625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 45.00003800990599955866855452768504619598388671875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 6.8284271247461898468600338674150407314300537109375. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 1.99999999999999993338661852249060757458209991455078125. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

Unit Test # 9: Test the normal TRAPEZOID constructor and TRAPEZOID copy constructor using valid function inputs and the TRAPEZOID print method.

TRAPEZOID trapezoid_0 = TRAPEZOID("pink", POINT(-5,-10), POINT(0,5), POINT(15,5), POINT(30,-10));
trapezoid_0.print(output);
TRAPEZOID trapezoid_1 = TRAPEZOID("brown", POINT(0,0), POINT(4,-5), POINT(15,-5), POINT(45,0));
trapezoid_1.print(output);
TRAPEZOID trapezoid_2 = TRAPEZOID(trapezoid_0);
trapezoid_2.print(output);

----
this = 0x7ffe7c17e4c0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e548. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e4e8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e528. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e530. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e538. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e540. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRAPEZOID) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRAPEZOID type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRAPEZOID *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRAPEZOID type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRAPEZOID **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRAPEZOID type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/TRAPEZOID. // This is an immutable string type value which is a data member of the caller TRAPEZOID object.

color = pink. // This is a string type value which is a data member of the caller TRAPEZOID object.
A = POINT(-5,-10). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(15,5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(30,-10). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 15. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 21.2132034355964265159855131059885025024414O625. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 35. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 15.8113883008418962816676867078058421611785888671875. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = 3. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 71.5651162554177474248717771843075752258300781 25. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).

interior_angle_ABC = interior_angle_of_B = 108.4350404140823940224436228163540363311767578125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 135.00011402971807683570659719407558441162109375. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 45.00003800990604219123270013369619846343994140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.00030407924833752986160106956958770751953125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 87.0245917364383245740100392140448093414306640625. // The method returns the sum of the four approximated side lengths of the trapezoid which the caller TRAPEZOID object represents.
get_area() = 375. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e570. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e5f8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e598. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e5d8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.

&B = 0x7ffe7c17e5e0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e5e8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e5f0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRAPEZOID) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRAPEZOID type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRAPEZOID *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRAPEZOID type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRAPEZOID **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRAPEZOID type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/TRAPEZOID. // This is an immutable string type value which is a data member of the caller TRAPEZOID object.
color = brown. // This is a string type value which is a data member of the caller TRAPEZOID object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(4,-5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(15,-5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(45,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 11. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 30.413812651491099359191139228641986846923828125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.

c = D.get_distance_from(A) = 45. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 6.4031242374328485311707481741905212402343750. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = -1.25. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = 0.1666666666666666574148081281236954964697360992431640625. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 51.340235111151308444732421776279807090759277343750. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 128.659916928472881636480451561510562896728515625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 170.53782183910999492582050152122974395751953125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 9.462330200513985545285322587005794048309326171875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.00030407924816699960501864552497863769531 25. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)

get_perimeter() = a + b + c + d = 92.816936888923947890361887402832508087158203125. // The method returns the sum of the four approximated side lengths of the trapezoid which the caller TRAPEZOID object represents.
get_area() = 140.000000000001136868377216160297393798828125. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e620. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e6a8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e648. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e690. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e698. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e6a0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRAPEZOID) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRAPEZOID type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRAPEZOID *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRAPEZOID type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRAPEZOID **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRAPEZOID type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/TRAPEZOID. // This is an immutable string type value which is a data member of the caller TRAPEZOID object.

color = pink. // This is a string type value which is a data member of the caller TRAPEZOID object.
A = POINT(-5,-10). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(15,5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(30,-10). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 15. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 21.21320343559642651598551310598850250244140625. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 35. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 15.8113883008418962816676867078058421611785888671875. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = 3. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 71.5651116255417747424871777184307575225830078125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).

interior_angle_ABC = interior_angle_of_B =
108.4350404140823940224436228163540363311767578125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C =
135.000114029718076835706597194075584411621093750. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D =
45.0000380099060421912327001336961984634399414062 5. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D =
360.0003040792483375298616010695695877075195312 5. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 87.0245917364383245740100392140448093414306640625.
// The method returns the sum of the four approximated side lengths of the trapezoid which the caller TRAPEZOID object represents.
get_area() = 375. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

Unit Test # 10: Test the default RECTANGLE constructor and the RECTANGLE print method.

RECTANGLE rectangle;
rectangle.print(); // Test the default argument (which is std::cout).
rectangle.print(output);
output < print(output); // Indirectly call the POLYGON print method.

----

memory_address = 0x562bfd008670.
category = POLYGON.
color = orange.
&category = 0x562bfd008678.
&color = 0x562bfd008698.

// COMMENTED OUT (does not work): pointer_to_polygon -> quadrilateral_test(). // Indirectly call the QUADRILATERAL quadrilateral_test() method.
pointer_to_polygon -> get_area() = 12. // Indirectly call the POLYGON get_area() method.
pointer_to_polygon -> get_perimeter() = 14. // Indirectly call the POLYGON get_permieter() method.
delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated RECTANGLE instance.

Unit Test # 12: Create a pointer-to-QUADRILATERAL type variable to store the memory address of a dynamically allocated RECTANGLE instance. Use that pointer-to-QUADRILATERAL type variable to call the POLYGON print method and the QUADRILATERAL getter methods.

QUADRILATERAL * pointer_to_quadrilateral; // The pointer-to-QUADRILATERAL type variable can store the memory address of an object whose data type is a non-abstract derived class of QUADRILATERAL such as RECTANGLE.
pointer_to_quadrilateral = new RECTANGLE; // Assign memory to a dynamic RECTANGLE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
pointer_to_quadrilateral -> print(output); // Indirectly call the QUADRILATERAL print method.

----
this = 0x562bfd008670. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x562bfd0086b8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x562bfd008698. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x562bfd0086d8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x562bfd0086e0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x562bfd0086e8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x562bfd0086f0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.

sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).

category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.
color = orange. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,3). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,3). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line

segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 14. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 12. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

// COMMENTED OUT (does not work): pointer_to_quadrilateral -> rectangle_test(); // Indirectly call the RECTANGLE rectangle_test() method.
pointer_to_quadrilateral -> get_area() = 12. // Indirectly call the QUADRILATERAL get_area() method.
pointer_to_quadrilateral -> get_perimeter() = 14. // Indirectly call the QUADRILATERAL get_permieter() method.
delete pointer_to_quadrilateral; // De-allocate memory which was assigned to the dynamically allocated RECTANGLE instance.

Unit Test # 13: Test the normal RECTANGLE constructor and RECTANGLE copy constructor using valid function inputs and the RECTANGLE print method.

RECTANGLE rectangle_0 = RECTANGLE("gray", POINT(9,10), POINT(9,5), POINT(3,5), POINT(3,10));
rectangle_0.print(output);
RECTANGLE rectangle_1 = RECTANGLE("black", POINT(0,0), POINT(0,1), POINT(1,1), POINT(1,0));
rectangle_1.print(output);
RECTANGLE rectangle_2 = RECTANGLE(rectangle_0);
rectangle_2.print(output);

----
this = 0x7ffe7c17e4c0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e548. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e4e8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e528. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e530. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e538. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e540. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.

color = gray. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(9,10). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(9,5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(3,5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(3,10). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 6. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 6. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).

interior_angle_ABC = interior_angle_of_B = 90.00007601981205596075596986338496208190917968750. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.00007601981205596075596986338496208190917968750. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.0003040792482806864427402615547180175781250. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 22. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.
get_area() = 30. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e570. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e5f8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e598. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e5d8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.

&B = 0x7ffe7c17e5e0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e5e8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e5f0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = black. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,1). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(1,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.

c = D.get_distance_from(A) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.0000760198120843824654002673923969268798828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.000076019812041749901254661381244659423828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.000076019812041749901254661381244659423828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 4. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.

get_area() = 0.999999999999999555910790149937383830547332763671875. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----

this = 0x7ffe7c17e620. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e6a8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e648. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e690. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e698. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e6a0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = gray. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(9,10). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position

along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(9,5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(3,5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(3,10). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 6. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 6. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.0000760198120559607559698633849620819091796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line

segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C =
90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D =
90.000076019812055960755969863384962081909179 6875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D =
360.0003040792482806864427402615547180175781 25. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 22. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.
get_area() = 30. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

Unit Test # 14: Test the normal RECTANGLE constructor using invalid function inputs and the RECTANGLE print method.

RECTANGLE rectangle_0 = RECTANGLE("red", POINT(-1,-1), POINT(0,0), POINT(1,1), POINT(2,2));
rectangle_0.print(output);
RECTANGLE rectangle_1 = RECTANGLE("green", POINT(-5,-10), POINT(0,5), POINT(15,5), POINT(30,-10));
rectangle_1.print(output);
RECTANGLE rectangle_2 = RECTANGLE("blue", POINT(-5,-5), POINT(0,0), POINT(-5,-5), POINT(0,0));
rectangle_2.print(output);

----
this = 0x7ffe7c17e4c0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.

&category = 0x7ffe7c17e548. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e4e8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e528. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e530. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e538. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e540. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = red. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,3). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,3). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).

D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.00007601981207017161068506538867950439453125. // The method returns the

approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 14. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.
get_area() = 12. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e570. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e5f8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e598. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e5d8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e5e0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e5e8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e5f0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = green. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,3). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,3). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.

interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 14. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.
get_area() = 12. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e620. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e6a8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.

&color = 0x7ffe7c17e648. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e690. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e698. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e6a0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = blue. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,3). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,3). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).

a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).

interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 14. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.
get_area() = 12. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

Unit Test # 15: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated SQUARE instance. Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class and the getter methods of the POLYGON class.

// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.
POLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as SQUARE.
pointer_to_polygon = new SQUARE; // Assign memory to a dynamic SQUARE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.

----
memory_address = 0x562bfd0089b0.
category = POLYGON.
color = orange.
&category = 0x562bfd0089b8.
&color = 0x562bfd0089d8.
----
pointer_to_polygon -> get_area() = 25. // Indirectly call the POLYGON get_area() method.
pointer_to_polygon -> get_perimeter() = 20. // Indirectly call the POLYGON get_permieter() method.
delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated SQUARE instance.

Unit Test # 16: Test the default SQUARE constructor and the SQUARE print method.

SQUARE square;
square.print(); // Test the default argument (which is std::cout).

square.print(output);
output < print(output); // Indirectly call the QUADRILATERAL print method.

----
this = 0x562bfd0089b0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x562bfd0089f8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x562bfd0089d8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x562bfd008a18. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x562bfd008a20. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x562bfd008a28. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x562bfd008a30. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL. // This is an immutable string type value which is a data member of the caller QUADRILATERAL object.
color = orange. // This is a string type value which is a data member of the caller QUADRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(5,5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(5,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).

a = B.get_distance_from(C) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A =
90.0000760198120843824654002673923969268798828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B =
90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C =
90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D =
90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).

interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248337529861601069569587707519531 25. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 20. // The method returns the sum of the four approximated side lengths of the quadrilateral which the caller QUADRILATERAL object represents.
get_area() = 25. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.
----
pointer_to_quadrilateral -> get_area() = 25. // Indirectly call the QUADRILATERAL get_area() method.
pointer_to_quadrilateral -> get_perimeter() = 20. // Indirectly call the QUADRILATERAL get_permieter() method.
delete pointer_to_quadrilateral; // De-allocate memory which was assigned to the dynamically allocated SQUARE instance.

Unit Test # 18: Create a pointer-to-RECTANGLE type variable to store the memory address of a dynamically allocated SQUARE instance. Use that pointer-to-RECTANGLE type variable to call the RECTANGLE print method and the RECTANGLE getter methods.

RECTANGLE * pointer_to_rectangle; // The pointer-to-RECTANGLE type variable can store the memory address of an object whose data type is a non-abstract derived class of RECTANGLE such as SQUARE.
pointer_to_rectangle = new SQUARE; // Assign memory to a dynamic SQUARE instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
pointer_to_rectangle -> print(output); // Indirectly call the RECTANGLE print method.

----
this = 0x562bfd0089b0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x562bfd008a38. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x562bfd0089d8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x562bfd008a18. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.

&B = 0x562bfd008a20. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x562bfd008a28. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x562bfd008a30. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = orange. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,5). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(5,5). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(5,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.

c = D.get_distance_from(A) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981208438246540026739239692687988828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.00007601981208438246540026739239692687988828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981208438246540026739239692687988828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.00007601981208438246540026739239692687988828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.0003040792483375298616010695695877075195 3125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 20. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.

get_area() = 25. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----

pointer_to_rectangle -> get_area() = 25. // Indirectly call the RECTANGLE get_area() method.
pointer_to_rectangle -> get_perimeter() = 20. // Indirectly call the RECTANGLE get_permieter() method.
delete pointer_to_rectangle; // De-allocate memory which was assigned to the dynamically allocated SQUARE instance.

Unit Test # 19: Test the normal SQUARE constructor and SQUARE copy constructor using valid function inputs and the SQUARE print method.

SQUARE square_0 = SQUARE("yellow", POINT(-3,-3), POINT(-3,0), POINT(0,0), POINT(0,-3));
square_0.print(output);
SQUARE square_1 = SQUARE("white", POINT(-1,-1), POINT(-1,1), POINT(1,1), POINT(1,-1));
square_1.print(output);
SQUARE square_2 = SQUARE(square_0);
square_2.print(output);

----

this = 0x7ffe7c17e460. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e508. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e488. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e4c8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e4d0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e4d8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e4e0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.

sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE) = 200. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a SQUARE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE/SQUARE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = yellow. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(-3,-3). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(-3,0). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(0,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(0,-3). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.

c = D.get_distance_from(A) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.0000760198120559607559698633849620819091796875. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.0000760198120559607559698633849620819091796875. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 12. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.

get_area() = 8.9999999999999946709294817992486059665679931640625. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----

this = 0x7ffe7c17e530. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e5d8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e558. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e598. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e5a0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e5a8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e5b0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE) = 200. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a SQUARE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(SQUARE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE/SQUARE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = white. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(-1,-1). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(-1,1). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,1). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(1,-1). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 2. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 2. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 2. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 2. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.

interior_angle_DAB = interior_angle_of_A = 90.0000760198120843824654002673923969268798828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.00007601981204174990125466138124465942382812 5. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.0000760198120417499012546613812446594238281 25. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.0003040792482806864427402615547180175781 25. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 8. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.
get_area() = 3.99999999999999822364316059974953532218933105 46875. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e600. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e6a8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.

&color = 0x7ffe7c17e628. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e668. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e670. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e678. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e680. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE) = 200. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a SQUARE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE/SQUARE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = yellow. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(-3,-3). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(-3,0). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position

along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(0,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(0,-3). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.0000760198120559607559698633849620819091796875. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.0000760198120843824654002673923969268798828125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.0000760198120559607559698633849620819091796875. // The method returns the

approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.00007601981208438246540026739239692687988281 25. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.0003040792482806864427402615547180175781 25. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 12. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.
get_area() = 8.9999999999999946709294817992486059665679931640625. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

Unit Test # 20: Test the normal SQUARE constructor using invalid function inputs and the SQUARE print method.

SQUARE square_0 = SQUARE("red", POINT(0,0), POINT(0,1), POINT(0,2), POINT(0,3));
square_0.print(output);
SQUARE square_1 = SQUARE("green", POINT(0,0), POINT(0,1), POINT(5,1), POINT(5,0));
square_1.print(output);
SQUARE square_2 = SQUARE("blue", POINT(0,0), POINT(0,1), POINT(1,1), POINT(0,0));
square_2.print(output);

----
this = 0x7ffe7c17e460. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e508. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e488. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..

&A = 0x7ffe7c17e4c8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e4d0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e4d8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e4e0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE) = 200. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a SQUARE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE/SQUARE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = red. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,3). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,3). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position

along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).

interior_angle_CDA = interior_angle_of_D = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 14. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.
get_area() = 12. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----
this = 0x7ffe7c17e530. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e5d8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e558. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e598. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e5a0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e5a8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e5b0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE) = 200. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a SQUARE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE/SQUARE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = green. // This is a string type value which is a data member of the caller RECTANGLE object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,3). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,3). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.

A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 14. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.
get_area() = 12. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

----

this = 0x7ffe7c17e600. // The keyword named this is a pointer which stores the memory address of the first memory cell of a QUADRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRIANGLE object.
&category = 0x7ffe7c17e6a8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e628. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e668. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e670. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e678. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
&D = 0x7ffe7c17e680. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named D.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL) = 136. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a QUADRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(QUADRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-QUADRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE) = 168. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RECTANGLE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RECTANGLE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RECTANGLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE) = 200. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a SQUARE type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(SQUARE **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-SQUARE type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/QUADRILATERAL/RECTANGLE/SQUARE. // This is an immutable string type value which is a data member of the caller RECTANGLE object.
color = blue. // This is a string type value which is a data member of the caller RECTANGLE object.

A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,3). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(4,3). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
D = POINT(4,0). // D represents a point (which is neither A nor B nor C) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(D) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and D.
c = D.get_distance_from(A) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points D and A.
d = A.get_distance_from(B) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
B.get_slope_of_line_to(C) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
C.get_slope_of_line_to(D) = -inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and D.
D.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
interior_angle_DAB = interior_angle_of_A = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are D and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_ABC = interior_angle_of_B = 90.00007601981207017161068506538867950439453125. // The method returns the

approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_BCD = interior_angle_of_C = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and D such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_CDA = interior_angle_of_D = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and D with the line segment whose endpoints are D and A such that those two line segments intersect at D (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C + interior_angle_of_D = 360.000304079248280686442740261554718017578125. // sum of all four approximate interior angle measurements of the quadrilateral represented by the caller QUADRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c + d = 14. // The method returns the sum of the four approximated side lengths of the rectangle which the caller RECTANGLE object represents.
get_area() = 12. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the four line segments which connect points A to B, B to C, C to D, and D to A.

Unit Test # 21: Create a pointer-to-POLYGON type variable to store the memory address of a dynamically allocated TRILATERAL instance. Use that pointer-to-POLYGON type variable to call the print method of the POLYGON class and the getter methods of the POLYGON class.

// COMMENTED OUT: POLYGON polygon; // This command does not work because POLYGON is an abstract class.
POLYGON * pointer_to_polygon; // The pointer-to-POLYGON type variable can store the memory address of an object whose data type is a non-abstract derived class of POLYGON such as TRILATERAL.
pointer_to_polygon = new TRILATERAL; // Assign memory to a dynamic TRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
pointer_to_polygon -> print(output); // Indirectly call the POLYGON print method.

----
memory_address = 0x562bfd0084b0.

category = POLYGON.
color = orange.
&category = 0x562bfd0084b8.
&color = 0x562bfd0084d8.
----
pointer_to_polygon -> get_area() = 6. // Indirectly call the POLYGON get_area() method.
pointer_to_polygon -> get_perimeter() = 12. // Indirectly call the POLYGON get_permieter() method.
delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated TRILATERAL instance.

Unit Test # 22: Test the default TRILATERAL constructor and the TRILATERAL print method.

TRILATERAL trilateral;
trilateral.print(); // Test the default argument (which is std::cout).
trilateral.print(output);
output < print(output); // Indirectly call the POLYGON print method.

----
memory_address = 0x562bfd008670.
category = POLYGON.
color = orange.
&category = 0x562bfd008678.
&color = 0x562bfd008698.
----
pointer_to_polygon -> get_area() = 0.49999999999999997779553950749686919152736663818359375. // Indirectly call the POLYGON get_area() method.
pointer_to_polygon -> get_perimeter() = 3.41421356237309492343001693370752036571502685546875. // Indirectly call the POLYGON get_permieter() method.
delete pointer_to_polygon; // De-allocate memory which was assigned to the dynamically allocated RIGHT_TRILATERAL instance.

Unit Test # 25: Create a pointer-to-TRILATERAL type variable to store the memory address of a dynamically allocated RIGHT_TRILATERAL instance. Use that pointer-to-TRILATERAL type variable to call the print method of the TRILATERAL class and the getter methods of the TRILATERAL class.

TRILATERAL * pointer_to_trilateral; // The pointer-to-TRILATERAL type variable can store the memory address of an object whose data type is a non-abstract derived class of TRILATERAL such as RIGHT_TRILATERAL.
pointer_to_trilateral = new RIGHT_TRILATERAL; // Assign memory to a dynamic RIGHT_TRILATERAL instance (i.e. and dynamic implies that the variable was created during program runtime instead of program compile time).
pointer_to_trilateral -> print(output); // Indirectly call the TRILATERAL print method.

----
this = 0x562bfd008670. // The keyword named this is a pointer which stores the memory address of the first memory cell of a TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller TRILATERAL object.
&category = 0x562bfd0086b8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x562bfd008698. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x562bfd0086d8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x562bfd0086e0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x562bfd0086e8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL) = 128. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/TRILATERAL. // This is an immutable string type value which is a data member of the caller TRILATERAL object.
color = orange. // This is a string type value which is a data member of the caller TRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 1.4142135623730951454746218587388284504413604736328125. // The method returns the

approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(A) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = A.get_distance_from(B) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
slope_of_side_a = B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
slope_of_side_b = C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
slope_of_side_c = A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
interior_angle_of_A = get_interior_angle_CAB() = 90.00007601981208438246540026739239692687988281 25. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_of_B = get_interior_angle_ABC() = 45.000038009906020874950627330690622329711914 0625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_of_C = get_interior_angle_BCA() = 45.000038009906020874950627330690622329711914 0625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C = 180.000152039624140343221370130777359008789062 5. // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.41421356237309492343001693370752036571502685546875. // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.
get_area() = 0.49999999999999977795539507496869191527366638183 59375. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.
----

pointer_to_trilateral -> get_area() =
0.4999999999999997779553950749686919152736663818359375. // Indirectly call the TRILATERAL get_area() method.
pointer_to_trilateral -> get_perimeter() =
3.41421356237309492343001693370752036571502685546875. // Indirectly call the TRILATERAL get_permieter() method.
delete pointer_to_trilateral; // De-allocate memory which was assigned to the dynamically allocated RIGHT_TRILATERAL instance.

Unit Test # 27: Test the default RIGHT_TRILATERAL constructor and the RIGHT_TRILATERAL print method.

RIGHT_TRILATERAL right_trilateral;
right_trilateral.print(); // Test the default argument (which is std::cout).
right_trilateral.print(output);
output << right_trilateral; // overloaded ostream operator as defined in right_trilateral.cpp
----

----
this = 0x7ffe7c17e620. // The keyword named this is a pointer which stores the memory address of the first memory cell of a RIGHT_TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller RIGHT_TRILATERAL object.
&category = 0x7ffe7c17e6a0. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e648. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e690. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e698. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL) = 128. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL) = 160. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RIGHT_TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(RIGHT_TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/TRILATERAL/RIGHT_TRILATERAL. // This is an immutable string type value which is a data member of the caller RIGHT_TRILATERAL object.
color = orange. // This is a string type value which is a data member of the caller RIGHT_TRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(A) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = A.get_distance_from(B) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
slope_of_side_a = B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
slope_of_side_b = C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
slope_of_side_c = A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
interior_angle_of_A = get_interior_angle_CAB() = 90.0000760198120843824654002673923969268798828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_of_B = get_interior_angle_ABC() = 45.00003800990602087495062733069062232971191406250. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle

formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_of_C = get_interior_angle_BCA() = 45.00003800990602087495062733069062232971191406 25. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.414213562373094923430016933707520365715026855 46875. // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.
get_area() = 0.4999999999999997779553950749686919152736663818359375. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.

----
this = 0x7ffe7c17e620. // The keyword named this is a pointer which stores the memory address of the first memory cell of a RIGHT_TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller RIGHT_TRILATERAL object.
&category = 0x7ffe7c17e6a0. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e648. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e690. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e698. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL) = 128. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL) = 160. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RIGHT_TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(RIGHT_TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/TRILATERAL/RIGHT_TRILATERAL. // This is an immutable string type value which is a data member of the caller RIGHT_TRILATERAL object.
color = orange. // This is a string type value which is a data member of the caller RIGHT_TRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(A) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = A.get_distance_from(B) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
slope_of_side_a = B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
slope_of_side_b = C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
slope_of_side_c = A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
interior_angle_of_A = get_interior_angle_CAB() = 90.00007601981208438246540026739239692687988828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).

interior_angle_of_B = get_interior_angle_ABC() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_of_C = get_interior_angle_BCA() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.41421356237309492343001693370752036571502685546875. // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.
get_area() = 0.4999999999999997779553950749686919152736663818359375. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.

Unit Test # 28: Test the normal RIGHT_TRILATERAL constructor and RIGHT_TRILATERAL copy constructor using valid function inputs and the RIGHT_TRILATERAL print method.

RIGHT_TRILATERAL right_trilateral_0 = RIGHT_TRILATERAL("purple", POINT(0,0), POINT(0,100), POINT(100,0));
right_trilateral_0.print(output);
RIGHT_TRILATERAL right_trilateral_1 = RIGHT_TRILATERAL("green", POINT(-3,0), POINT(0,-4), POINT(0,0);
right_trilateral_1.print(output);
RIGHT_TRILATERAL right_trilateral_2 = RIGHT_TRILATERAL(right_trilateral_0);
right_trilateral_2.print(output);

----

this = 0x7ffe7c17e4e0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a RIGHT_TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller RIGHT_TRILATERAL object.

&category = 0x7ffe7c17e560. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e508. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e548. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e550. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e558. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL) = 128. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL) = 160. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RIGHT_TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/TRILATERAL/RIGHT_TRILATERAL. // This is an immutable string type value which is a data member of the caller RIGHT_TRILATERAL object.
color = purple. // This is a string type value which is a data member of the caller RIGHT_TRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,100). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(100,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 141.421356237309510106570087373256683349609375. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.

b = C.get_distance_from(A) = 100. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = A.get_distance_from(B) = 100. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
slope_of_side_a = B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
slope_of_side_b = C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
slope_of_side_c = A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
interior_angle_of_A = get_interior_angle_CAB() = 90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_of_B = get_interior_angle_ABC() = 45.000038009906042191232700133696198463439941 40625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_of_C = get_interior_angle_BCA() = 45.00003800990604219123270013369619846343994140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c = 341.421356237309510106570087373256683349609375. // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.
get_area() = 5000. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.

----

this = 0x7ffe7c17e580. // The keyword named this is a pointer which stores the memory address of the first memory cell of a RIGHT_TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller RIGHT_TRILATERAL object.
&category = 0x7ffe7c17e600. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e5a8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e5e8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e5f0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e5f8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL) = 128. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL) = 160. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RIGHT_TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/TRILATERAL/RIGHT_TRILATERAL. // This is an immutable string type value which is a data member of the caller RIGHT_TRILATERAL object.
color = green. // This is a string type value which is a data member of the caller RIGHT_TRILATERAL object.
A = POINT(-3,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,-4). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(0,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).

a = B.get_distance_from(C) = 4. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(A) = 3. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = A.get_distance_from(B) = 5. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
slope_of_side_a = B.get_slope_of_line_to(C) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
slope_of_side_b = C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
slope_of_side_c = A.get_slope_of_line_to(B) = -1.3333333333333332593184650249895639717578887939453125. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
interior_angle_of_A = get_interior_angle_CAB() = 53.1301472312714935242183855734765529632568359375. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_of_B = get_interior_angle_ABC() = 36.86992878854056954196494189091026782989501953125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_of_C = get_interior_angle_BCA() = 90.00007601981207017161068506538867950439453125. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c = 12. // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.
get_area() = 6. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.

----

this = 0x7ffe7c17e620. // The keyword named this is a pointer which stores the memory address of the first memory cell of a RIGHT_TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller RIGHT_TRILATERAL object.
&category = 0x7ffe7c17e6a0. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e648. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e690. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e698. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL) = 128. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL) = 160. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RIGHT_TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/TRILATERAL/RIGHT_TRILATERAL. // This is an immutable string type value which is a data member of the caller RIGHT_TRILATERAL object.
color = purple. // This is a string type value which is a data member of the caller RIGHT_TRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,100). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(100,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position

along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 141.421356237309510106570087373256683349609375. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(A) = 100. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = A.get_distance_from(B) = 100. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
slope_of_side_a = B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
slope_of_side_b = C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
slope_of_side_c = A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
interior_angle_of_A = get_interior_angle_CAB() =
90.00007601981207017161068506538867950439453125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_of_B = get_interior_angle_ABC() =
45.00003800990604219123270013369619846343994140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_of_C = get_interior_angle_BCA() =
45.00003800990604219123270013369619846343994140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C =
180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c = 341.421356237309510106570087373256683349609375. // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.

get_area() = 5000. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.

Unit Test # 29: Test the normal RIGHT_TRILATERAL constructor using invalid function inputs and the RIGHT_TRILATERAL print method.

RIGHT_TRILATERAL right_trilateral_0 = RIGHT_TRILATERAL("red", POINT(-2,-2), POINT(0,0), POINT(4,4));
right_trilateral_0.print(output);
RIGHT_TRILATERAL right_trilateral_1 = RIGHT_TRILATERAL("green", POINT(0,0), POINT(4,5), POINT(9,-3));
right_trilateral_1.print(output);
RIGHT_TRILATERAL right_trilateral_2 = RIGHT_TRILATERAL("blue", POINT(0,0), POINT(4,5), POINT(0,0));
right_trilateral_2.print(output);

----

this = 0x7ffe7c17e4e0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a RIGHT_TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller RIGHT_TRILATERAL object.
&category = 0x7ffe7c17e560. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e508. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e548. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e550. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e558. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL) = 128. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL) = 160. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RIGHT_TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(RIGHT_TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/TRILATERAL/RIGHT_TRILATERAL. // This is an immutable string type value which is a data member of the caller RIGHT_TRILATERAL object.
color = red. // This is a string type value which is a data member of the caller RIGHT_TRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(A) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = A.get_distance_from(B) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
slope_of_side_a = B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
slope_of_side_b = C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
slope_of_side_c = A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
interior_angle_of_A = get_interior_angle_CAB() = 90.00007601981208438246540026739239692687988828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_of_B = get_interior_angle_ABC() = 45.000038009906020874950627330690622329711914 0625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle

formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_of_C = get_interior_angle_BCA() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.41421356237309492343001693370752036571502685546875. // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.
get_area() = 0.4999999999999997779553950749686919152736663818359375. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.

----
this = 0x7ffe7c17e580. // The keyword named this is a pointer which stores the memory address of the first memory cell of a RIGHT_TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller RIGHT_TRILATERAL object.
&category = 0x7ffe7c17e600. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e5a8. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e5e8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e5f0. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e5f8. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.
sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL) = 128. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL) = 160. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RIGHT_TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(RIGHT_TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/TRILATERAL/RIGHT_TRILATERAL. // This is an immutable string type value which is a data member of the caller RIGHT_TRILATERAL object.
color = green. // This is a string type value which is a data member of the caller RIGHT_TRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(A) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = A.get_distance_from(B) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
slope_of_side_a = B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
slope_of_side_b = C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
slope_of_side_c = A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
interior_angle_of_A = get_interior_angle_CAB() = 90.0000760198120843824654002673923969268798828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).

interior_angle_of_B = get_interior_angle_ABC() = 45.00003800990602087495062733069062229719140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_of_C = get_interior_angle_BCA() = 45.00003800990602087495062733069062229719140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.41421356237309492343001693370752036571502685546875. // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.
get_area() = 0.499999999999999777955395074968691915273666381835937 5. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.

----
this = 0x7ffe7c17e620. // The keyword named this is a pointer which stores the memory address of the first memory cell of a RIGHT_TRILATERAL sized chunk of contiguous memory cells which are allocated to the caller RIGHT_TRILATERAL object.
&category = 0x7ffe7c17e6a0. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named category.
&color = 0x7ffe7c17e648. // The reference operation returns the memory address of the first memory cell of a string sized chunk of contiguous memory cells which are allocated to the string data attribute named color..
&A = 0x7ffe7c17e688. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named A.
&B = 0x7ffe7c17e690. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named B.
&C = 0x7ffe7c17e698. // The reference operation returns the memory address of the first memory cell of a POINT sized chunk of contiguous memory cells which are allocated to the POINT data attribute named C.

sizeof(int) = 4. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which an pointer-to-pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POINT type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POINT **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POINT type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON) = 72. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a POLYGON type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(POLYGON **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-POLYGON type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL) = 128. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(RIGHT_TRILATERAL) = 160. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a RIGHT_TRILATERAL type object occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL *) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(RIGHT_TRILATERAL **) = 8. // The sizeof() operation returns the nonnegative integer number of bytes of memory which a pointer-to-pointer-to-RIGHT_TRILATERAL type variable occupies. (Each memory cell has a data capacity of 1 byte).
category = POLYGON/TRILATERAL/RIGHT_TRILATERAL. // This is an immutable string type value which is a data member of the caller RIGHT_TRILATERAL object.
color = blue. // This is a string type value which is a data member of the caller RIGHT_TRILATERAL object.
A = POINT(0,0). // A represents a point (which is neither B nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
B = POINT(0,1). // B represents a point (which is neither A nor C nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
C = POINT(1,0). // C represents a point (which is neither A nor B nor D) plotted on a two-dimensional Cartesian grid (such that the X value represents a real whole number position along the horizontal axis of the Cartesian grid while Y represents a real whole number position along the vertical axis of the same Cartesian grid).
a = B.get_distance_from(C) = 1.4142135623730951454746218587388284504413604736328125. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points B and C.
b = C.get_distance_from(A) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points C and A.
c = A.get_distance_from(B) = 1. // The method returns the approximate nonnegative real number of Cartesian grid unit lengths which span the length of the shortest path between points A and B.
slope_of_side_a = B.get_slope_of_line_to(C) = -1. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points B and C.
slope_of_side_b = C.get_slope_of_line_to(A) = 0. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points C and A.
slope_of_side_c = A.get_slope_of_line_to(B) = inf. // The method returns the approximate nonnegative real number which represents the slope of the line which intersects points A and B.
interior_angle_of_A = get_interior_angle_CAB() = 90.00007601981208438246540026739239692687988828125. // The value represents the approximate nonnegative real number angle measurement of the acute or else right angle

formed by the intersection of the line segment whose endpoints are C and A with the line segment whose endpoints are A and B such that those two line segments intersect at A (and the angle measurement is in degrees and not in radians).
interior_angle_of_B = get_interior_angle_ABC() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are A and B with the line segment whose endpoints are B and C such that those two line segments intersect at B (and the angle measurement is in degrees and not in radians).
interior_angle_of_C = get_interior_angle_BCA() = 45.0000380099060208749506273306906223297119140625. // The method returns the approximate nonnegative real number angle measurement of the acute or else right angle formed by the intersection of the line segment whose endpoints are B and C with the line segment whose endpoints are C and A such that those two line segments intersect at C (and the angle measurement is in degrees and not in radians).
interior_angle_of_A + interior_angle_of_B + interior_angle_of_C = 180.0001520396241403432213701307773590087890625. // sum of all three approximate interior angle measurements of the trilateral represented by the caller TRILATERAL object (in degrees and not in radians)
get_perimeter() = a + b + c = 3.41421356237309492343001693370752036571502685546875. // The method returns the sum of the three approximated side lengths of the trilateral which the caller TRILATERAL object represents.
get_area() = 0.4999999999999997779553950749686919152736663818359375. // The method returns the approximate nonnegative real number of Cartesian grid unit squares which are enclosed inside of the two-dimensional region formed by the three line segments which connect points A to B, B to C, and C to A.

End Of Program

This web page was last updated on 07_JULY_2023. The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

[End of abridged plain-text content from POLYGON]

LINKED_LIST

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/linked_list_image.png

---

The C++ program featured in this tutorial web page demonstrates the concept of Object Oriented Programming (OOP). The program implements a user defined data type for instantiating LINKED_LIST type objects. Each LINKED_LIST type object represents a singly-linked list whose elements are NODE type variables. A NODE type variable is a user defined data type for instantiating data structures comprised of two variables: a string type variable named key and a pointer-to-NODE type variable named next. A LINKED_LIST object can execute various functions including the ability to insert NODE type elements to the end of the linked list and the ability to remove all elements from the list whose key values match an input string type value. (Note that, unlike a C++ array, the elements of a C++ linked list are not necessarily homogeneous (i.e. of the same data type). Also, unlike a C++ array, a C++ linked list can change size after it is instantiated. Lastly, unlike a C++ array, the elements of a C++ linked list are not necessarily contiguous in memory).

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

SOFTWARE_APPLICATION_COMPONENTS

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/linked_list.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/linked_list.cpp

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/linked_list_driver.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/linked_list_driver_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ code from the files named linked_list.h, linked_list.cpp, and linked_list_driver.cpp into their own new text editor documents and save those documents using their corresponding file names:

linked_list.h

linked_list.cpp

linked_list_driver.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ linked_list_driver.cpp linked_list.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: Observe program results on the command line terminal and in the output file.

LINKED_LIST_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the LINKED_LIST class.

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/linked_list.h

---

```
/**
 * file: linked_list.h
 * type: C++ (header file)
 * author: karbytes
 * date: 07_JULY_2023
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#ifndef LINKED_LIST_H // If linked_list.h has not already been linked to a source file (.cpp),
#define LINKED_LIST_H // then link this header file to the source file(s) which include this
header file.

/* preprocessing directives */
#include < iostream > // library for defining objects which handle command line input and
command line output
#include < fstream > // library for defining objects which handle file input and file output
#include < string > // library which defines a sequence of text characters (i.e. char type values)
as a string type variable

/**
 * A variable whose data type is NODE is a tuple consisting of two variables
 * such that one of those variables stores some arbitrary piece of information (i.e. key)
 * while the other variable stores the address of a NODE variable (i.e. next).
 *
 * Like a C++ class, a C++ struct is a user defined data type.
 *
 * Note that each of the members of a struct variable is public and neither private nor protected.
 *
 * Note that each of the members of a struct variable is a variable and not a function.
 */
struct NODE
{
	std::string key; // key stores an arbitrary sequence of characters
	NODE * next; // next stores the memory address of a NODE type variable.
};

/**
```

```
 * A variable whose data type is LINKED_LIST is a software object whose data attributes
 * consist of exactly one pointer-to-NODE type variable which is assumed to be the
 * first node of a linear and unidirectional (i.e. singly-linked) linked list.
 *
 * When a LINKED_LIST type variable is declared, a dynamic NODE type variable is created
 * and the memory address of that dynamic NODE type variable is stored in a pointer-to-NODE
 * type variable named head.
 *
 * After a LINKED_LIST type variable is created and before that variable is deleted,
 * NODE type elements can be inserted into the list which the LINKED_LIST type variable
represents
 * and NODE type elements can be removed from the list which the LINKED_LIST type variable
represents.
 *
 * After a LINKED_LIST type variable is created and before that variable is deleted,
 * that variable (i.e. object) can invoke a print function which prints a description
 * of the caller LINKED_LIST object.
 *
 * When a LINKED_LIST variable is deleted, the pointer-to-NODE type variable named head
 * (which was assigned memory during program runtime rather than during program compilation
time)
 * is deleted.
 */
class LINKED_LIST
{
private:
	NODE * head; // head stores the memory address of the first NODE type element of a
LINKED_LIST type data structure.
	bool remove_node_with_key(std::string key); // helper method
public:
	LINKED_LIST(); // constructor
	void insert_node_at_end_of_list(NODE * node); // setter method
	void remove_nodes_with_key(std::string key); // setter method
	int get_number_of_nodes_in_list(); // getter method
	void print(std::ostream & output = std::cout); // descriptor method
	friend std::ostream & operator << (std::ostream & output, LINKED_LIST & linked_list); //
descriptor method
	~LINKED_LIST(); // destructor
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

LINKED_LIST_CLASS_SOURCE_CODE

The following source code defines the functions of the LINKED_LIST class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/linked_list.cpp

---

```
/**
 * file: linked_list_driver.cpp
 * type: C++ (source file)
 * date: 07_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */

#include "linked_list.h" // Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the LINKED_LIST class.

/* function prototypes */
void unit_test_0(std::ostream & output);
void unit_test_1(std::ostream & output);
void unit_test_2(std::ostream & output);
void unit_test_3(std::ostream & output);
void unit_test_4(std::ostream & output);

/**
 * Unit Test # 0: LINKED_LIST constructor, print method, and destructor.
 */
void unit_test_0(std::ostream & output)
{
	output << "\n\n************************************************";
	output << "\nUnit Test # 0: LINKED_LIST constructor, print method, and destructor.";
	output << "\n************************************************";
	output << "\nLINKED_LIST linked_list;";
	output << "\nlinked_list.print(output);";
	LINKED_LIST linked_list;
	linked_list.print(output);
}

/**
```

```
 * Unit Test # 1: LINKED_LIST constructor, insert method, print method, and destructor.
 */
void unit_test_1(std::ostream & output)
{
	output << "\n\n************************************************";
	output << "\nUnit Test # 1: LINKED_LIST constructor, insert method, print method, and 
destructor.";
	output << "\n************************************************";
	output << "\nLINKED_LIST linked_list;";
	output << "\nNODE node;";
	output << "\nnode.key = \"unit_test_1\";";
	output << "\nnode.next = NULL;";
	output << "\nlinked_list.insert_node_at_end_of_list(&node);";
	output << "\nlinked_list.print(output);";
	LINKED_LIST linked_list;
	NODE node;
	node.key = "unit_test_1";
	node.next = NULL;
	linked_list.insert_node_at_end_of_list(&node);
	linked_list.print(output);
}

/**
 * Unit Test # 2: LINKED_LIST constructor, insert method, print method, and destructor.
 */
void unit_test_2(std::ostream & output)
{
	output << "\n\n************************************************";
	output << "\nUnit Test # 2: LINKED_LIST constructor, insert method, print method, and 
destructor.";
	output << "\n************************************************";
	output << "\nLINKED_LIST linked_list;";
	output << "\nNODE node_A = { key : \"node_A\", next : NULL };";
	output << "\nNODE node_B = { key : \"node_B\", next : NULL };";
	output << "\nNODE node_C = { key : \"node_C\", next : NULL };";
	output << "\nlinked_list.insert_node_at_end_of_list(&node_A);";
	output << "\nlinked_list.insert_node_at_end_of_list(&node_B);";
	output << "\nlinked_list.insert_node_at_end_of_list(&node_C);";
	output << "\noutput << linked_list; // functionally identical to linked_list.print(output)";
	LINKED_LIST linked_list;
	NODE node_A = { key : "node_A", next : NULL };
	NODE node_B = { key : "node_B", next : NULL };
	NODE node_C = { key : "node_C", next : NULL };
	linked_list.insert_node_at_end_of_list(&node_A);
```

```
	linked_list.insert_node_at_end_of_list(&node_B);
	linked_list.insert_node_at_end_of_list(&node_C);
	output << linked_list;
}

/**
 * Unit Test # 3: LINKED_LIST constructor, insert method, remove method, print method, and destructor.
 */
void unit_test_3(std::ostream & output)
{
	output << "\n\n*********************************************";
	output << "\nUnit Test # 3: LINKED_LIST constructor, insert method, remove method, print method, and destructor.";
	output << "\n*********************************************";
	output << "\nLINKED_LIST linked_list;";
	output << "\nNODE node_X = { key : \"node_X\", next : NULL };";
	output << "\nNODE node_Y = { key : \"node_Y\", next : NULL };";
	output << "\nNODE node_Z = { key : \"node_Z\", next : NULL };";
	output << "\nlinked_list.insert_node_at_end_of_list(&node_X);";
	output << "\nlinked_list.insert_node_at_end_of_list(&node_Y);";
	output << "\nlinked_list.insert_node_at_end_of_list(&node_Z);";
	output << "\nlinked_list.print(output);";
	output << "\nlinked_list.remove_nodes_with_key(\"node_Y\");";
	output << "\nlinked_list.print(output);";
	LINKED_LIST linked_list;
	NODE node_X = { key : "node_X", next : NULL };
	NODE node_Y = { key : "node_Y", next : NULL };
	NODE node_Z = { key : "node_Z", next : NULL };
	linked_list.insert_node_at_end_of_list(&node_X);
	linked_list.insert_node_at_end_of_list(&node_Y);
	linked_list.insert_node_at_end_of_list(&node_Z);
	linked_list.print(output);
	linked_list.remove_nodes_with_key("node_Y");
	linked_list.print(output);
}

/**
 * Unit Test # 4: LINKED_LIST constructor, insert method, remove method, print method, and destructor.
 */
void unit_test_4(std::ostream & output)
{
	output << "\n\n*********************************************";
```

```
	output << "\nUnit Test # 4: LINKED_LIST constructor, insert method, remove method, print method, and destructor.";
	output << "\n************************************************";
	output << "\nLINKED_LIST linked_list;";
	output << "\nNODE n0 = { key : \"red\", next : NULL };";
	output << "\nNODE n1 = { key : \"blue\", next : NULL };";
	output << "\nNODE n2 = { key : \"green\", next : NULL };";
	output << "\nNODE n3 = { key : \"red\", next : NULL };";
	output << "\nNODE n4 = { key : \"green\", next : NULL };";
	output << "\nNODE n5 = { key : \"red\", next : NULL };";
	output << "\nNODE n6 = { key : \"red\", next : NULL };";
	output << "\nNODE n7 = { key : \"red\", next : NULL };";
	output << "\nlinked_list.insert_node_at_end_of_list(&n0);";
	output << "\nlinked_list.insert_node_at_end_of_list(&n1);";
	output << "\nlinked_list.insert_node_at_end_of_list(&n2);";
	output << "\nlinked_list.insert_node_at_end_of_list(&n3);";
	output << "\nlinked_list.insert_node_at_end_of_list(&n4);";
	output << "\nlinked_list.insert_node_at_end_of_list(&n5);";
	output << "\nlinked_list.insert_node_at_end_of_list(&n6);";
	output << "\nlinked_list.insert_node_at_end_of_list(&n7);";
	output << "\nlinked_list.print(output);";
	output << "\nlinked_list.remove_nodes_with_key(\"red\");";
	output << "\nlinked_list.print(output);";
	output << "\nlinked_list.remove_nodes_with_key(\"green\");";
	output << "\nlinked_list.print(output);";
	output << "\nlinked_list.remove_nodes_with_key(\"blue\");";
	output << "\nlinked_list.print(output);";
	LINKED_LIST linked_list;
	NODE n0 = { key : "red", next : NULL };
	NODE n1 = { key : "blue", next : NULL };
	NODE n2 = { key : "green", next : NULL };
	NODE n3 = { key : "red", next : NULL };
	NODE n4 = { key : "green", next : NULL };
	NODE n5 = { key : "red", next : NULL };
	NODE n6 = { key : "red", next : NULL };
	NODE n7 = { key : "red", next : NULL };
	linked_list.insert_node_at_end_of_list(&n0);
	linked_list.insert_node_at_end_of_list(&n1);
	linked_list.insert_node_at_end_of_list(&n2);
	linked_list.insert_node_at_end_of_list(&n3);
	linked_list.insert_node_at_end_of_list(&n4);
	linked_list.insert_node_at_end_of_list(&n5);
	linked_list.insert_node_at_end_of_list(&n6);
	linked_list.insert_node_at_end_of_list(&n7);
```

```
	linked_list.print(output);
	linked_list.remove_nodes_with_key("red");
	linked_list.print(output);
	linked_list.remove_nodes_with_key("green");
	linked_list.print(output);
	linked_list.remove_nodes_with_key("blue");
	linked_list.print(output);
}

/* program entry point */
int main()
{
	// Declare a file output stream object.
	std::ofstream file;

	/**
	* If linked_list_driver_output.txt does not already exist in the same directory as
linked_list_driver.cpp,
	* create a new file named linked_list_driver_output.txt.
	*
	* Open the plain-text file named linked_list_driver_output.txt
	* and set that file to be overwritten with program data.
	*/
	file.open("linked_list_driver_output.txt");

	// Print an opening message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nStart Of Program";
	std::cout << "\n--------------------------------";

	// Print an opening message to the file output stream.
	file << "--------------------------------";
	file << "\nStart Of Program";
	file << "\n--------------------------------";

	// Implement a series of unit tests which demonstrate the functionality of LINKED_LIST
class variables.
	unit_test_0(std::cout);
	unit_test_0(file);
	unit_test_1(std::cout);
	unit_test_1(file);
	unit_test_2(std::cout);
	unit_test_2(file);
	unit_test_3(std::cout);
```

```
	unit_test_3(file);
	unit_test_4(std::cout);
	unit_test_4(file);

	// Print a closing message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nEnd Of Program";
	std::cout << "\n--------------------------------\n\n";

	// Print a closing message to the file output stream.
	file << "\n\n--------------------------------";
	file << "\nEnd Of Program";
	file << "\n--------------------------------";

	// Close the file output stream.
	file.close();

	// Exit the program.
	return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/linked_list_driver_output.txt

---

```
--------------------------------
Start Of Program
--------------------------------

************************************************
Unit Test # 0: LINKED_LIST constructor, print method, and destructor.
************************************************
LINKED_LIST linked_list;
linked_list.print(output);
```

----
this = 0x7fff55749340. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x7fff55749340. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x5647196a14b0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
	NODE_0 := {
		p := 0x5647196a14b0.
		p -> key = HEAD.
		p -> next = 0.
	}.
}.
```

****

Unit Test # 1: LINKED_LIST constructor, insert method, print method, and destructor.

****

```
LINKED_LIST linked_list;
NODE node;
node.key = "unit_test_1";
node.next = NULL;
linked_list.insert_node_at_end_of_list(&node);
linked_list.print(output);
```

----

this = 0x7fff55749318. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x7fff55749318. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x5647196a14b0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7fff55749320. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 2.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
	NODE_0 := {
		p := 0x5647196a14b0.
		p -> key = HEAD.
		p -> next = 0x7fff55749320.
	}.
	NODE_1 := {
		p := 0x7fff55749320.
		p -> key = unit_test_1.
		p -> next = 0.
	}.
}.
```

****

Unit Test # 2: LINKED_LIST constructor, insert method, print method, and destructor.

****

```
LINKED_LIST linked_list;
NODE node_A = { key : "node_A", next : NULL };
NODE node_B = { key : "node_B", next : NULL };
NODE node_C = { key : "node_C", next : NULL };
```

linked_list.insert_node_at_end_of_list(&node_A);
linked_list.insert_node_at_end_of_list(&node_B);
linked_list.insert_node_at_end_of_list(&node_C);
output < key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7fff557492c0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 4.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
	NODE_0 := {
		p := 0x5647196a14b0.
		p -> key = HEAD.
		p -> next = 0x7fff557492c0.
	}.
	NODE_1 := {
		p := 0x7fff557492c0.
		p -> key = node_A.
		p -> next = 0x7fff557492f0.
	}.
	NODE_2 := {
		p := 0x7fff557492f0.
		p -> key = node_B.
		p -> next = 0x7fff55749320.
	}.
	NODE_3 := {
		p := 0x7fff55749320.
		p -> key = node_C.
		p -> next = 0.
	}.
}.
```

****

Unit Test # 3: LINKED_LIST constructor, insert method, remove method, print method, and destructor.

****

LINKED_LIST linked_list;
NODE node_X = { key : "node_X", next : NULL };
NODE node_Y = { key : "node_Y", next : NULL };
NODE node_Z = { key : "node_Z", next : NULL };
linked_list.insert_node_at_end_of_list(&node_X);
linked_list.insert_node_at_end_of_list(&node_Y);
linked_list.insert_node_at_end_of_list(&node_Z);

linked_list.print(output);
linked_list.remove_nodes_with_key("node_Y");
linked_list.print(output);

----

this = 0x7fff55749298. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x7fff55749298. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x5647196a14b0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7fff557492c0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 4.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
	NODE_0 := {
		p := 0x5647196a14b0.
		p -> key = HEAD.
		p -> next = 0x7fff557492c0.
	}.
	NODE_1 := {
		p := 0x7fff557492c0.
		p -> key = node_X.
		p -> next = 0x7fff557492f0.
	}.
	NODE_2 := {
		p := 0x7fff557492f0.
		p -> key = node_Y.
		p -> next = 0x7fff55749320.
	}.
```

NODE_3 := {
p := 0x7fff55749320.
p -> key = node_Z.
p -> next = 0.
}.
}.

----
this = 0x7fff55749298. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x7fff55749298. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x5647196a14b0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7fff557492c0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 3.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
NODE_0 := {
p := 0x5647196a14b0.
p -> key = HEAD.
p -> next = 0x7fff557492c0.
}.
NODE_1 := {
p := 0x7fff557492c0.
p -> key = node_X.
p -> next = 0x7fff55749320.
}.
NODE_2 := {

```
                    p := 0x7fff55749320.
                    p -> key = node_Z.
                    p -> next = 0.
          }.
}.
```

****

Unit Test # 4: LINKED_LIST constructor, insert method, remove method, print method, and destructor.

****

```
LINKED_LIST linked_list;
NODE n0 = { key : "red", next : NULL };
NODE n1 = { key : "blue", next : NULL };
NODE n2 = { key : "green", next : NULL };
NODE n3 = { key : "red", next : NULL };
NODE n4 = { key : "green", next : NULL };
NODE n5 = { key : "red", next : NULL };
NODE n6 = { key : "red", next : NULL };
NODE n7 = { key : "red", next : NULL };
linked_list.insert_node_at_end_of_list(&n0);
linked_list.insert_node_at_end_of_list(&n1);
linked_list.insert_node_at_end_of_list(&n2);
linked_list.insert_node_at_end_of_list(&n3);
linked_list.insert_node_at_end_of_list(&n4);
linked_list.insert_node_at_end_of_list(&n5);
linked_list.insert_node_at_end_of_list(&n6);
linked_list.insert_node_at_end_of_list(&n7);
linked_list.print(output);
linked_list.remove_nodes_with_key("red");
linked_list.print(output);
linked_list.remove_nodes_with_key("green");
linked_list.print(output);
linked_list.remove_nodes_with_key("blue");
linked_list.print(output);
```

----

this = 0x7fff557491a8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.

&head = 0x7fff557491a8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.

sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x5647196a14b0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7fff557491d0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 9.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
	NODE_0 := {
		p := 0x5647196a14b0.
		p -> key = HEAD.
		p -> next = 0x7fff557491d0.
	}.
	NODE_1 := {
		p := 0x7fff557491d0.
		p -> key = red.
		p -> next = 0x7fff55749200.
	}.
	NODE_2 := {
		p := 0x7fff55749200.
		p -> key = blue.
		p -> next = 0x7fff55749230.
	}.
	NODE_3 := {
		p := 0x7fff55749230.
		p -> key = green.
		p -> next = 0x7fff55749260.
	}.
	NODE_4 := {
		p := 0x7fff55749260.
		p -> key = red.
		p -> next = 0x7fff55749290.
	}.
	NODE_5 := {
```

p := 0x7fff55749290.
p -> key = green.
p -> next = 0x7fff557492c0.
}.
NODE_6 := {
p := 0x7fff557492c0.
p -> key = red.
p -> next = 0x7fff557492f0.
}.
NODE_7 := {
p := 0x7fff557492f0.
p -> key = red.
p -> next = 0x7fff55749320.
}.
NODE_8 := {
p := 0x7fff55749320.
p -> key = red.
p -> next = 0.
}.
}.

----
this = 0x7fff557491a8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x7fff557491a8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x5647196a14b0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7fff55749200. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.

get_number_of_nodes_in_list() = 4.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
NODE_0 := {
p := 0x5647196a14b0.
p -> key = HEAD.
p -> next = 0x7fff55749200.
}.
NODE_1 := {
p := 0x7fff55749200.
p -> key = blue.
p -> next = 0x7fff55749230.
}.
NODE_2 := {
p := 0x7fff55749230.
p -> key = green.
p -> next = 0x7fff55749290.
}.
NODE_3 := {
p := 0x7fff55749290.
p -> key = green.
p -> next = 0.
}.
}.

----
this = 0x7fff557491a8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x7fff557491a8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x5647196a14b0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).

head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7fff55749200. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 2.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
	NODE_0 := {
		p := 0x5647196a14b0.
		p -> key = HEAD.
		p -> next = 0x7fff55749200.
	}.
	NODE_1 := {
		p := 0x7fff55749200.
		p -> key = blue.
		p -> next = 0.
	}.
}.
```

----
this = 0x7fff557491a8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x7fff557491a8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x5647196a14b0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
          NODE_0 := {
                    p := 0x5647196a14b0.
                    p -> key = HEAD.
                    p -> next = 0.
          }.
}.
--------------------------------------------------------------------------------------------------
```

End Of Program

This web page was last updated on 08_JULY_2023. The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

[End of abridged plain-text content from LINKED_LIST]

HASH_TABLE

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/hash_table_image.png

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/linked_list_image.png

---

The C++ program featured in this tutorial web page demonstrates the concept of Object Oriented Programming (OOP). The program implements a user defined data type for instantiating HASH_TABLE type objects. Each HASH_TABLE type object represents an array whose elements are LINKED_LIST type objects (and each LINKED_LIST object represents a singly-linked list whose elements are NODE type struct variables). A NODE can be inserted into the HASH_TABLE using a hash function which takes that NODE's key value as the function input and returns an index of the HASH_TABLE array in which to store that NODE as the last element of the LINKED_LIST which is located at that particular array index.

To view hidden text inside each of the preformatted text boxes below, scroll horizontally.

---

array_length := HASH_TABLE.N. // array_length is a nonnegative integer.
array_index := HASH_TABLE.hash(NODE.key). // array_index is a nonnegative integer which is smaller than array_length.

SOFTWARE_APPLICATION_COMPONENTS

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/public_linked_list.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/public_linked_list.cpp

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/hash_table.h

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/hash_table.cpp

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/hash_table_driver.cpp

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/hash_table_driver_output.txt

---

PROGRAM_COMPILATION_AND_EXECUTION

STEP_0: Copy and paste the C++ code from the files named public_linked_list.h, public_linked_list.cpp, hash_table.h, hash_table.cpp, and hash_table_driver.cpp into their own new text editor documents and save those documents using their corresponding file names:

public_linked_list.h

public_linked_list.cpp

hash_table.h

hash_table.cpp

hash_table_driver.cpp

STEP_1: Open a Unix command line terminal application and set the current directory to wherever the C++ is located on the local machine (e.g. Desktop).

cd Desktop

STEP_2: Compile the C++ file into machine-executable instructions (i.e. object file) and then into an executable piece of software named app using the following command:

g++ hash_table_driver.cpp hash_table.cpp public_linked_list.cpp -o app

STEP_3: If the program compilation command does not work, then use the following command to install the C++ compiler:

sudo apt install build-essential

STEP_4: Observe program results on the command line terminal and in the output file.

LINKED_LIST_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the LINKED_LIST class.

When copy-pasting the source code from the preformatted text box below into a text editor document, remove the spaces between the angle brackets and the library names in the preprocessing directives code block. (The spaces were inserted between the library names and angle brackets in the preformatted text box below in order to prevent the WordPress server from misinterpreting those C++ library references as HTML tags in the source code of this web page).

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/public_linked_list.h

---

```
/**
 * file: public_linked_list.h
 * type: C++ (header file)
 * author: karbytes
 * date: 08_JULY_2023
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#ifndef LINKED_LIST_H // If public_linked_list.h has not already been linked to a source file
(.cpp),
#define LINKED_LIST_H // then link this header file to the source file(s) which include this
header file.

/* preprocessing directives */
#include < iostream > // library for defining objects which handle command line input and
command line output
#include < fstream > // library for defining objects which handle file input and file output
#include < string > // library which defines a sequence of text characters (i.e. char type values)
as a string type variable

/**
 * A variable whose data type is NODE is a tuple consisting of two variables
 * such that one of those variables stores some arbitrary piece of information (i.e. key)
 * while the other variable stores the address of a NODE variable (i.e. next).
 *
 * Like a C++ class, a C++ struct is a user defined data type.
 *
 * Note that each of the members of a struct variable is public and neither private nor protected.
 *
 * Note that each of the members of a struct variable is a variable and not a function.
 */
struct NODE
{
	std::string key; // key stores an arbitrary sequence of characters
	NODE * next; // next stores the memory address of a NODE type variable.
};
```

```
/**
 * A variable whose data type is LINKED_LIST is a software object whose data attributes
 * consist of exactly one pointer-to-NODE type variable which is assumed to be the
 * first node of a linear and unidirectional (i.e. singly-linked) linked list.
 *
 * When a LINKED_LIST type variable is declared, a dynamic NODE type variable is created
 * and the memory address of that dynamic NODE type variable is stored in a pointer-to-NODE
 * type variable named head.
 *
 * After a LINKED_LIST type variable is created and before that variable is deleted,
 * NODE type elements can be inserted into the list which the LINKED_LIST type variable
represents
 * and NODE type elements can be removed from the list which the LINKED_LIST type variable
represents.
 *
 * After a LINKED_LIST type variable is created and before that variable is deleted,
 * that variable (i.e. object) can invoke a print function which prints a description
 * of the caller LINKED_LIST object.
 *
 * When a LINKED_LIST variable is deleted, the pointer-to-NODE type variable named head
 * (which was assigned memory during program runtime rather than during program compilation
time)
 * is deleted.
 */
class LINKED_LIST
{
public:
	NODE * head; // head stores the memory address of the first NODE type element of a
LINKED_LIST type data structure.
	bool remove_node_with_key(std::string key); // helper method
	LINKED_LIST(); // constructor
	void insert_node_at_end_of_list(NODE * node); // setter method
	void remove_nodes_with_key(std::string key); // setter method
	int get_number_of_nodes_in_list(); // getter method
	void print(std::ostream & output = std::cout); // descriptor method
	friend std::ostream & operator << (std::ostream & output, LINKED_LIST & linked_list); //
descriptor method
	~LINKED_LIST(); // destructor
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

LINKED_LIST_CLASS_SOURCE_CODE

The following source code defines the functions of the LINKED_LIST class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/public_linked_list.cpp

---

```
/**
 * file: public_linked_list.cpp
 * type: C++ (source file)
 * author: karbytes
 * date: 08_JULY_2023
 * license: PUBLIC_DOMAIN
 */

// Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the LINKED_LIST class.
#include "public_linked_list.h"

/**
 * Starting at the NODE type element whose memory address is stored in head and ending at NULL,
 * traverse the singly-linked list comprised of NODE type elements such that each of those elements are visited.
 *
 * If the current element's key is identical to the input key,
 * set the next property of the previous node to
 * the memory address of the node which is next in the list after the current element
 * and return true.
 *
 * If at least one node in the list has a key which is identical to the input key,
 * the function will return true. Otherwise, the function will return false.
 *
 * (The method of using two pointers, p and q, to traverse the list is colloquially described as "inchworming"
 * because those two pointers metaphorically resemble opposite ends of an inchworm as that insect stretches its
 * front end forward by approximately one inch before moving its back end to where its front end is located).
```

```
 */
bool LINKED_LIST::remove_node_with_key(std::string key)
{
    NODE * p = head;
    NODE * q = head;
    while (q)
    {
        if ((q -> key == key) && (q != head))
        {
            std::cout << "\n\nThe NODE whose memory address is " << q << " is being removed
from the LINKED_LIST...";
            p -> next = q -> next;
            return true;
        }
        p = q;
        q = p -> next;
    }
    return false;
}

/**
 * Instantiate an "empty" linked list (i.e. a linked list with only a head node and no "body nodes").
 *
 * Note that only "body nodes" may be inserted into or else removed from a linked list which a
LINKED_LIST type object represents.
 */
LINKED_LIST::LINKED_LIST()
{
    std::cout << "\n\nCreating the LINKED_LIST type object whose memory address is " << this
<< "...";
    head = new NODE;
    head -> key = "HEAD";
    head -> next = NULL;
}

/**
 * Make the input NODE type variable the last element of the linked list represented by the caller
LINKED_LIST object.
 *
 * (Note that using a NODE which is currently an element of the caller LINKED_LIST as the
input to this function
 * will turn the linked list represented by the caller LINKED_LIST object into a closed loop rather
than a finite linear sequence).
 *
```

```
 * (The method of using two pointers, p and q, to traverse the list is colloquially described as
"inchworming"
 * because those two pointers metaphorically resemble opposite ends of an inchworm as that
insect stretches its
 * front end forward by approximately one inch before moving its back end to where its front end
is located).
 */
void LINKED_LIST::insert_node_at_end_of_list(NODE * node)
{
   NODE * p = head;
   NODE * q = head;
   while (q)
   {
      p = q;
      q = p -> next;
   }
   std::cout << "\n\nThe NODE whose memory address is " << p << " is being inserted into the
LINKED_LIST as the last element of that list...";
   p -> next = node;
   node -> next = NULL;
}

/**
 * Remove all nodes from the linked list represented by the caller LINKED_LIST object
 * whose key values are identical to the input key value.
 */
void LINKED_LIST::remove_nodes_with_key(std::string key)
{
   bool at_least_one_node_was_removed = false;
   at_least_one_node_was_removed = remove_node_with_key(key);
   while (at_least_one_node_was_removed) at_least_one_node_was_removed =
remove_node_with_key(key);
}

/**
 * Return the natural number count of NODE type elements inside the singly-linked list which
 * the caller LINKED_LIST object represents.
 *
 * Starting with the head and ending with NULL,
 * traverse sequentially down the list of NODE type elements and
 * count each element in the exist.
 *
 * If the linked list is "empty" (i.e. the head is the only NODE in the caller LINKED_LIST),
 * one will be returned.
```

```
 *
 * (The method of using two pointers, p and q, to traverse the list is colloquially described as "inchworming"
 * because those two pointers metaphorically resemble opposite ends of an inchworm as that insect stretches its
 * front end forward by approximately one inch before moving its back end to where its front end is located).
 */
int LINKED_LIST::get_number_of_nodes_in_list()
{
    int node_count = 0;
    NODE * p = head;
    NODE * q = head;
    while (q)
    {
        p = q;
        q = p -> next;
        node_count += 1;
    }
    return node_count;
}

/**
 * The print method of the LINKED_LIST class prints a description of the caller LINKED_LIST object to the output stream.
 *
 * A description of each NODE type element of the linked list which the caller LINKED_LIST object represents will be printed to the output stream
 * in the order those elements were inserted into the list by "inchworming" from NODE_0 to NODE_N (where N is the total number of nodes in the list).
 *
 * (The method of using two pointers, p and q, to traverse the list is colloquially described as "inchworming"
 * because those two pointers metaphorically resemble opposite ends of an inchworm as that insect stretches its
 * front end forward by approximately one inch before moving its back end to where its front end is located).
 *
 * Note that the default value of the function input parameter is the standard command line output stream (std::cout).
 *
 * The default parameter is defined in the LINKED_LIST class header file (i.e. public_linked_list.h) and not in the LINKED_LIST class source file (i.e. public_linked_list.cpp).
 */
```

```
void LINKED_LIST::print(std::ostream & output)
{
   int node_count = 0;
   NODE * p = head;
   NODE * q = head;
   output << "\n\n----------------------------------------------------------------------------------------------------";
   output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.";
   output << "\n&head = " << &head << ". // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.";
   output << "\nsizeof(NODE) = " << sizeof(NODE) << ". // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).";
   output << "\nsizeof(std::string) = " << sizeof(std::string) << ". // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).";
   output << "\nsizeof(NODE *) = " << sizeof(NODE *) << ". // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).";
   output << "\nsizeof(LINKED_LIST) = " << sizeof(LINKED_LIST) << ". // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).";
   output << "\nhead = " << head << ". // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).";
   output << "\nhead -> key = " << head -> key << ". // The arrow operator returns the string type property named key of the NODE type variable which head points to.";
   output << "\nhead -> next = " << head -> next << ". // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.";
   output << "\nget_number_of_nodes_in_list() = " << get_number_of_nodes_in_list() << ".";
   output << "\n// p is a pointer to a NODE type variable.";
   output << "\nLINKED_LIST := {";
   while (q)
   {
      p = q;
      output << "\n\tNODE_" << node_count << " := {";
      output << "\n\t\tp := " << p << ".";
      output << "\n\t\tp -> key = " << p -> key << ".";
      output << "\n\t\tp -> next = " << p -> next << ".";
      output << "\n\t}.";
      q = p -> next;
      node_count += 1;
```

```
    }
    output << "\n}.";
    output << "\n----------------------------------------------------------------------------------------------------";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (<<).
 *
 * (Overloading an operator is assigning a different function to a native operator other than the
function which that operator is used to represent by default).
 *
 * Note that the default value of the leftmost function input parameter is the standard command
line output stream (std::cout).
 * The default parameter is defined in the LINKED_LIST class header file (i.e.
public_linked_list.h).
 *
 * The friend function is not a member of the LINKED_LIST class,
 * but the friend function has access to the private and protected members
 * of the LINKED_LIST class and not just to the public members of the LINKED_LIST class.
 *
 * The friend keyword only prefaces the function prototype of this function
 * (and the prototype of this function is declared in the LINKED_LIST class header file (i.e.
public_linked_list.h)).
 *
 * The friend keyword does not preface the definition of this function
 * (and the definition of this function is specified in the LINKED_LIST class source file (i.e.
public_linked_list.cpp)).
 *
 * // overloaded print function example one
 * LINKED_LIST linked_list_0;
 * std::cout << linked_list_0; // identical to linked_list_0.print();
 *
 * // overloaded print function example two
 * std::ofstream file;
 * LINKED_LIST linked_list_1;
 * file << linked_list_1; // identical to linked_list_1.print(file);
 */
std::ostream & operator << (std::ostream & output, LINKED_LIST & linked_list)
{
    linked_list.print(output);
    return output;
}
```

```
/**
 * The destructor method of the LINKED_LIST class de-allocates memory which was used to
 * instantiate the LINKED_LIST object which is calling this function.
 *
 * The destructor method of the LINKED_LIST class is automatically called when
 * the program scope in which the caller LINKED_LIST object was instantiated terminates.
 */
LINKED_LIST::~LINKED_LIST()
{
   std::cout << "\n\nDeleting the LINKED_LIST type object whose memory address is " << this
<< "...";
   delete head;
}
```

HASH_TABLE_CLASS_HEADER

The following header file contains the preprocessing directives and function prototypes of the HASH_TABLE class.

C++_header_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/hash_table.h

---

```
/**
 * file: hash_table.h
 * type: C++ (header file)
 * author: karbytes
 * date: 08_JULY_2023
 * license: PUBLIC_DOMAIN
 */

/* preprocessing directives */
#ifndef HASH_TABLE_H // If hash_table.h has not already been linked to a source file (.cpp),
#define HASH_TABLE_H // then link this header file to the source file(s) which include this
header file.

/* preprocessing directives */
#include "public_linked_list.h" // Include the C++ header file which contains preprocessing
directives, variable declarations, and function prototypes for the LINKED_LIST class.
#define MAXIMUM_N 100 // constant which represents maximum N value
```

```
/**
 * A variable whose data type is HASH_TABLE is a software object whose data attributes
 * consist of exactly one pointer-to-LINKED_LIST type variable named array
 * and exactly one int type variable named N.
 *
 * N stores a nonnegative integer value no larger than MAXIMUM_N.
 *
 * array stores the memory address of the first memory cell of a LINKED_LIST sized chunk of
contiguous memory cells
 * (and array is a pointer to the first element of an array comprised of N LINKED_LIST type
elements).
 *
 * When a HASH_TABLE type variable is declared, a dynamic LINKED_LIST type variable is
created
 * and the memory address of that dynamic LINKED_LIST type variable is stored in a
pointer-to-LINKED_LIST
 * type variable named array.
 *
 * After a HASH_TABLE type variable is created and before that variable is deleted,
 * NODE type elements can be inserted into the hash table array and
 * NODE type elements can be removed from the hash table array.
 *
 * When a NODE is inserted into the hash table array, a hash function takes that NODE's key as
 * a hash function input and then the hash function outputs a corresponding nonnegative integer
 * no larger than (N - 1) which represents the index of the hash table array where that
 * NODE will be inserted (and that NODE will be appended to the end of the LINKED_LIST
which is stored at
 * that particular array index).
 *
 * After a HASH_TABLE type variable is created and before that variable is deleted,
 * that variable (i.e. object) can invoke a print function which prints a description
 * of the caller HASH_TABLE object.
 *
 * When a HASH_TABLE variable is deleted, the pointer-to-LINKED_LIST type variable named
array
 * (which was assigned memory during program runtime rather than during program compilation
time)
 * and every head property of every LINKED_LIST type element of that array
 * is deleted.
 */
class HASH_TABLE
{
private:
```

```
    // array stores the memory address of the first element of an array of N LINKED_LIST type elements.
    LINKED_LIST * array;

    // N stores a nonnegative integer value no larger than MAXIMUM_N.
    int N;

    // The hash function returns an array index which corresponds with the input key value.
    int hash(std::string key);

public:

    // The default constructor sets the length of the hash table array to 10 by default.
    HASH_TABLE(int hash_table_length = 10);

    // The setter method appends the input NODE to the end of the LINKED_LIST located at array[hash(node -> key)].
    void insert_node(NODE * node);

    // The setter method removes all NODE type instances from the hash table array whose key values match the input key value.
    void remove_nodes_with_key(std::string key);

    // The getter method returns a singly-linked list of all NODE type instances in the hash table array whose key values match the input key value.
    LINKED_LIST get_nodes_with_key(std::string key);

    // The getter method returns the number of LINKED_LIST type values stored in the hash table array (and the value returned is N).
    int get_number_of_linked_lists_in_hash_table();

    // The getter method returns the total number of NODE type values stored in the hash table array (and the value returned is an integer which is equal to or larger than N).
    int get_number_of_nodes_in_hash_table();

    // The descriptor method prints a description of the caller HASH_TABLE object to the output stream (and the command line terminal is the default output stream parameter).
    void print(std::ostream & output = std::cout);

    // The descriptor method overloads the ostream operator to make it identical to calling the HASH_TABLE print function.
    friend std::ostream & operator << (std::ostream & output, HASH_TABLE & hash_table);
```

```
    // The destructor de-allocates memory which was assigned to the caller HASH_TABLE object.
    ~HASH_TABLE();
};

/* preprocessing directives */
#endif // Terminate the conditional preprocessing directives code block in this header file.
```

HASH_TABLE_CLASS_SOURCE_CODE

The following source code defines the functions of the HASH_TABLE class.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/hash_table.cpp

---

```
/**
 * file: hash_table.cpp
 * type: C++ (source file)
 * author: karbytes
 * date: 08_JULY_2023
 * license: PUBLIC_DOMAIN
 */

// Include the C++ header file which contains preprocessing directives, variable declarations,
and function prototypes for the HASH_TABLE class.
#include "hash_table.h"

/**
 * The hash function returns an array index which corresponds with the input key value.
 */
int HASH_TABLE::hash(std::string key)
{
        int value = 0, i = 0;
        for (i = 0; i < key.length(); i += 1) value += int(key[i]);
        return value % N;
}

/**
 * The default constructor sets the length of the hash table array to 10 by default.
 *
```

```
 * The function returns a HASH_TABLE type object.
 */
HASH_TABLE::HASH_TABLE(int hash_table_length)
{
	std::cout << "\n\nCreating the HASH_TABLE type object whose memory address is " << 
this << "...";
	N = ((hash_table_length < 1) || (hash_table_length > MAXIMUM_N)) ? 10 : 
hash_table_length;
	array = new LINKED_LIST[N];
}

/**
 * The setter method appends the input NODE to the end of the LINKED_LIST located at 
array[hash(node -> key)].
 */
void HASH_TABLE::insert_node(NODE * node)
{
	int index = hash(node -> key);
	array[index].insert_node_at_end_of_list(node);
}

/**
 * The setter method removes all NODE type instances from the hash table array whose key 
values match the input key value.
 */
void HASH_TABLE::remove_nodes_with_key(std::string key)
{
	int index = hash(key);
	return array[index].remove_nodes_with_key(key);
}

/**
 * The getter method returns a singly-linked list of all NODE type instances in the hash table 
array whose key values match the input key value.
 *
 * Set the next pointer value of the final NODE element in the returned LINKED_LIST to NULL.
 */
LINKED_LIST HASH_TABLE::get_nodes_with_key(std::string key)
{
	int index = hash(key);
	LINKED_LIST search_results;
	NODE * p = array[index].head;
	NODE * q = array[index].head;
	while (q)
```

```
	{
	if ((p -> key == key) && (p != array[index].head))
	{
	search_results.insert_node_at_end_of_list(p);
	p -> next = NULL;
	}
	p = q;
	q = p -> next;
	}

	return search_results;
}

/**
 * The getter method returns the number of LINKED_LIST type values stored in the hash table
array (and the value returned is N).
 */
int HASH_TABLE::get_number_of_linked_lists_in_hash_table()
{
	return N;
}

/**
 * The getter method returns the total number of NODE type values stored in the hash table
array (and the value returned is an integer which is equal to or larger than N).
 */
int HASH_TABLE::get_number_of_nodes_in_hash_table()
{
	int node_count = 0, i = 0;
	for (i = 0; i < N; i += 1) node_count += array[i].get_number_of_nodes_in_list();
	return node_count;
}

/**
 * The descriptor method prints a description of the caller HASH_TABLE object to the output
stream (and the command line terminal is the default output stream parameter).
 */
void HASH_TABLE::print(std::ostream & output)
{
	int i = 0;
	output <<
"\n\n--------------------------------------------------------------------------------------------------";
```

output << "\nthis = " << this << ". // The keyword named this is a pointer which stores the memory address of the first memory cell of a HASH_TABLE sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE object.";

output << "\n&array = " << &array << ". // The reference operation returns the memory address of the first memory cell of a pointer-to-LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE data attribute named array.";

output << "\n&N = " << &N << ". // The reference operation returns the memory address of the first memory cell of a pointer-to-int sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE data attribute named N.";

output << "\nsizeof(int) = " << sizeof(int) << ". // The sizeof() operation returns the number of bytes of memory which a int type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(int *) = " << sizeof(int *) << ". // The sizeof() operation returns the number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string) = " << sizeof(std::string) << ". // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(std::string *) = " << sizeof(std::string *) << ". // The sizeof() operation returns the number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(NODE) = " << sizeof(NODE) << ". // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(NODE *) = " << sizeof(NODE *) << ". // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(LINKED_LIST) = " << sizeof(LINKED_LIST) << ". // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(LINKED_LIST *) = " << sizeof(LINKED_LIST *) << ". // The sizeof() operation returns the number of bytes of memory which a pointer-to-LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(HASH_TABLE) = " << sizeof(HASH_TABLE) << ". // The sizeof() operation returns the number of bytes of memory which a HASH_TABLE type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\nsizeof(HASH_TABLE *) = " << sizeof(HASH_TABLE *) << ". // The sizeof() operation returns the number of bytes of memory which a pointer-to-HASH_TABLE type variable occupies. (Each memory cell has a data capacity of 1 byte).";

output << "\narray = " << array << ". // array stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a LINKED_LIST type variable or else array stores NULL (and the value NULL is displayed as 0).";

output << "\nN = " << N << ". // N stores the total number of LINKED_LIST types elements which are represented by the array property of the caller HASH_TABLE object.";

```
	output << "\nHASH_TABLE := {";
	for (i = 0; i < N; i += 1)
	{
	output << 
"\n\n################################################################";
	output << "\narray[" << i << "] := {";
	output << array[i];
	output << "\n}.";
	output << 
"\n################################################################";
	}
	output << "\n}.";
	output << "\n----------------------------------------------------------------------------------------------------------";
}

/**
 * The friend function is an alternative to the print method.
 * The friend function overloads the ostream operator (<<).
 *
 * (Overloading an operator is assigning a different function to a native operator other than the 
function which that operator is used to represent by default).
 *
 * Note that the default value of the leftmost function input parameter is the standard command 
line output stream (std::cout).
 * The default parameter is defined in the HASH_TABLE class header file (i.e. hash_table.h).
 *
 * The friend function is not a member of the HASH_TABLE class,
 * but the friend function has access to the private and protected members
 * of the HASH_TABLE class and not just to the public members of the HASH_TABLE class.
 *
 * The friend keyword only prefaces the function prototype of this function
 * (and the prototype of this function is declared in the HASH_TABLE class header file (i.e. 
hash_table.h)).
 *
 * The friend keyword does not preface the definition of this function
 * (and the definition of this function is specified in the HASH_TABLE class source file (i.e. 
hash_table.cpp)).
 *
 * // overloaded print function example one
 * HASH_TABLE hash_table_0;
 * std::cout << hash_table_0; // identical to hash_table_0.print();
 *
 * // overloaded print function example two
 * std::ofstream file;
```

```
 * HASH_TABLE hash_table_1;
 * file << hash_table_1; // identical to hash_table_1.print(file);
 */
std::ostream & operator << (std::ostream & output, HASH_TABLE & hash_table)
{
	hash_table.print(output);
	return output;
}

/**
 * The destructor method of the HASH_TABLE class de-allocates memory which was used to
 * instantiate the HASH_TABLE object which is calling this function.
 *
 * The destructor method of the HASH_TABLE class is automatically called when
 * the program scope in which the caller HABLE_TABLE object was instantiated terminates.
 */
HASH_TABLE::~HASH_TABLE()
{
	std::cout << "\n\nDeleting the HASH_TABLE type object whose memory address is " <<
this << "...";
	delete [] array;
}
```

PROGRAM_SOURCE_CODE

The following source code defines the client which implements the HASH_TABLE class. The client executes a series of unit tests which demonstrate how the HASH_TABLE class methods work.

C++_source_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/hash_table_driver.cpp

---

```
/**
 * file: hash_table_driver.cpp
 * type: C++ (source file)
 * date: 08_JULY_2023
 * author: karbytes
 * license: PUBLIC_DOMAIN
 */
```

```
#include "hash_table.h" // Include the C++ header file which contains preprocessing directives, variable declarations, and function prototypes for the HASH_TABLE class.

/* function prototypes */
void unit_test_0(std::ostream & output);
void unit_test_1(std::ostream & output);
void unit_test_2(std::ostream & output);
void unit_test_3(std::ostream & output);
void unit_test_4(std::ostream & output);
void unit_test_5(std::ostream & output);

/**
 * Unit Test # 0: HASH_TABLE constructor, print method, and destructor.
 */
void unit_test_0(std::ostream & output)
{
	output << "\n\n************************************************";
	output << "\nUnit Test # 0: HASH_TABLE constructor, print method, and destructor.";
	output << "\n************************************************";
	output << "\nHASH_TABLE hash_table;";
	output << "\nhash_table.print(output);";
	HASH_TABLE hash_table;
	hash_table.print(output);
}

/**
 * Unit Test # 1: HASH_TABLE constructor, insert method, print method, and destructor.
 */
void unit_test_1(std::ostream & output)
{
	output << "\n\n************************************************";
	output << "\nUnit Test # 1: HASH_TABLE constructor, insert method, print method, and destructor.";
	output << "\n************************************************";
	output << "\nHASH_TABLE hash_table;";
	output << "\nNODE node;";
	output << "\nnode.key = \"unit_test_1\";";
	output << "\nnode.next = NULL;";
	output << "\nhash_table.insert_node(&node);";
	output << "\nhash_table.print(output);";
	HASH_TABLE hash_table;
	NODE node;
	node.key = "unit_test_1";
```

```
	node.next = NULL;
	hash_table.insert_node(&node);
	hash_table.print(output);
}

/**
 * Unit Test # 2: HASH_TABLE constructor, insert method, print method, and destructor.
 */
void unit_test_2(std::ostream & output)
{
	output << "\n\n************************************************";
	output << "\nUnit Test # 2: HASH_TABLE constructor, insert method, print method, and
destructor.";
	output << "\n************************************************";
	output << "\nHASH_TABLE hash_table;";
	output << "\nNODE node_A = { key : \"node_A\", next : NULL };";
	output << "\nNODE node_B = { key : \"node_B\", next : NULL };";
	output << "\nNODE node_C = { key : \"node_C\", next : NULL };";
	output << "\nhash_table.insert_node(&node_A);";
	output << "\nhash_table.insert_node(&node_B);";
	output << "\nhash_table.insert_node(&node_C);";
	output << "\noutput << hash_table; // functionally identical to hash_table.print(output)";
	HASH_TABLE hash_table;
	NODE node_A = { key : "node_A", next : NULL };
	NODE node_B = { key : "node_B", next : NULL };
	NODE node_C = { key : "node_C", next : NULL };
	hash_table.insert_node(&node_A);
	hash_table.insert_node(&node_B);
	hash_table.insert_node(&node_C);
	output << hash_table; // functionally identical to hash_table.print(output);
}

/**
 * Unit Test # 3: HASH_TABLE constructor, insert method, number of linked lists method,
number of node method, print method, and destructor.
 */
void unit_test_3(std::ostream & output)
{
	output << "\n\n************************************************";
	output << "\nUnit Test # 3: HASH_TABLE constructor, insert method, number of linked
lists method, number of node method, print method, and destructor.";
	output << "\n************************************************";
	output << "\nHASH_TABLE hash_table = HASH_TABLE(5);";
	output << "\nNODE node_A = { key : \"node_A\", next : NULL };";
```

```
	output << "\nNODE node_B = { key : \"node_B\", next : NULL };";
	output << "\nNODE node_C = { key : \"node_C\", next : NULL };";
	output << "\nNODE node_AA = { key : \"node_AA\", next : NULL };";
	output << "\nNODE node_BB = { key : \"node_BB\", next : NULL };";
	output << "\nNODE node_CC = { key : \"node_CC\", next : NULL };";
	output << "\nNODE node_Z = { key : \"node_Z\", next : NULL };";
	output << "\nNODE node_666 = { key : \"node_666\", next : NULL };";
	output << "\nhash_table.insert_node(&node_A);";
	output << "\nhash_table.insert_node(&node_B);";
	output << "\nhash_table.insert_node(&node_C);";
	output << "\nhash_table.insert_node(&node_AA);";
	output << "\nhash_table.insert_node(&node_BB);";
	output << "\nhash_table.insert_node(&node_CC);";
	output << "\nhash_table.insert_node(&node_Z);";
	output << "\nhash_table.insert_node(&node_666);";
	output << "\noutput << hash_table; // functionally identical to hash_table.print(output)";
	HASH_TABLE hash_table = HASH_TABLE(5);
	NODE node_A = { key : "node_A", next : NULL };
	NODE node_B = { key : "node_B", next : NULL };
	NODE node_C = { key : "node_C", next : NULL };
	NODE node_AA = { key : "node_AA", next : NULL };
	NODE node_BB = { key : "node_BB", next : NULL };
	NODE node_CC = { key : "node_CC", next : NULL };
	NODE node_Z = { key : "node_Z", next : NULL };
	NODE node_666 = { key : "node_666", next : NULL };
	hash_table.insert_node(&node_A);
	hash_table.insert_node(&node_B);
	hash_table.insert_node(&node_C);
	hash_table.insert_node(&node_AA);
	hash_table.insert_node(&node_BB);
	hash_table.insert_node(&node_CC);
	hash_table.insert_node(&node_Z);
	hash_table.insert_node(&node_666);
	output << "\nhash_table.get_number_of_linked_lists_in_hash_table() = " <<
hash_table.get_number_of_linked_lists_in_hash_table() << ".";
	output << "\nhash_table.get_number_of_nodes_in_hash_table()= " <<
hash_table.get_number_of_nodes_in_hash_table() << ".";
	output << hash_table; // functionally identical to hash_table.print(output);
}

/**
 * Unit Test # 4: HASH_TABLE constructor, insert method, remove method, print method, and
destructor.
 */
```

```
void unit_test_4(std::ostream & output)
{
	output << "\n\n********************************************";
	output << "\nUnit Test # 4: HASH_TABLE constructor, insert method, remove method,
print method, and destructor.";
	output << "\n********************************************";
	output << "\nHASH_TABLE hash_table;";
	output << "\nNODE node_0 = { key : \"XXX\", next : NULL };";
	output << "\nNODE node_1 = { key : \"YYY\", next : NULL };";
	output << "\nNODE node_2 = { key : \"ZZZ\", next : NULL };";
	output << "\nNODE node_3 = { key : \"XXX\", next : NULL };";
	output << "\nNODE node_4 = { key : \"YYY\", next : NULL };";
	output << "\nNODE node_5 = { key : \"ZZZ\", next : NULL };";
	output << "\nNODE node_6 = { key : \"XXX\", next : NULL };";
	output << "\nNODE node_7 = { key : \"YYY\", next : NULL };";
	output << "\nNODE node_8 = { key : \"ZZZ\", next : NULL };";
	output << "\nNODE node_9 = { key : \"XXX\", next : NULL };";
	output << "\nNODE node_10 = { key : \"YYY\", next : NULL };";
	output << "\nNODE node_11 = { key : \"ZZZ\", next : NULL };";
	output << "\nhash_table.insert_node(&node_0);";
	output << "\nhash_table.insert_node(&node_1);";
	output << "\nhash_table.insert_node(&node_2);";
	output << "\nhash_table.insert_node(&node_3);";
	output << "\nhash_table.insert_node(&node_4);";
	output << "\nhash_table.insert_node(&node_5);";
	output << "\nhash_table.insert_node(&node_6);";
	output << "\nhash_table.insert_node(&node_7);";
	output << "\nhash_table.insert_node(&node_8);";
	output << "\nhash_table.insert_node(&node_9);";
	output << "\nhash_table.insert_node(&node_10);";
	output << "\nhash_table.insert_node(&node_11);";
	output << "\nhash_table.print(output);";
	output << "\nhash_table.remove_nodes_with_key(\"XXX\");";
	output << "\nhash_table.print(output);";
	HASH_TABLE hash_table;
	NODE node_0 = { key : "XXX", next : NULL };
	NODE node_1 = { key : "YYY", next : NULL };
	NODE node_2 = { key : "ZZZ", next : NULL };
	NODE node_3 = { key : "XXX", next : NULL };
	NODE node_4 = { key : "YYY", next : NULL };
	NODE node_5 = { key : "ZZZ", next : NULL };
	NODE node_6 = { key : "XXX", next : NULL };
	NODE node_7 = { key : "YYY", next : NULL };
	NODE node_8 = { key : "ZZZ", next : NULL };
```

```
	NODE node_9 = { key : "XXX", next : NULL };
	NODE node_10 = { key : "YYY", next : NULL };
	NODE node_11 = { key : "ZZZ", next : NULL };
	hash_table.insert_node(&node_0);
	hash_table.insert_node(&node_1);
	hash_table.insert_node(&node_2);
	hash_table.insert_node(&node_3);
	hash_table.insert_node(&node_4);
	hash_table.insert_node(&node_5);
	hash_table.insert_node(&node_6);
	hash_table.insert_node(&node_7);
	hash_table.insert_node(&node_8);
	hash_table.insert_node(&node_9);
	hash_table.insert_node(&node_10);
	hash_table.insert_node(&node_11);
	hash_table.print(output);
	hash_table.remove_nodes_with_key("XXX");
	hash_table.print(output);
}

/**
 * HASH_TABLE constructor, insert method, get nodes with key method, print method, and
destructor.
 */
void unit_test_5(std::ostream & output)
{
	output << "\n\n************************************************";
	output << "\nUnit Test # 5: HASH_TABLE constructor, insert method, get nodes with key
method, print method, and destructor.";
	output << "\n************************************************";
	output << "\nHASH_TABLE hash_table = HASH_TABLE(6);";
	output << "\nNODE node_0 = { key : \"AAAA\", next : NULL };";
	output << "\nNODE node_1 = { key : \"ABAB\", next : NULL };";
	output << "\nNODE node_2 = { key : \"AABB\", next : NULL };";
	output << "\nNODE node_3 = { key : \"CCCC\", next : NULL };";
	output << "\nNODE node_4 = { key : \"ABAB\", next : NULL };";
	output << "\nNODE node_5 = { key : \"CCCC\", next : NULL };";
	output << "\nNODE node_6 = { key : \"BBBB\", next : NULL };";
	output << "\nNODE node_7 = { key : \"ABAB\", next : NULL };";
	output << "\nNODE node_8 = { key : \"AAAA\", next : NULL };";
	output << "\nNODE node_9 = { key : \"CCCC\", next : NULL };";
	output << "\nNODE node_10 = { key : \"DDDD\", next : NULL };";
	output << "\nNODE node_11 = { key : \"AABB\", next : NULL };";
	output << "\nNODE node_12 = { key : \"EEEE\", next : NULL };";
```

```
	output << "\nNODE node_13 = { key : \"DDDD\", next : NULL };";
	output << "\nNODE node_14 = { key : \"ABAB\", next : NULL };";
	output << "\nhash_table.insert_node(&node_0);";
	output << "\nhash_table.insert_node(&node_1);";
	output << "\nhash_table.insert_node(&node_2);";
	output << "\nhash_table.insert_node(&node_3);";
	output << "\nhash_table.insert_node(&node_4);";
	output << "\nhash_table.insert_node(&node_5);";
	output << "\nhash_table.insert_node(&node_6);";
	output << "\nhash_table.insert_node(&node_7);";
	output << "\nhash_table.insert_node(&node_8);";
	output << "\nhash_table.insert_node(&node_9);";
	output << "\nhash_table.insert_node(&node_10);";
	output << "\nhash_table.insert_node(&node_11);";
	output << "\nhash_table.insert_node(&node_12);";
	output << "\nhash_table.insert_node(&node_13);";
	output << "\nhash_table.insert_node(&node_14);";
	output << "\noutput << hash_table; // functionally identical to hash_table.print(output)";
	output << "\nLINKED_LIST search_results =
hash_table.get_nodes_with_key(\"AAAA\");";
	output << "\noutput << search_results; // functionally identical to
search_results.print(output);";
	HASH_TABLE hash_table = HASH_TABLE(6);
	NODE node_0 = { key : "AAAA", next : NULL };
	NODE node_1 = { key : "ABAB", next : NULL };
	NODE node_2 = { key : "AABB", next : NULL };
	NODE node_3 = { key : "CCCC", next : NULL };
	NODE node_4 = { key : "ABAB", next : NULL };
	NODE node_5 = { key : "CCCC", next : NULL };
	NODE node_6 = { key : "BBBB", next : NULL };
	NODE node_7 = { key : "ABAB", next : NULL };
	NODE node_8 = { key : "AAAA", next : NULL };
	NODE node_9 = { key : "CCCC", next : NULL };
	NODE node_10 = { key : "DDDD", next : NULL };
	NODE node_11 = { key : "AABB", next : NULL };
	NODE node_12 = { key : "EEEE", next : NULL };
	NODE node_13 = { key : "DDDD", next : NULL };
	NODE node_14 = { key : "ABAB", next : NULL };
	hash_table.insert_node(&node_0);
	hash_table.insert_node(&node_1);
	hash_table.insert_node(&node_2);
	hash_table.insert_node(&node_3);
	hash_table.insert_node(&node_4);
	hash_table.insert_node(&node_5);
```

```
        hash_table.insert_node(&node_6);
        hash_table.insert_node(&node_7);
        hash_table.insert_node(&node_8);
        hash_table.insert_node(&node_9);
        hash_table.insert_node(&node_10);
        hash_table.insert_node(&node_11);
        hash_table.insert_node(&node_12);
        hash_table.insert_node(&node_13);
        hash_table.insert_node(&node_14);
        output << hash_table; // functionally identical to hash_table.print(output);
        LINKED_LIST search_results = hash_table.get_nodes_with_key("AAAA");
        output << search_results; // functionally identical to search_results.print(output);
}

/* program entry point */
int main()
{
        // Declare a file output stream object.
        std::ofstream file;

        /**
        * If hash_table_driver_output.txt does not already exist in the same directory as
hash_table_driver.cpp,
        * create a new file named hash_table_driver_output.txt.
        *
        * Open the plain-text file named hash_table_driver_output.txt
        * and set that file to be overwritten with program data.
        */
        file.open("hash_table_driver_output.txt");

        // Print an opening message to the command line terminal.
        std::cout << "\n\n--------------------------------";
        std::cout << "\nStart Of Program";
        std::cout << "\n--------------------------------";

        // Print an opening message to the file output stream.
        file << "--------------------------------";
        file << "\nStart Of Program";
        file << "\n--------------------------------";

        // Implement a series of unit tests which demonstrate the functionality of LINKED_LIST
class variables.
        unit_test_0(std::cout);
        unit_test_0(file);
```

```
	unit_test_1(std::cout);
	unit_test_1(file);
	unit_test_2(std::cout);
	unit_test_2(file);
	unit_test_3(std::cout);
	unit_test_3(file);
	unit_test_4(std::cout);
	unit_test_4(file);
	unit_test_5(std::cout);
	unit_test_5(file);

	// Print a closing message to the command line terminal.
	std::cout << "\n\n--------------------------------";
	std::cout << "\nEnd Of Program";
	std::cout << "\n--------------------------------\n\n";

	// Print a closing message to the file output stream.
	file << "\n\n--------------------------------";
	file << "\nEnd Of Program";
	file << "\n--------------------------------";

	// Close the file output stream.
	file.close();

	// Exit the program.
	return 0;
}
```

SAMPLE_PROGRAM_OUTPUT

The text in the preformatted text box below was generated by one use case of the C++ program featured in this computer programming tutorial web page.

plain-text_file:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/hash_table_driver_output.txt

---

```
--------------------------------
Start Of Program
--------------------------------
```

****
Unit Test # 0: HASH_TABLE constructor, print method, and destructor.
****

HASH_TABLE hash_table;
hash_table.print(output);

----

this = 0x7ffc9c06b880. // The keyword named this is a pointer which stores the memory address of the first memory cell of a HASH_TABLE sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE object.
&array = 0x7ffc9c06b880. // The reference operation returns the memory address of the first memory cell of a pointer-to-LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE data attribute named array.
&N = 0x7ffc9c06b888. // The reference operation returns the memory address of the first memory cell of a pointer-to-int sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE data attribute named N.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which a int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(HASH_TABLE) = 16. // The sizeof() operation returns the number of bytes of memory which a HASH_TABLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(HASH_TABLE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-HASH_TABLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
array = 0x564e5da804b8. // array stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a LINKED_LIST type variable or else array stores NULL (and the value NULL is displayed as 0).

N = 10. // N stores the total number of LINKED_LIST types elements which are represented by the array property of the caller HASH_TABLE object.
HASH_TABLE := {

###
array[0] := {

----
this = 0x564e5da804b8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804b8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da805a0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da805a0.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.

}.
###

###
array[1] := {

----
this = 0x564e5da804c0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804c0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da805d0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da805d0.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##

##
array[2] := {

----
this = 0x564e5da804c8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.

&head = 0x564e5da804c8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80600. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80600.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##

##
array[3] := {

----
this = 0x564e5da804d0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804d0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80630. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80630.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.
```

}.
##

##
array[4] := {

----
this = 0x564e5da804d8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804d8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).

head = 0x564e5da80660. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80660.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##

##
array[5] := {

----
this = 0x564e5da804e0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804e0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80690. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.

head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
    p := 0x564e5da80690.
    p -> key = HEAD.
    p -> next = 0.
  }.
}.

}.
##

##
array[6] := {

----
this = 0x564e5da804e8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804e8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da806c0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {

```
          p := 0x564e5da806c0.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.
--------------------------------------------------------------------------------------------------
}.
############################################################
```

```
############################################################
array[7] := {

--------------------------------------------------------------------------------------------------
```

this = 0x564e5da804f0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804f0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80510. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80510.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.
--------------------------------------------------------------------------------------------------
```

}.
##

##
array[8] := {

----
this = 0x564e5da804f8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804f8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80570. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80570.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.

}.
##

##
array[9] := {

----
this = 0x564e5da80500. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80500. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80540. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
 NODE_0 := {
        p := 0x564e5da80540.
        p -> key = HEAD.
        p -> next = 0.
 }.
}.
```

}.
##
}.

****

Unit Test # 1: HASH_TABLE constructor, insert method, print method, and destructor.

****

```
HASH_TABLE hash_table;
NODE node;
node.key = "unit_test_1";
node.next = NULL;
```

hash_table.insert_node(&node);
hash_table.print(output);

----
this = 0x7ffc9c06b860. // The keyword named this is a pointer which stores the memory address of the first memory cell of a HASH_TABLE sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE object.
&array = 0x7ffc9c06b860. // The reference operation returns the memory address of the first memory cell of a pointer-to-LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE data attribute named array.
&N = 0x7ffc9c06b868. // The reference operation returns the memory address of the first memory cell of a pointer-to-int sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE data attribute named N.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which a int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(HASH_TABLE) = 16. // The sizeof() operation returns the number of bytes of memory which a HASH_TABLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(HASH_TABLE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-HASH_TABLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
array = 0x564e5da804b8. // array stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a LINKED_LIST type variable or else array stores NULL (and the value NULL is displayed as 0).
N = 10. // N stores the total number of LINKED_LIST types elements which are represented by the array property of the caller HASH_TABLE object.
HASH_TABLE := {

###

array[0] := {

----
this = 0x564e5da804b8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804b8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da806c0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da806c0.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.

}.
###

###
array[1] := {

----
this = 0x564e5da804c0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.

&head = 0x564e5da804c0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80510. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80510.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##

##
array[2] := {

----
this = 0x564e5da804c8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804c8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80570. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80570.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.

}.
##

##
array[3] := {

----
this = 0x564e5da804d0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804d0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).

head = 0x564e5da80540. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80540.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
###

###
array[4] := {

----
this = 0x564e5da804d8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804d8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da805a0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.

head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
 p := 0x564e5da805a0.
 p -> key = HEAD.
 p -> next = 0.
 }.
}.

}.
##

##
array[5] := {

----
this = 0x564e5da804e0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804e0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80600. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b870. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 2.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {

p := 0x564e5da80600.
p -> key = HEAD.
p -> next = 0x7ffc9c06b870.
}.
NODE_1 := {
p := 0x7ffc9c06b870.
p -> key = unit_test_1.
p -> next = 0.
}.
}.

}.
##

##
array[6] := {

----
this = 0x564e5da804e8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804e8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da805d0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
NODE_0 := {
p := 0x564e5da805d0.

p -> key = HEAD.
p -> next = 0.
}.
}.

}.
###

###
array[7] := {

----
this = 0x564e5da804f0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804f0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80630. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
NODE_0 := {
p := 0x564e5da80630.
p -> key = HEAD.
p -> next = 0.
}.
}.

}.

###

###
array[8] := {

----
this = 0x564e5da804f8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804f8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80690. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80690.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
###

###
array[9] := {

----

this = 0x564e5da80500. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80500. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80660. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80660.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.
--------------------------------------------------------------------------------------------------
}.
#################################################################
}.
--------------------------------------------------------------------------------------------------
```

****

Unit Test # 2: HASH_TABLE constructor, insert method, print method, and destructor.

****

```
HASH_TABLE hash_table;
NODE node_A = { key : "node_A", next : NULL };
NODE node_B = { key : "node_B", next : NULL };
NODE node_C = { key : "node_C", next : NULL };
hash_table.insert_node(&node_A);
```

hash_table.insert_node(&node_B);
hash_table.insert_node(&node_C);
output < key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da805d0.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
###

###
array[1] := {

----
this = 0x564e5da804c0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804c0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80630. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.

get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
    p := 0x564e5da80630.
    p -> key = HEAD.
    p -> next = 0.
  }.
}.

}.
##

##
array[2] := {

----
this = 0x564e5da804c8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804c8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80690. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b810. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 2.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
    p := 0x564e5da80690.
    p -> key = HEAD.

```
        p -> next = 0x7ffc9c06b810.
  }.
  NODE_1 := {
        p := 0x7ffc9c06b810.
        p -> key = node_A.
        p -> next = 0.
  }.
}.
--------------------------------------------------------------------------------------------------
}.
#################################################################

#################################################################
array[3] := {

--------------------------------------------------------------------------------------------------
```

this = 0x564e5da804d0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804d0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80660. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b840. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 2.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
  NODE_0 := {
        p := 0x564e5da80660.
        p -> key = HEAD.
        p -> next = 0x7ffc9c06b840.
```

}.
NODE_1 := {
p := 0x7ffc9c06b840.
p -> key = node_B.
p -> next = 0.
}.
}.

}.
##

##
array[4] := {

----
this = 0x564e5da804d8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804d8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da806c0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b870. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 2.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
NODE_0 := {
p := 0x564e5da806c0.
p -> key = HEAD.
p -> next = 0x7ffc9c06b870.
}.

```
  NODE_1 := {
          p := 0x7ffc9c06b870.
          p -> key = node_C.
          p -> next = 0.
  }.
}.
--------------------------------------------------------------------------------------------------
}.
#################################################################

#################################################################
array[5] := {

--------------------------------------------------------------------------------------------------
```

this = 0x564e5da804e0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804e0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80570. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80570.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.
```

}.
##

##
array[6] := {

----
this = 0x564e5da804e8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804e8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80510. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80510.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##

##
array[7] := {

----
this = 0x564e5da804f0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804f0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80540. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80540.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##

##
array[8] := {

----
this = 0x564e5da804f8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.

&head = 0x564e5da804f8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80600. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80600.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.

}.
##

##
array[9] := {

----
this = 0x564e5da80500. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80500. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da805a0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da805a0.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
###
}.

****

Unit Test # 3: HASH_TABLE constructor, insert method, number of linked lists method, number of node method, print method, and destructor.

****

HASH_TABLE hash_table = HASH_TABLE(5);
NODE node_A = { key : "node_A", next : NULL };
NODE node_B = { key : "node_B", next : NULL };
NODE node_C = { key : "node_C", next : NULL };
NODE node_AA = { key : "node_AA", next : NULL };
NODE node_BB = { key : "node_BB", next : NULL };
NODE node_CC = { key : "node_CC", next : NULL };
NODE node_Z = { key : "node_Z", next : NULL };
NODE node_666 = { key : "node_666", next : NULL };
hash_table.insert_node(&node_A);
hash_table.insert_node(&node_B);
hash_table.insert_node(&node_C);

hash_table.insert_node(&node_AA);
hash_table.insert_node(&node_BB);
hash_table.insert_node(&node_CC);
hash_table.insert_node(&node_Z);
hash_table.insert_node(&node_666);
output < key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
 p := 0x564e5da80660.
 p -> key = HEAD.
 p -> next = 0.
 }.
}.

}.
###

###
array[1] := {

----
this = 0x564e5da80700. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80700. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da806c0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).

head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b810. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 2.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da806c0.
  p -> key = HEAD.
  p -> next = 0x7ffc9c06b810.
 }.
 NODE_1 := {
  p := 0x7ffc9c06b810.
  p -> key = node_CC.
  p -> next = 0.
 }.
}.

}.
###

###
array[2] := {

----
this = 0x564e5da80708. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80708. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80570. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).

head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b720. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 4.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
 p := 0x564e5da80570.
 p -> key = HEAD.
 p -> next = 0x7ffc9c06b720.
 }.
 NODE_1 := {
 p := 0x7ffc9c06b720.
 p -> key = node_A.
 p -> next = 0x7ffc9c06b7b0.
 }.
 NODE_2 := {
 p := 0x7ffc9c06b7b0.
 p -> key = node_AA.
 p -> next = 0x7ffc9c06b840.
 }.
 NODE_3 := {
 p := 0x7ffc9c06b840.
 p -> key = node_Z.
 p -> next = 0.
 }.
}.

}.
##

##
array[3] := {

----
this = 0x564e5da80710. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80710. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80510. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b750. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 2.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80510.
          p -> key = HEAD.
          p -> next = 0x7ffc9c06b750.
  }.
  NODE_1 := {
          p := 0x7ffc9c06b750.
          p -> key = node_B.
          p -> next = 0.
  }.
}.
```

}.
##

##
array[4] := {

----
this = 0x564e5da80718. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80718. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).

```
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80540. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b780. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 4.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80540.
          p -> key = HEAD.
          p -> next = 0x7ffc9c06b780.
  }.
  NODE_1 := {
          p := 0x7ffc9c06b780.
          p -> key = node_C.
          p -> next = 0x7ffc9c06b7e0.
  }.
  NODE_2 := {
          p := 0x7ffc9c06b7e0.
          p -> key = node_BB.
          p -> next = 0x7ffc9c06b870.
  }.
  NODE_3 := {
          p := 0x7ffc9c06b870.
          p -> key = node_666.
          p -> next = 0.
  }.
}.
--------------------------------------------------------------------------------------------------------
}.
################################################################
}.
--------------------------------------------------------------------------------------------------------

***************************************************
```

Unit Test # 4: HASH_TABLE constructor, insert method, remove method, print method, and destructor.

****

HASH_TABLE hash_table;
NODE node_0 = { key : "XXX", next : NULL };
NODE node_1 = { key : "YYY", next : NULL };
NODE node_2 = { key : "ZZZ", next : NULL };
NODE node_3 = { key : "XXX", next : NULL };
NODE node_4 = { key : "YYY", next : NULL };
NODE node_5 = { key : "ZZZ", next : NULL };
NODE node_6 = { key : "XXX", next : NULL };
NODE node_7 = { key : "YYY", next : NULL };
NODE node_8 = { key : "ZZZ", next : NULL };
NODE node_9 = { key : "XXX", next : NULL };
NODE node_10 = { key : "YYY", next : NULL };
NODE node_11 = { key : "ZZZ", next : NULL };
hash_table.insert_node(&node_0);
hash_table.insert_node(&node_1);
hash_table.insert_node(&node_2);
hash_table.insert_node(&node_3);
hash_table.insert_node(&node_4);
hash_table.insert_node(&node_5);
hash_table.insert_node(&node_6);
hash_table.insert_node(&node_7);
hash_table.insert_node(&node_8);
hash_table.insert_node(&node_9);
hash_table.insert_node(&node_10);
hash_table.insert_node(&node_11);
hash_table.print(output);
hash_table.remove_nodes_with_key("XXX");
hash_table.print(output);

----

this = 0x7ffc9c06b630. // The keyword named this is a pointer which stores the memory address of the first memory cell of a HASH_TABLE sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE object.
&array = 0x7ffc9c06b630. // The reference operation returns the memory address of the first memory cell of a pointer-to-LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE data attribute named array.
&N = 0x7ffc9c06b638. // The reference operation returns the memory address of the first memory cell of a pointer-to-int sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE data attribute named N.
sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which a int type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(int *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(HASH_TABLE) = 16. // The sizeof() operation returns the number of bytes of memory which a HASH_TABLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(HASH_TABLE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-HASH_TABLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
array = 0x564e5da804b8. // array stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a LINKED_LIST type variable or else array stores NULL (and the value NULL is displayed as 0).
N = 10. // N stores the total number of LINKED_LIST types elements which are represented by the array property of the caller HASH_TABLE object.
HASH_TABLE := {

##
array[0] := {

----
this = 0x564e5da804b8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804b8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80510. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b6c0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 5.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
   p := 0x564e5da80510.
   p -> key = HEAD.
   p -> next = 0x7ffc9c06b6c0.
 }.
 NODE_1 := {
   p := 0x7ffc9c06b6c0.
   p -> key = ZZZ.
   p -> next = 0x7ffc9c06b750.
 }.
 NODE_2 := {
   p := 0x7ffc9c06b750.
   p -> key = ZZZ.
   p -> next = 0x7ffc9c06b7e0.
 }.
 NODE_3 := {
   p := 0x7ffc9c06b7e0.
   p -> key = ZZZ.
   p -> next = 0x7ffc9c06b870.
 }.
 NODE_4 := {
   p := 0x7ffc9c06b870.
   p -> key = ZZZ.
   p -> next = 0.
 }.
}.

}.
##

##
array[1] := {

----
this = 0x564e5da804c0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804c0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80540. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80540.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
###

###
array[2] := {

----
this = 0x564e5da804c8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.

&head = 0x564e5da804c8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80600. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80600.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##

##
array[3] := {

----
this = 0x564e5da804d0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804d0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da805a0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da805a0.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##

##
array[4] := {

----
this = 0x564e5da804d8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804d8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).

```
head = 0x564e5da805d0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b660. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 5.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
	p := 0x564e5da805d0.
	p -> key = HEAD.
	p -> next = 0x7ffc9c06b660.
 }.
 NODE_1 := {
	p := 0x7ffc9c06b660.
	p -> key = XXX.
	p -> next = 0x7ffc9c06b6f0.
 }.
 NODE_2 := {
	p := 0x7ffc9c06b6f0.
	p -> key = XXX.
	p -> next = 0x7ffc9c06b780.
 }.
 NODE_3 := {
	p := 0x7ffc9c06b780.
	p -> key = XXX.
	p -> next = 0x7ffc9c06b810.
 }.
 NODE_4 := {
	p := 0x7ffc9c06b810.
	p -> key = XXX.
	p -> next = 0.
 }.
}.
--------------------------------------------------------------------------------------------------
}.
################################################################

################################################################
array[5] := {

--------------------------------------------------------------------------------------------------
```

this = 0x564e5da804e0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804e0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80690. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80690.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.
--------------------------------------------------------------------------------------------------
}.
#################################################################

#################################################################
array[6] := {

--------------------------------------------------------------------------------------------------
```

this = 0x564e5da804e8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804e8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.

sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80630. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
 p := 0x564e5da80630.
 p -> key = HEAD.
 p -> next = 0.
 }.
}.

}.
##

##
array[7] := {

----
this = 0x564e5da804f0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804f0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80660. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b690. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 5.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
 NODE_0 := {
        p := 0x564e5da80660.
        p -> key = HEAD.
        p -> next = 0x7ffc9c06b690.
 }.
 NODE_1 := {
        p := 0x7ffc9c06b690.
        p -> key = YYY.
        p -> next = 0x7ffc9c06b720.
 }.
 NODE_2 := {
        p := 0x7ffc9c06b720.
        p -> key = YYY.
        p -> next = 0x7ffc9c06b7b0.
 }.
 NODE_3 := {
        p := 0x7ffc9c06b7b0.
        p -> key = YYY.
        p -> next = 0x7ffc9c06b840.
 }.
 NODE_4 := {
        p := 0x7ffc9c06b840.
        p -> key = YYY.
        p -> next = 0.
 }.
}.
--------------------------------------------------------------------------------------------------
}.
################################################################

################################################################
array[8] := {
```

----
this = 0x564e5da804f8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804f8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80570. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80570.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.

}.
##

##
array[9] := {

----
this = 0x564e5da80500. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.

&head = 0x564e5da80500. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da806c0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
 NODE_0 := {
          p := 0x564e5da806c0.
          p -> key = HEAD.
          p -> next = 0.
 }.
}.
--------------------------------------------------------------------------------------------
}.
#################################################################
}.
--------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------------------
```

this = 0x7ffc9c06b630. // The keyword named this is a pointer which stores the memory address of the first memory cell of a HASH_TABLE sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE object.
&array = 0x7ffc9c06b630. // The reference operation returns the memory address of the first memory cell of a pointer-to-LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE data attribute named array.
&N = 0x7ffc9c06b638. // The reference operation returns the memory address of the first memory cell of a pointer-to-int sized chunk of contiguous memory cells which are allocated to the caller HASH_TABLE data attribute named N.

sizeof(int) = 4. // The sizeof() operation returns the number of bytes of memory which a int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(int *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-int type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(HASH_TABLE) = 16. // The sizeof() operation returns the number of bytes of memory which a HASH_TABLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(HASH_TABLE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-HASH_TABLE type variable occupies. (Each memory cell has a data capacity of 1 byte).
array = 0x564e5da804b8. // array stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a LINKED_LIST type variable or else array stores NULL (and the value NULL is displayed as 0).
N = 10. // N stores the total number of LINKED_LIST types elements which are represented by the array property of the caller HASH_TABLE object.
HASH_TABLE := {

##
array[0] := {

----
this = 0x564e5da804b8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804b8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80510. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b6c0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 5.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
 NODE_0 := {
        p := 0x564e5da80510.
        p -> key = HEAD.
        p -> next = 0x7ffc9c06b6c0.
 }.
 NODE_1 := {
        p := 0x7ffc9c06b6c0.
        p -> key = ZZZ.
        p -> next = 0x7ffc9c06b750.
 }.
 NODE_2 := {
        p := 0x7ffc9c06b750.
        p -> key = ZZZ.
        p -> next = 0x7ffc9c06b7e0.
 }.
 NODE_3 := {
        p := 0x7ffc9c06b7e0.
        p -> key = ZZZ.
        p -> next = 0x7ffc9c06b870.
 }.
 NODE_4 := {
        p := 0x7ffc9c06b870.
        p -> key = ZZZ.
        p -> next = 0.
 }.
}.
--------------------------------------------------------------------------------------------------
}.
#################################################################
```

##
array[1] := {

----
this = 0x564e5da804c0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804c0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80540. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80540.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##

##
array[2] := {

----

this = 0x564e5da804c8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804c8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80600. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
NODE_0 := {
p := 0x564e5da80600.
p -> key = HEAD.
p -> next = 0.
}.
}.

}.
##

##
array[3] := {

----
this = 0x564e5da804d0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804d0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.

sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da805a0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da805a0.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.
--------------------------------------------------------------------------------------------------
}.
############################################################

############################################################
array[4] := {

--------------------------------------------------------------------------------------------------
```

this = 0x564e5da804d8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804d8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da805d0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da805d0.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
###

###
array[5] := {

----
this = 0x564e5da804e0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804e0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80690. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).

head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
      p := 0x564e5da80690.
      p -> key = HEAD.
      p -> next = 0.
  }.
}.

}.
##

##
array[6] := {

----
this = 0x564e5da804e8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804e8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80630. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.

LINKED_LIST := {
  NODE_0 := {
      p := 0x564e5da80630.
      p -> key = HEAD.
      p -> next = 0.
  }.
}.

}.
##

##
array[7] := {

----
this = 0x564e5da804f0. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804f0. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80660. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b690. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 5.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
      p := 0x564e5da80660.
      p -> key = HEAD.
      p -> next = 0x7ffc9c06b690.
  }.

```
  NODE_1 := {
        p := 0x7ffc9c06b690.
        p -> key = YYY.
        p -> next = 0x7ffc9c06b720.
  }.
  NODE_2 := {
        p := 0x7ffc9c06b720.
        p -> key = YYY.
        p -> next = 0x7ffc9c06b7b0.
  }.
  NODE_3 := {
        p := 0x7ffc9c06b7b0.
        p -> key = YYY.
        p -> next = 0x7ffc9c06b840.
  }.
  NODE_4 := {
        p := 0x7ffc9c06b840.
        p -> key = YYY.
        p -> next = 0.
  }.
}.
--------------------------------------------------------------------------------------------
}.
#################################################################

#################################################################
array[8] := {

--------------------------------------------------------------------------------------------
this = 0x564e5da804f8. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da804f8. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
```

head = 0x564e5da80570. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80570.
          p -> key = HEAD.
          p -> next = 0.
  }.
}.

}.
###

###
array[9] := {

----
this = 0x564e5da80500. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80500. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da806c0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.

head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
      p := 0x564e5da806c0.
      p -> key = HEAD.
      p -> next = 0.
  }.
}.

}.
###
}.

****

Unit Test # 5: HASH_TABLE constructor, insert method, get nodes with key method, print method, and destructor.

****

HASH_TABLE hash_table = HASH_TABLE(6);
NODE node_0 = { key : "AAAA", next : NULL };
NODE node_1 = { key : "ABAB", next : NULL };
NODE node_2 = { key : "AABB", next : NULL };
NODE node_3 = { key : "CCCC", next : NULL };
NODE node_4 = { key : "ABAB", next : NULL };
NODE node_5 = { key : "CCCC", next : NULL };
NODE node_6 = { key : "BBBB", next : NULL };
NODE node_7 = { key : "ABAB", next : NULL };
NODE node_8 = { key : "AAAA", next : NULL };
NODE node_9 = { key : "CCCC", next : NULL };
NODE node_10 = { key : "DDDD", next : NULL };
NODE node_11 = { key : "AABB", next : NULL };
NODE node_12 = { key : "EEEE", next : NULL };
NODE node_13 = { key : "DDDD", next : NULL };
NODE node_14 = { key : "ABAB", next : NULL };
hash_table.insert_node(&node_0);
hash_table.insert_node(&node_1);
hash_table.insert_node(&node_2);
hash_table.insert_node(&node_3);
hash_table.insert_node(&node_4);
hash_table.insert_node(&node_5);
hash_table.insert_node(&node_6);

hash_table.insert_node(&node_7);
hash_table.insert_node(&node_8);
hash_table.insert_node(&node_9);
hash_table.insert_node(&node_10);
hash_table.insert_node(&node_11);
hash_table.insert_node(&node_12);
hash_table.insert_node(&node_13);
hash_table.insert_node(&node_14);
output << hash_table; // functionally identical to hash_table.print(output)
LINKED_LIST search_results = hash_table.get_nodes_with_key("AAAA");
output < key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b6f0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 3.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da805a0.
          p -> key = HEAD.
          p -> next = 0x7ffc9c06b6f0.
  }.
  NODE_1 := {
          p := 0x7ffc9c06b6f0.
          p -> key = BBBB.
          p -> next = 0x7ffc9c06b810.
  }.
  NODE_2 := {
          p := 0x7ffc9c06b810.
          p -> key = EEEE.
          p -> next = 0.
  }.
}.

}.
##

##
array[1] := {

----
this = 0x564e5da80700. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.

&head = 0x564e5da80700. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da805d0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da805d0.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##

##
array[2] := {

----
this = 0x564e5da80708. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80708. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80690. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b5d0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 5.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
  NODE_0 := {
          p := 0x564e5da80690.
          p -> key = HEAD.
          p -> next = 0x7ffc9c06b5d0.
  }.
  NODE_1 := {
          p := 0x7ffc9c06b5d0.
          p -> key = AAAA.
          p -> next = 0x7ffc9c06b750.
  }.
  NODE_2 := {
          p := 0x7ffc9c06b750.
          p -> key = AAAA.
          p -> next = 0x7ffc9c06b7b0.
  }.
  NODE_3 := {
          p := 0x7ffc9c06b7b0.
          p -> key = DDDD.
          p -> next = 0x7ffc9c06b840.
  }.
  NODE_4 := {
          p := 0x7ffc9c06b840.
          p -> key = DDDD.
          p -> next = 0.
  }.
}.
--------------------------------------------------------------------------------------------------
}.
```

###

###
array[3] := {

----
this = 0x564e5da80710. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80710. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80630. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80630.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
###

###
array[4] := {

----

this = 0x564e5da80718. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x564e5da80718. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80660. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b600. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 10.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
 NODE_0 := {
        p := 0x564e5da80660.
        p -> key = HEAD.
        p -> next = 0x7ffc9c06b600.
 }.
 NODE_1 := {
        p := 0x7ffc9c06b600.
        p -> key = ABAB.
        p -> next = 0x7ffc9c06b630.
 }.
 NODE_2 := {
        p := 0x7ffc9c06b630.
        p -> key = AABB.
        p -> next = 0x7ffc9c06b660.
 }.
 NODE_3 := {
        p := 0x7ffc9c06b660.
        p -> key = CCCC.
        p -> next = 0x7ffc9c06b690.
 }.
```

```
 NODE_4 := {
        p := 0x7ffc9c06b690.
        p -> key = ABAB.
        p -> next = 0x7ffc9c06b6c0.
 }.
 NODE_5 := {
        p := 0x7ffc9c06b6c0.
        p -> key = CCCC.
        p -> next = 0x7ffc9c06b720.
 }.
 NODE_6 := {
        p := 0x7ffc9c06b720.
        p -> key = ABAB.
        p -> next = 0x7ffc9c06b780.
 }.
 NODE_7 := {
        p := 0x7ffc9c06b780.
        p -> key = CCCC.
        p -> next = 0x7ffc9c06b7e0.
 }.
 NODE_8 := {
        p := 0x7ffc9c06b7e0.
        p -> key = AABB.
        p -> next = 0x7ffc9c06b870.
 }.
 NODE_9 := {
        p := 0x7ffc9c06b870.
        p -> key = ABAB.
        p -> next = 0.
 }.
}.
--------------------------------------------------------------------------------------------------
}.
#################################################################

#################################################################
array[5] := {

--------------------------------------------------------------------------------------------------
this = 0x564e5da80720. // The keyword named this is a pointer which stores the memory
address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells
which are allocated to the caller LINKED_LIST object.
```

&head = 0x564e5da80720. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da80570. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 1.
// p is a pointer to a NODE type variable.
LINKED_LIST := {
 NODE_0 := {
  p := 0x564e5da80570.
  p -> key = HEAD.
  p -> next = 0.
 }.
}.

}.
##
}.

----
this = 0x7ffc9c06b598. // The keyword named this is a pointer which stores the memory address of the first memory cell of a LINKED_LIST sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST object.
&head = 0x7ffc9c06b598. // The reference operation returns the memory address of the first memory cell of a pointer-to-NODE sized chunk of contiguous memory cells which are allocated to the caller LINKED_LIST data attribute named head.
sizeof(NODE) = 40. // The sizeof() operation returns the number of bytes of memory which a NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(std::string) = 32. // The sizeof() operation returns the number of bytes of memory which a string type variable occupies. (Each memory cell has a data capacity of 1 byte).

sizeof(NODE *) = 8. // The sizeof() operation returns the number of bytes of memory which a pointer-to-NODE type variable occupies. (Each memory cell has a data capacity of 1 byte).
sizeof(LINKED_LIST) = 8. // The sizeof() operation returns the number of bytes of memory which a LINKED_LIST type variable occupies. (Each memory cell has a data capacity of 1 byte).
head = 0x564e5da806c0. // head stores either the first memory cell of a contiguous chunk of memory cells which are allocated to a NODE type variable or else head stores NULL (and the value NULL is displayed as 0).
head -> key = HEAD. // The arrow operator returns the string type property named key of the NODE type variable which head points to.
head -> next = 0x7ffc9c06b5d0. // The arrow operator returns the pointer-to-NODE type property named next of the NODE type variable which head points to.
get_number_of_nodes_in_list() = 3.
// p is a pointer to a NODE type variable.

```
LINKED_LIST := {
 NODE_0 := {
         p := 0x564e5da806c0.
         p -> key = HEAD.
         p -> next = 0x7ffc9c06b5d0.
 }.
 NODE_1 := {
         p := 0x7ffc9c06b5d0.
         p -> key = AAAA.
         p -> next = 0x7ffc9c06b750.
 }.
 NODE_2 := {
         p := 0x7ffc9c06b750.
         p -> key = AAAA.
         p -> next = 0.
 }.
}.
```

End Of Program

[End of abridged plain-text content from HASH_TABLE]

MULTIVERSE

---

image_link:
https://raw.githubusercontent.com/karlinarayberinger/KARLINA_OBJECT_summer_2023_starter_pack/main/continuum_and_slices_07_september_2022.jpg

---

The following terms and their respective definitions describe nature (i.e. the whole of reality) as being the set which contains all universes simultaneously and immutably.

UNIVERSE: exactly one space-time continuum which contains a finite amount of matter and energy which is arranged in a particular pattern (according to a partial frame of reference which appears to itself to be inhabiting that space-time continuum) and such that the observed pattern of energy and matter changes as the time within that space-time continuum elapses according to that observing partial frame of reference which observes that passage of time as the appearance and the disappearance of specific phenomena.

An information processing agent, A, which perceives itself as being a finite subset of its encompassing universe (i.e. environment), E, may possibly perceive phenomena which are interpreted by A as evidence of there being information processing agents which are not A and which are also inhabiting E.

Suppose A's mental model of reality suggests that E simultaneously contains A and some other information processing agents, B and C. If B and C are each information processing agents with a similar degree of information processing ability as A, then it could be said that A, B, and C are each information processing agents which are either directly or indirectly interacting with each other inside of one shared meta-universe due to the fact that each A, B, and C inevitably updates E by interacting with E (yet each A, B, and C technically inhabits its own unique solipsistic universe).

ABSOLUTE_SOLIPSISM: the hypothetical state of there being exactly one information processing agent throughout all of nature such that no phenomena exist outside of that information processing agent's frame of reference.

(Logically speaking, all imaginable noumena exist unconditionally).

If absolute solipsism is true, then the frame of reference which is observing this web page is the only frame of reference throughout all of nature which currently exists (and, also, no information processing agent other than you has ever existed nor will ever exist nor presently exists).

MULTIVERSE: the hypothetical coexistence of multiple universes simultaneously.

Hypothetically speaking, nature can be described as a multiverse which contains every imaginable universe simultaneously and immutably such that nature contains infinitely many possible universes at all times (whether any of those universes are noumenal or else phenomenal).

According to the “many worlds hypothesis”, every possible outcome to every decision exists inside of its own separate universe (and when a decision is made, the universe in which that particular decision is made splits into as many separate universes as there are possible outcomes to that decision).

(The following quoted text is a modified copy of a Twitter post which karbytes posted on 08_AUGUST_2023: “I hypothesize that every phenomenon is the only one of its kind throughout some multiverse (and that multiverse may or may not be a subset of a larger encompassing multiverse). What I mean to suggest is that every phenomenon (but not necessarily every noumenon) is always instantiated using a unique and finite allocation of space, time, matter, and energy.”)

This web page was last updated on 20_OCTOBER_2023 The content displayed on this web page is licensed as PUBLIC_DOMAIN intellectual property.

[End of abridged plain-text content from MULTIVERSE]
