

Workshop

# Bug-free Coding with SPARK Ada



1

*Using your Web Browser,  
Open this URL:*

**<http://mlhlocal.host/lhd-resources>**

---

2

*Click on the workshop you're attending, and find:*

- Setup Instructions
- The Code Samples
- A demo project
- A Workshop FAQ
- These Workshop Slides
- More Learning Resources



***Our Mission** is to Empower Hackers.*

**65,000+**  
HACKERS

**12,000+**  
PROJECTS CREATED


**400+**  
CITIES

***We hope you learn something awesome today!***  
*Find more resources: <http://mlh.io/>*

# What will you **learn** today?

- 1 What makes the Ada programming language special & why you should use it.
- 2 How to write bug-free code with a provable subset called SPARK Ada.
- 3 How to implement the Stack Data Structure in SPARK Ada.

# Table of Contents

-  **1. What are Ada & SPARK?**
- 2. Intro to SPARK Ada**
- 3. Set Up your Environment**
- 4. Let's Build a Stack!**
- 5. Review & Quiz**
- 6. Next Steps**

# Meet the Ada programming language!

Ada is a general purpose, modern programming language like Java, C, or C++.

- Ada's creation was sponsored by the USA's Dept. of Defense (DoD) in the 1970s and 1980s.
- The goal of the Ada programming language is to give developers the tools necessary to write safer, more maintainable code.

The word "Ada" is displayed in a large, bold, blue sans-serif font, centered within a white square that has a thin black border.

# Ada

# Ada vs. C, C++, & Java

In high reliability software, you are usually choosing between C, C++, Java, and Ada. So, why Ada?

## What is Ada good at?

Ada is a mature language that enables you to write efficient code on a variety of platforms. It's known for its:

- Reliability
- Maintainability
- Testability
- Formal Proofs (Bug-Free!)
- Human-Friendliness

### Ada's Key Value

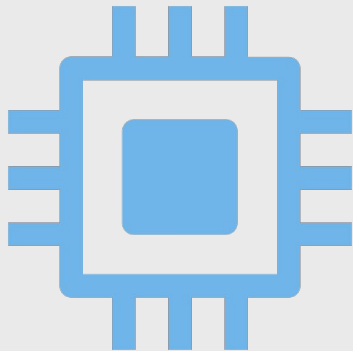
Unlike C, C++, and Java, Ada can actually help you write **more** reliable and maintainable code.



# When would you want to use Ada?

You could use Ada for just about anything, but it's especially useful for writing...

## High-Integrity Embedded Systems



Planes, trains,  
spaceships, boats, etc.

## Long-Lived Applications



Applications that need  
to last for 20+ years.

## Apps where Human Lives are at Stake

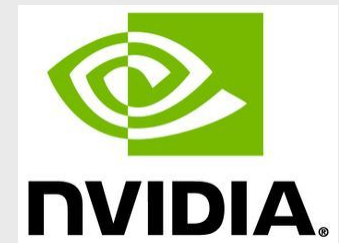


Autonomous cars,  
railroad signals, etc.



# Who uses Ada?

Ada is used by both big & small organizations to write mission critical software that is bug-free!



And so many more...

# What is AdaCore?


AdaCore creates tools for developers to write safe and secure software.

<http://www.adacore.com/>

The logo for AdaCore, featuring the word "AdaCore" in a blue, sans-serif font. The "Ada" is in a darker blue and the "Core" is in a lighter blue. The logo is enclosed in a thin black rectangular border.

- AdaCore was founded in 1994.
- AdaCore has over 100 employees in the US, France, UK, and Estonia.
- AdaCore's products are used in many "high integrity" industries (Military, Space, Railroad, Automotive, Aviation, etc.)
- AdaCore loves empowering hackers & sponsoring these workshops!

# Table of Contents

1. What are Ada & SPARK?
-  2. Intro to SPARK Ada
3. Set Up your Environment
4. Let's Build a Stack!
5. Review & Quiz
6. Next Steps

# Let's take a quick tour of SPARK Ada.

We're going to cover the language at a high level:

- If you're familiar with C, C++, or Java, Ada will look very familiar.
- Our goal is to enable you to write your first Ada program!
- We'll also cover some useful SPARK specific features.



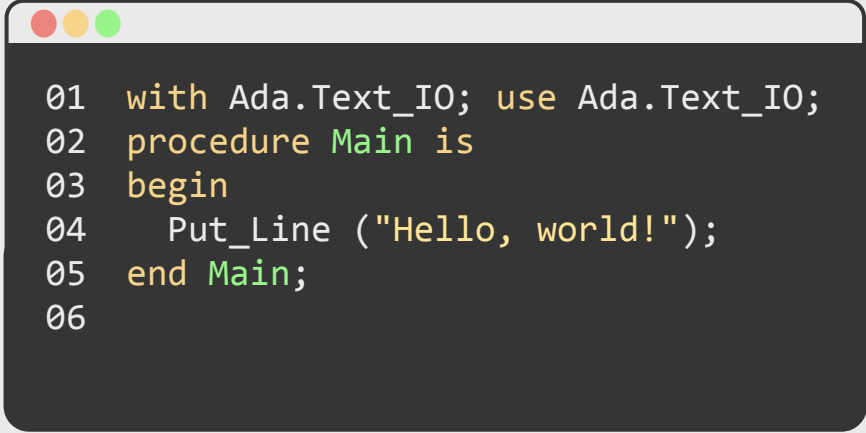
**Fun Fact:** Ada was named after Ada Lovelace, the first programmer!

# Hello World.

The first program we usually write in a new language is “Hello World”, which just prints those words.

## So, what is this code doing?

- **Line 1:** Import the Ada.Text\_IO package then make its namespace immediately visible.
- **Line 2 - 3:** Define a new procedure called “Main”.
- **Line 4:** Print the string “Hello, world!” to the console.
- **Line 5:** Close the procedure.
  - Ada uses keywords rather than symbols ({ }) to define scope.



```
01 with Ada.Text_IO; use Ada.Text_IO;
02 procedure Main is
03 begin
04   Put_Line ("Hello, world!");
05 end Main;
06
```

# Data Types.

Your programs will need to interact with data.  
What types does Ada have to offer?

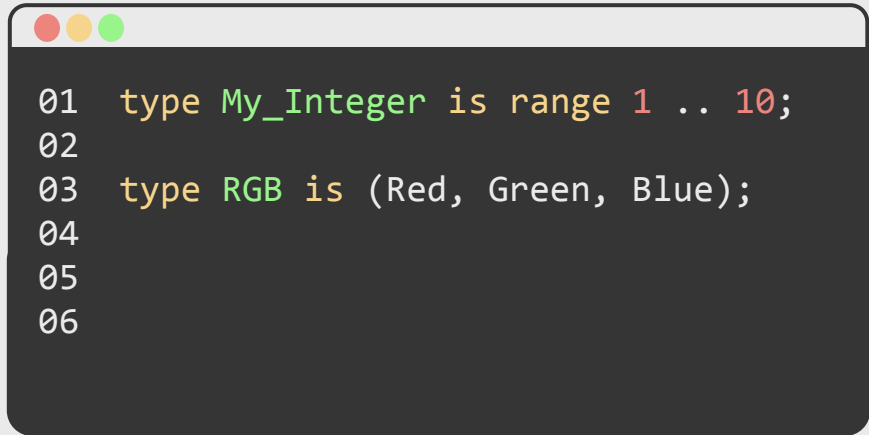
## Built-in Types:

Ada has a number of built-in data types that you can use in your programs. Today we'll be using:

- Integer
- Character
- String
- Boolean

## Custom Types:

You can also create your own types.



```
01  type My_Integer is range 1 .. 10;  
02  
03  type RGB is (Red, Green, Blue);  
04  
05  
06
```

# Variables & Assignment.

You're going to need some place to store data in your programs. Variables hold data & allow you to access it.

## Assignment

Assignment statements are written as `name := value`. For example, `X := 10;` sets the variable `X` to `10`.

## Initialization

Variable initializations use the following format:

```
[NAME] : [TYPE] := [VALUE];
```

For example: `A : Integer := 10;`

## Naming

In Ada, variable names are case insensitive. `HELLO` is the same as `Hello` is the same as `hello`.

## Note: = vs. :=

In Ada, the `=` (single equal) is for comparison and `:=` is for assignment.

# Strong Types.

Ada is a strongly typed language which means that variable types are predefined and do not change.

## Declaring Variables.

Since Ada is strongly typed, we need to declare variables up front.

```
01 declare
02   X : Integer := 10;
03 begin
04   Do_Something (X);
05 end;
06
```

## Mixing Types.

Since Ada is strongly typed, you can't combine objects of different types without explicit conversions.

```
01 Len_1 : Miles := 5.0;
02 Len_2 : Kilometers := 10.0;
03
04 -- This would cause an error!
05 D : Kilometers := Len_1 + Len_2;
06
```



# Control Flow.

Ada comes with a variety of control flow structures. The two we'll be looking at today are **if** statements and **for** loops.

## If Statements.

If statements are useful for logical branching of code.

```
01  if condition then
02      statement;
03  elsif condition2 then
04      statement2;
05  else
06      statement3;
07  end if;
```

## For Loops.

You can use for loops to iterate over ranges of objects or arrays.

```
01  for I in Integer range 1 .. 10 loop
02      statement;
03  end loop;
04
05
06
```

# Procedures vs. Functions.

Functions & procedures group code together to perform a task.  
For example, you could write a function to add numbers:

```
01  with Ada.Text_IO; use Ada.Text_IO;
02
03  procedure Main is
04
05      function Add (A : Integer; B : Integer) return Integer is
06      begin
07          return A + B;
08      end Add;
09
10  begin
11      Put_Line ("3 + 5 = " & Add(3, 5)'Image);
12  end Main;
```

Procedures are functions that don't return any value (like C void functions). They can alter the parameters that are passed in though.

# Ada is so much more...

This is just the tip of the iceberg. Ada is a very mature, modern language with lots of useful features.

## Ada also includes...

1. Object Oriented Programming
2. Generics (Templates)
3. Concurrency/Multitasking
4. Packages
5. And so much more!



# Meet SPARK Ada.

## Key Term

**SPARK Ada:** A subset of the Ada programming language, designed for program verification. In this workshop we'll be using SPARK Ada.

## Key Term

**Program Verification:** How to check that source code is well formed, performs as intended, and is bug free.

The logo for SPARK2014, featuring the word "SPARK" in white and "2014" in purple, set against a dark purple background with a subtle geometric pattern.

## SPARK Ada can prove...

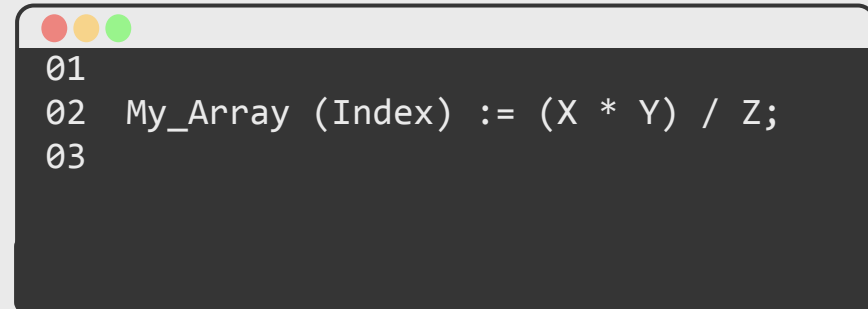
- **There are No Syntax Errors** - Is the code properly formatted & will it compile?
- **There are No Run-time Errors** - No exceptions, no buffer overflows, no division by zero, etc.
- **Your Theories** - True or False expressions that you write and the compiler verifies.

# How does SPARK find bugs?

SPARK Ada's tools analyze your code for exceptions & erroneous behavior.

## Questions SPARK asks...

- Are `My_Array`, `Index`, `X`, `Y`, and `Z` initialized?
- Is `Index` in the bounds of `My_Array`?
- Could `(X * Y) / Z` be potentially out of range?
- Could `(X * Y) / Z` potentially cause an overflow?
- Is `Z` potentially equal to `0`?



```
01
02  My_Array (Index) := (X * Y) / Z;
03
```

# A SPARK Ada example.

Here is a **procedure** called **Inc** that takes in a “small int” and increments it by **+1**.

inc.adb:

```
01 package body Inc
02   with SPARK_Mode => On
03   is
04
05     procedure Inc (X: in out Small_Int) is
06     begin
07       X := X + 1;
08     end Inc;
09
10 end Inc;
```

## What does this code do?

- **Line 1:** Define the package body for **Inc**.
- **Line 2:** Enable SPARK mode for program verification.
- **Line 5:** Define a procedure called **Inc** that takes a **Small\_Int** called **X** & updates it.
- **Lines 6 - 8:** Update the variable **X** to equal itself plus **1**.

# A SPARK Ada example.

What about the package specification for the Inc package?

inc.ads:

```
01 package Inc
02   with SPARK_Mode => On
03   is
04
05     type Small_Int is range -128 .. 127;
06
07     procedure Inc (X: in out Small_Int)
08       with Pre  => (X < Small_Int'Last),
09            Post => (X = X'Old + 1);
10
11   end Inc;
```

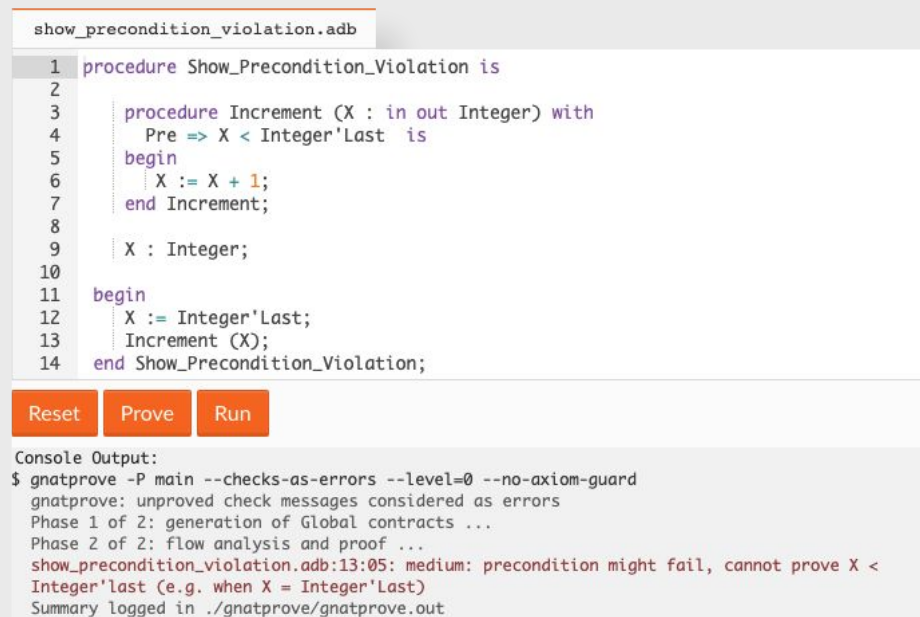
(\*) Lines 8 & 9 use SPARK's proof of properties feature.

## What does this code do?

- **Line 1:** Define the package specification for `Inc`.
- **Line 2:** Enable SPARK mode for program verification.
- **Line 5:** Define a new type called `Small_Int` that is an Integer between `-128` and `+127`.
- **Line 7:** Define a specification for calling `Inc`, a procedure that takes a `Small_Int` called `X` and updates it.
- **Line 8:** Verify that `X` is always less than `+127` before the procedure is called.
- **Line 9:** Verify that `X`'s new value is equal to `X`'s old value plus `1` after the procedure completes.

# Running the SPARK Prover.

When you run the SPARK prover, it will identify the specific lines in your code that could cause bugs.



The screenshot shows the SPARK Prover interface. At the top, a file named `show_precondition_violation.adb` is open in the editor. The code defines a procedure `Show_Precondition_Violation` that calls `Increment` with an argument that may violate its precondition. Below the editor are three buttons: `Reset`, `Prove`, and `Run`. The console output shows the execution of the prover, which identifies a precondition violation in the `Increment` procedure.

```
show_precondition_violation.adb
1 procedure Show_Precondition_Violation is
2
3   procedure Increment (X : in out Integer) with
4     Pre => X < Integer'Last is
5   begin
6     X := X + 1;
7   end Increment;
8
9   X : Integer;
10
11 begin
12   X := Integer'Last;
13   Increment (X);
14 end Show_Precondition_Violation;
```

Reset Prove Run


Console Output:

```
$ gnatprove -P main --checks-as-errors --level=0 --no-axiom-guard
gnatprove: unproved check messages considered as errors
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
show_precondition_violation.adb:13:05: medium: precondition might fail, cannot prove X <
Integer'last (e.g. when X = Integer'Last)
Summary logged in ./gnatprove/gnatprove.out
```

For example, here we are calling `Increment` with an argument that will cause the `Pre` check to fail.



# Table of Contents

1. What are Ada & SPARK?
2. Intro to SPARK Ada
-  3. Set Up your Environment
4. Let's Build a Stack!
5. Review & Quiz
6. Next Steps

## Let's Go to [mlhlocal.host/learn-ada](http://mlhlocal.host/learn-ada)

Navigate to [mlhlocal.host/learn-ada](http://mlhlocal.host/learn-ada) in your browser and go to Labs->Major League Hacking->Let's Build a Stack

[mlhlocal.host/MLHLearnAda](http://mlhlocal.host/MLHLearnAda)

*Raise your hand if you're  
having any trouble!*



# Meet [mlhlocal.host/learn-ada](http://mlhlocal.host/learn-ada)

[Learn Adacore](http://Learn Adacore) is an interactive learning website designed to help you get started with Ada without installing any software.

The screenshot displays the Learn Adacore website interface. On the left is a dark sidebar with a 'BETA' badge, the 'LEARN. ADACORE.COM' logo, a search bar, and navigation links for 'About', 'Courses', and 'Labs'. The main content area has a white header with an 'Edit on GitHub' link. Below this, a description states: 'Learn.adacore.com is an interactive learning platform designed to teach the Ada and SPARK programming languages.' A code editor shows the 'learn.adb' file with the following Ada code:

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Learn is
4
5     subtype Alphabet is Character range 'A' .. 'Z';
6
7 begin
8
9     Put_Line ("Learning Ada from " & Alphabet'First & " to " & Alphabet'Last);
10
11 end Learn;
```

Below the code editor are 'Reset' and 'Run' buttons. At the bottom, two blue course cards are shown with left and right navigation arrows. The first card, 'Introduction to Ada', describes a course for those with basic programming knowledge. The second card, 'Introduction to SPARK', describes an interactive tutorial on SPARK programming and verification tools.

# Use tabs to navigate between source files



classify\_number.ads

classify\_number.adb

main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classify_Number (X : Integer) is
4 begin
5     if X > 0 then
6         Put_Line ("Positive");
7     elsif X <= 0 then
8         Put_Line ("Negative");
9     else
10        Put_Line ("Zero");
11    end if;
12 end Classify_Number;

```

5

☒ Test against custom input

Reset

Prove

Run

Submit

Console Output:

```

$ gprbuild -q -P main -gnatwa
classify_number.adb:7:12: warning: condition can only be False if invalid values present
classify_number.adb:7:12: warning: condition is always True
classify_number.adb:7:12: warning: (see test at line 5)
$ ./main 5
Positive

```

# The Reset button resets the editor back to the initial code



classify\_number.ads

classify\_number.adb

main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classify_Number (X : Integer) is
4 begin
5     if X > 0 then
6         Put_Line ("Positive");
7     elsif X <= 0 then
8         Put_Line ("Negative");
9     else
10        Put_Line ("Zero");
11    end if;
12 end Classify_Number;

```

5

☒ Test against custom input

Reset

Prove

Run

Submit

Console Output:

```

$ gprbuild -q -P main -gnatwa
classify_number.adb:7:12: warning: condition can only be False if invalid values present
classify_number.adb:7:12: warning: condition is always True
classify_number.adb:7:12: warning: (see test at line 5)
$ ./main 5
Positive

```

# The Run button will run your code with the custom input as command line arguments

Check this box!

classifiy\_number.ads
classifiy\_number.adb
main.adb

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classifiy_Number (X : Integer) is
4 begin
5   if X > 0 then
6     Put_Line ("Positive");
7   elsif X <= 0 then
8     Put_Line ("Negative");
9   else
10    Put_Line ("Zero");
11  end if;
12 end Classifiy_Number;

```

5

☒ Test against custom input

Reset Prove Run Submit

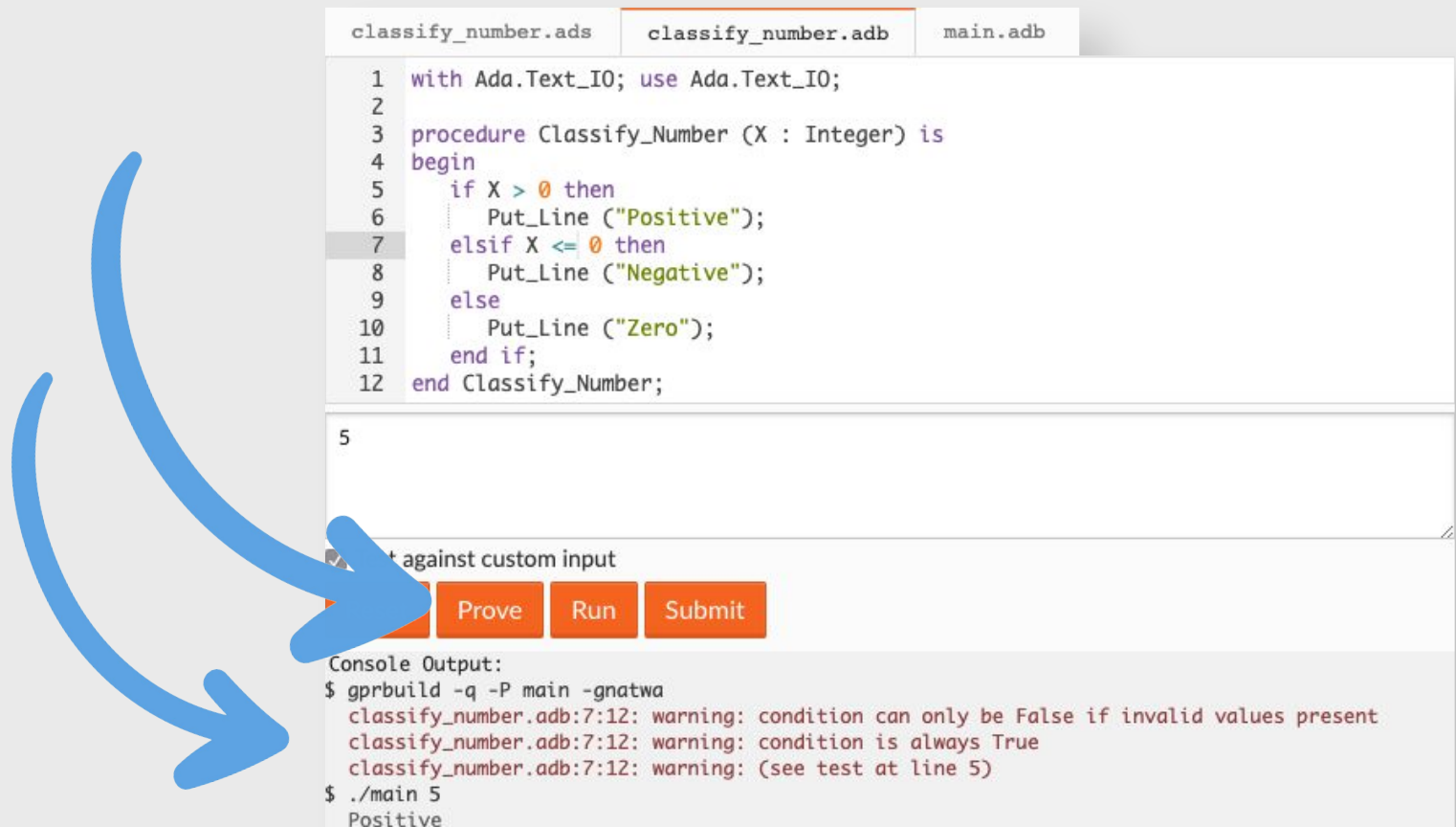
Console Output:

```

$ gprbuild -c -Ognatwa
classifiy_number.adb:7:12: warning: condition can only be False if invalid values present
classifiy_number.adb:7:12: warning: condition is always True
classifiy_number.adb:7:12: warning: (see test at line 5)
Positive

```

**The Prove button runs the SPARK provers. Warnings and errors will appear in the output window.**



The screenshot displays the SPARK IDE interface. At the top, there are three tabs: `classify_number.ads`, `classify_number.adb`, and `main.adb`. The `classify_number.adb` tab is active, showing the following Ada code:

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classify_Number (X : Integer) is
4 begin
5   if X > 0 then
6     Put_Line ("Positive");
7   elsif X <= 0 then
8     Put_Line ("Negative");
9   else
10    Put_Line ("Zero");
11  end if;
12 end Classify_Number;

```

Below the code editor, there is a text input field containing the number `5`. Underneath the input field, there are three buttons: `Prove`, `Run`, and `Submit`. A large blue arrow points from the `Prove` button to the console output area at the bottom.

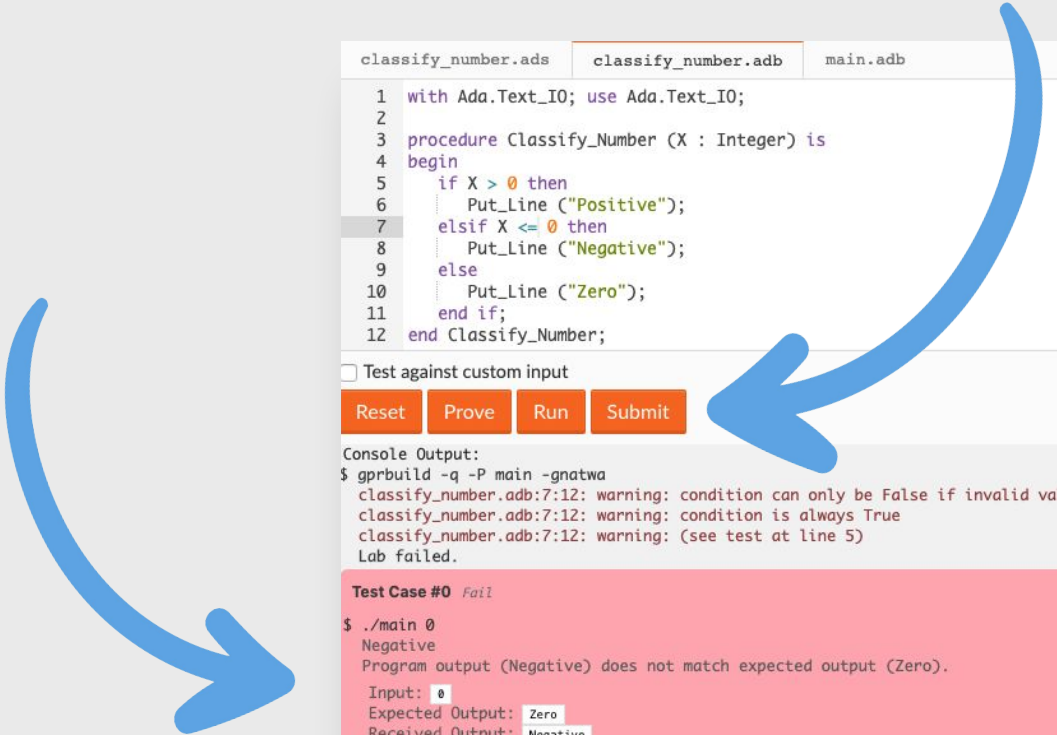
The console output area shows the following text:

```

Console Output:
$ gprbuild -q -P main -gnatwa
classify_number.adb:7:12: warning: condition can only be False if invalid values present
classify_number.adb:7:12: warning: condition is always True
classify_number.adb:7:12: warning: (see test at line 5)
$ ./main 5
Positive

```

**Submit your lab for grading using the Submit button.  
Your test results will appear below.**



The screenshot displays the SPARK Ada IDE interface. At the top, there are tabs for `classify_number.ads`, `classify_number.adb`, and `main.adb`. The `classify_number.adb` tab is active, showing the following Ada code:

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Classify_Number (X : Integer) is
4 begin
5   if X > 0 then
6     Put_Line ("Positive");
7   elsif X <= 0 then
8     Put_Line ("Negative");
9   else
10    Put_Line ("Zero");
11   end if;
12 end Classify_Number;
```

Below the code editor, there is a checkbox labeled "Test against custom input" which is unchecked. To the right of this checkbox are four buttons: "Reset", "Prove", "Run", and "Submit". A blue arrow points from the "Submit" button towards the test results section below.

The "Console Output" section shows the following text:


```
$ gprbuild -q -P main -gnatwa
classify_number.adb:7:12: warning: condition can only be False if invalid values present
classify_number.adb:7:12: warning: condition is always True
classify_number.adb:7:12: warning: (see test at line 5)
Lab failed.
```

The test results section is divided into four cases:

- Test Case #0 Fail** (highlighted in pink):  
\$ ./main 0  
Negative  
Program output (Negative) does not match expected output (Zero).  
Input: 0  
Expected Output: Zero  
Received Output: Negative  
Status: Failed
- Test Case #1 Pass** (highlighted in light green):
- Test Case #2 Pass** (highlighted in light green):
- Test Case #3 Pass** (highlighted in light green):
- Test Case #4 Pass** (highlighted in light green):



# Table of Contents

1. What are Ada & SPARK?
2. Intro to SPARK Ada
3. Set Up your Environment
-  4. Let's Build a Stack!
5. Review & Quiz
6. Next Steps

# What are we building today?

## We're building a Stack!

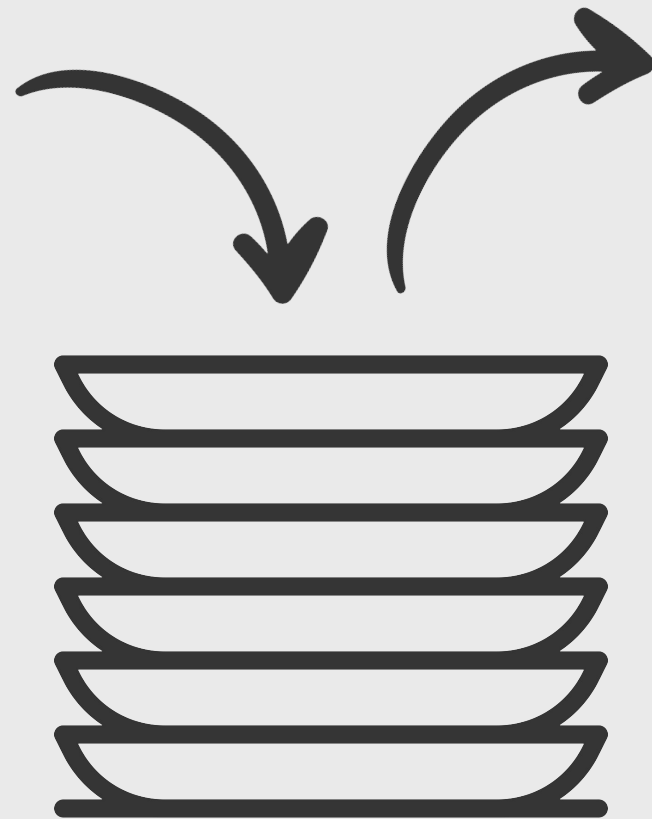
- *We're going to build a common data structure called a "Stack".*
- *You'll receive some sample code that's mostly working with a few bugs.*
- *We'll use some of SPARK Ada's cool features to find and fix the issues.*



# So, what is a Stack?

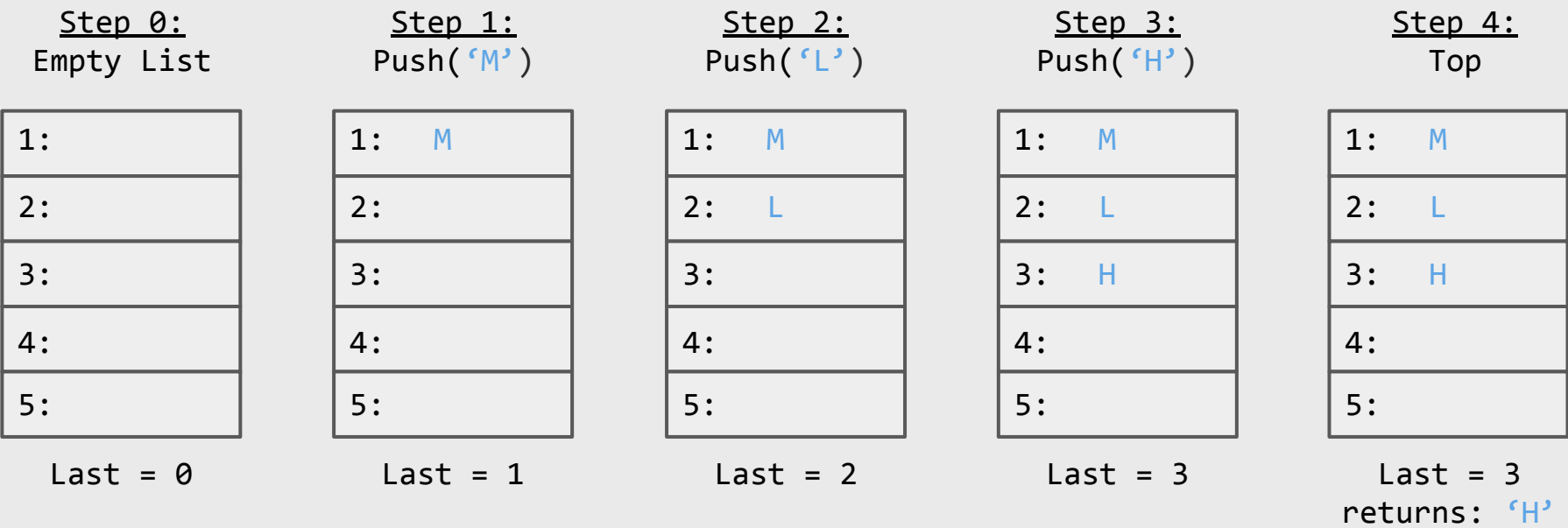
## A Stack is like a pile of dishes...

1. The pile starts out empty.
2. You add ("push") a new plate ("data") onto the Stack by placing it on the top of the pile.
3. To get plates ("data") out, you take the topmost one off the pile ("pop").
4. Our stack has a maximum height ("size") of 9 dishes.



# Pushing items onto the stack.

Here’s what should happen if we pushed the string “MLH” onto the stack.



The list starts out empty. Each time we push a character onto the stack, Last increments by 1.

# Popping items from the stack.

Here's what should happen if we popped 2 characters off our stack & then clear it.

Step 0:  
Starting List

1:	M
2:	L
3:	H
4:	
5:	

Last = 3

Step 1:  
Pop (Value)

1:	M
2:	L
3:	H
4:	
5:	

Last = 2  
returns: 'H'

Step 2:  
Pop (Value)

1:	M
2:	L
3:	H
4:	
5:	

Last = 1  
returns: 'L'

Step 3:  
Clear

1:	M
2:	L
3:	H
4:	
5:	

Last = 0

Note that **Pop** & **Clear** don't unset the array's elements, they just change the value of **Last**.

# Let's Try the Program

The program is supposed to push values onto the stack, print the stack, and pop some values off the stack.

- We can print the stack using the 'd' character.
- We can pop a value off the stack using the 'p' character.
- If we use any other character, it will be pushed onto the stack.

Consider the sequence: "M L H d p d"

This will:

1.) Push 'M'	4.) Print the stack "M L H"
2.) Push 'L'	5.) Pop the top item off the stack 'H'
3.) Push 'H'	6.) Print the stack "M L"

# Let's Try the Program

The program is supposed to push values onto the Stack. *What happens when you try to submit the lab?*

```

27
28   function Empty return Boolean is (Last < 1);
29
30   function Size return Integer is (Last);
31
32 end Stack;

```

☐ Test against custom input

Reset Prove Run Submit

Console Output:  
\$ gprbuild -q -P main -gnatwa  
Lab failed.

**Test Case #0 Fail**

```

$ ./main M L H d p d p d p d
raised CONSTRAINT_ERROR : stack.adb:20 index check failed
Process returned non-zero result: 1
Input:  M L H d p d p d p d
Expected Output:  [M, L, H] [M, L] [M] []
Received Output:
Status: Failed

```

**Test Case #1 Fail**

**Test Case #2 Fail**

As you can see, the program crashes.

SPARK Ada can actually warn us before we compile though!

# Using the Prover

If you press the Prove button, you'll see the SPARK Prover identifies 6 issues with our code that may cause bugs or unexpected behavior.

```
32
33
34  -- Main --
35  -----
36
37  for Arg in 1 .. Argument_Count loop
38    if Argument (Arg)'Length /= 1 then
39      Put_Line (Argument (Arg) & " is an invalid input to the stack.");
40    else
41      S := Argument (Arg)(Argument (Arg)'First);
42
43      if S = 'd' then
44        Debug;
45      elsif S = 'p' then
46        if not Stack.Empty then
47          Stack.Pop (S);
48        else
49          Put_Line ("Nothing to Pop, Stack is empty!");
50        end if;
```

☐ Test against custom input

Reset Prove Run Submit

Console Output:

```
$ gnatprove -P main --checks-as-errors --level=0 --no-axiom-guard
gnatprove: unproved check messages considered as errors
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stack.adb:20:12: medium: array index check might fail
stack.adb:31:17: medium: array index check might fail (e.g. when Last = 0)
stack.ads:5:19: medium: postcondition might fail, cannot prove Size = Size'Old + 1 (e.g. when
Last = 1)
stack.ads:12:19: medium: postcondition might fail, cannot prove Size = 0 (e.g. when Last = 1)
stack.ads:15:19: medium: postcondition might fail, cannot prove Top'Result = Tab(Last) (e.g.
when Last = 0 and Tab = (1 => 'NUL', others => 'SOH') and Top'Result = 'NUL')
stack.ads:15:36: medium: array index check might fail
Summary logged in ./gnatprove/gnatprove.out
```

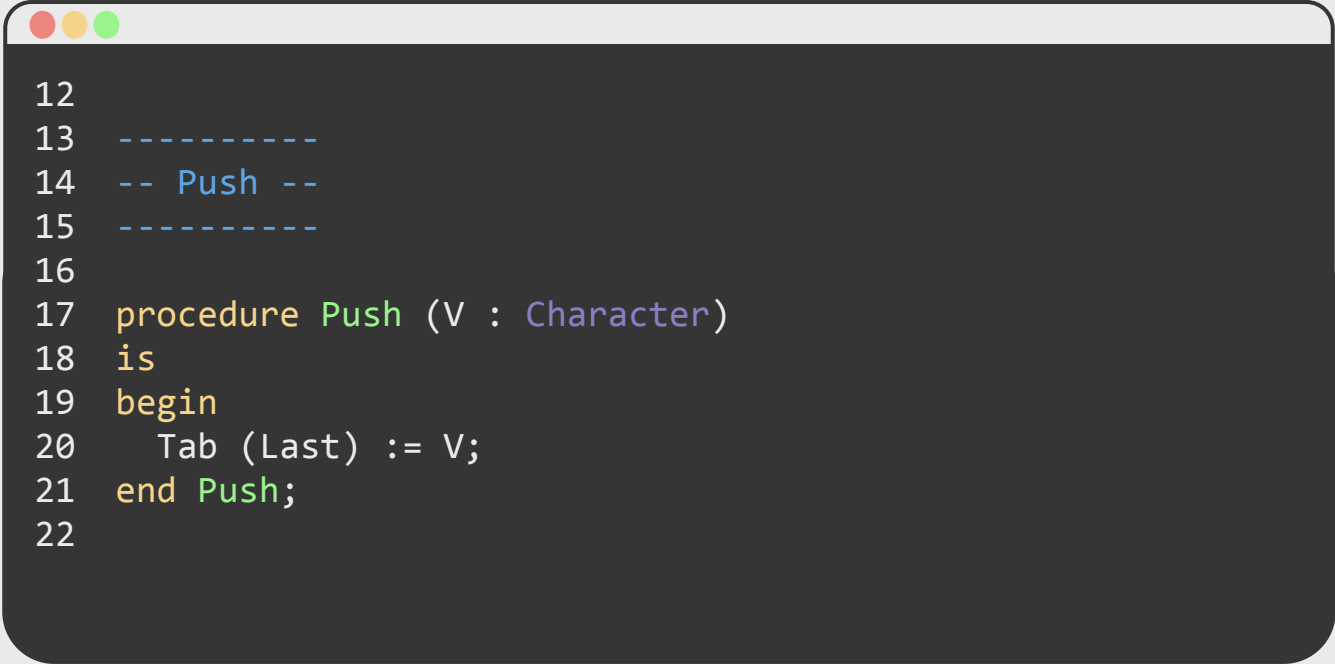
Click on any item in the list to go to the relevant section of the code.



# Fixing our first bug.

Click on the warning from line 20 of `stack.adb`.

This is the code that handles “pushing” values onto the stack.



```
12
13  -----
14  -- Push --
15  -----
16
17  procedure Push (V : Character)
18  is
19  begin
20      Tab (Last) := V;
21  end Push;
22
```

Can anyone spot the bug in this code?

**(Hint: It's a logical bug, not a syntax bug)**

# Fixing our first bug.

Let's draw out what happens when we **push** a value onto the stack.

Step 0:  
Empty Stack

1:	
2:	
3:	
4:	
5:	

Last = 0

Step 1:  
Push('a')

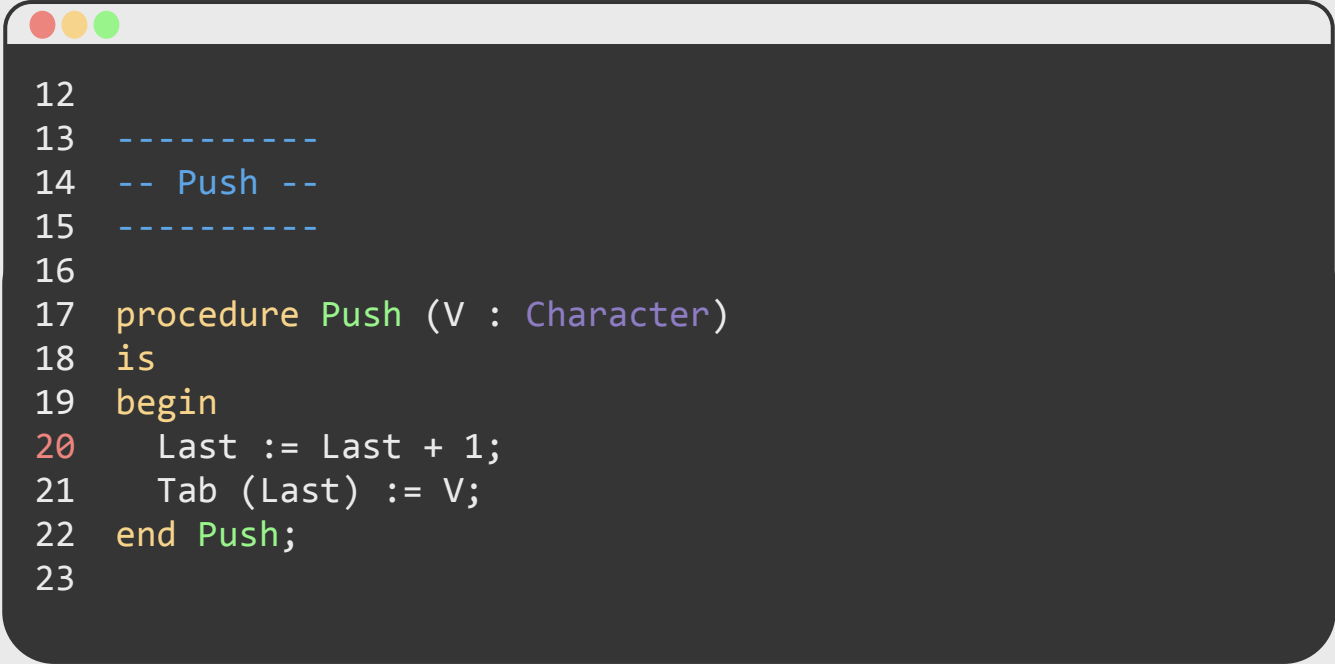
1:	
2:	
3:	
4:	
5:	

Last = 0

Look, **Last** never changes! We try to insert 'a' at index 0 of the **Tab** array, which raises an index out of bounds exception.

# Fixing our first bug.

To fix the bug, we need to increment `Last` by `1` before adding `V` to the `Tab` array.

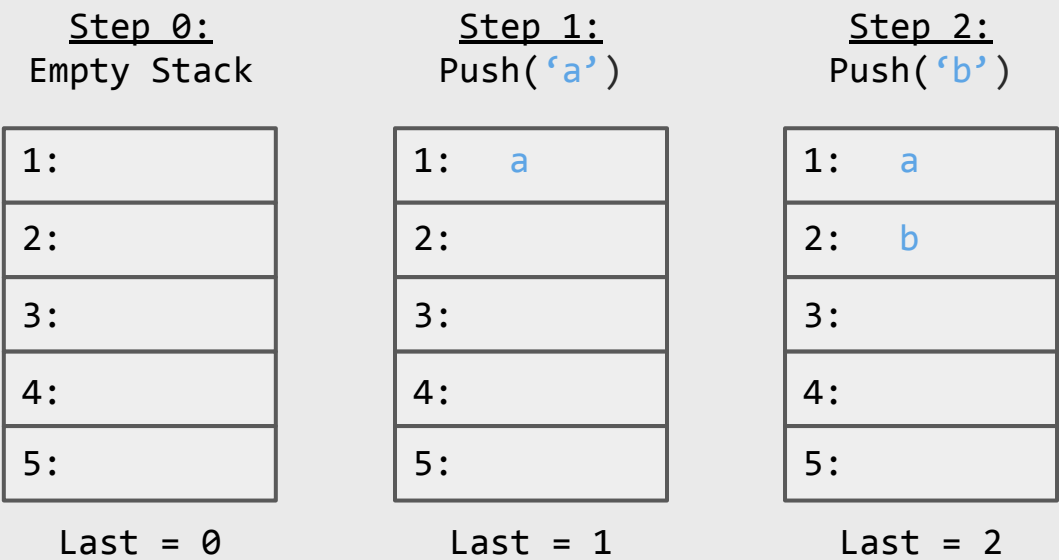


```
12
13  -----
14  -- Push --
15  -----
16
17  procedure Push (V : Character)
18  is
19  begin
20      Last := Last + 1;
21      Tab (Last) := V;
22  end Push;
23
```

*Going forward, line numbers will assume you've made the changes as we identify them.*

# Fixing our first bug.

Now let's draw out what happens when we **push** a few values onto the stack with our new code.



# Fixing our first bug.

If you run the program again, you can now push new values onto the stack without any issue.

```

31      Last := Last - 1;
32      V := Tab (Last);
33  end Pop;
34
35  -----
36  -- Top --
37  -----
38
39  function Top return Character
40  is
41  begin
42      return Tab (1);
43  end Top;
44
45  end Stack;

```

d a d b d

☒ Test against custom input

Reset Prove Run Submit

Console Output:

```

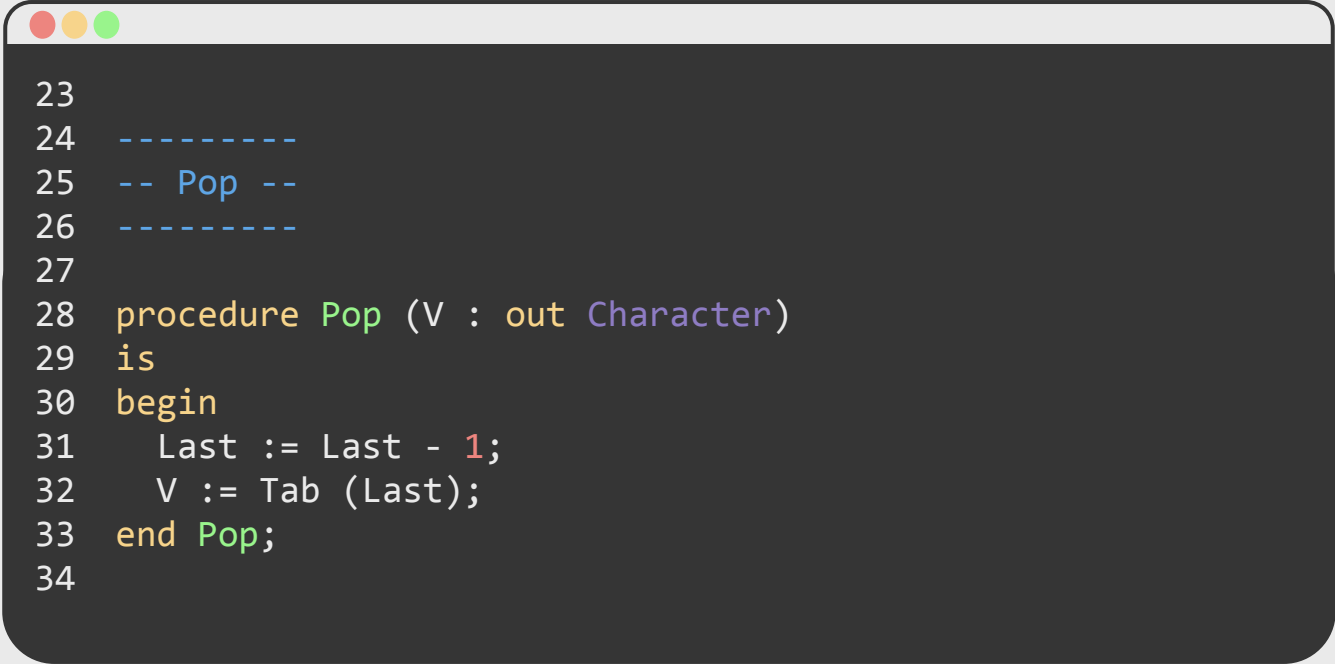
$ gprbuild -q -P main -gnatwa
$ ./main d a d b d
[]
[a]
[a, b]

```

*While you're at it, run the SPARK prover again to verify that our total warning count has gone down to 5.*

# Fixing the Pop procedure.

Next, let's look at the warning on line 32 of the `Pop` procedure.  
The warning is “array index check might fail”.

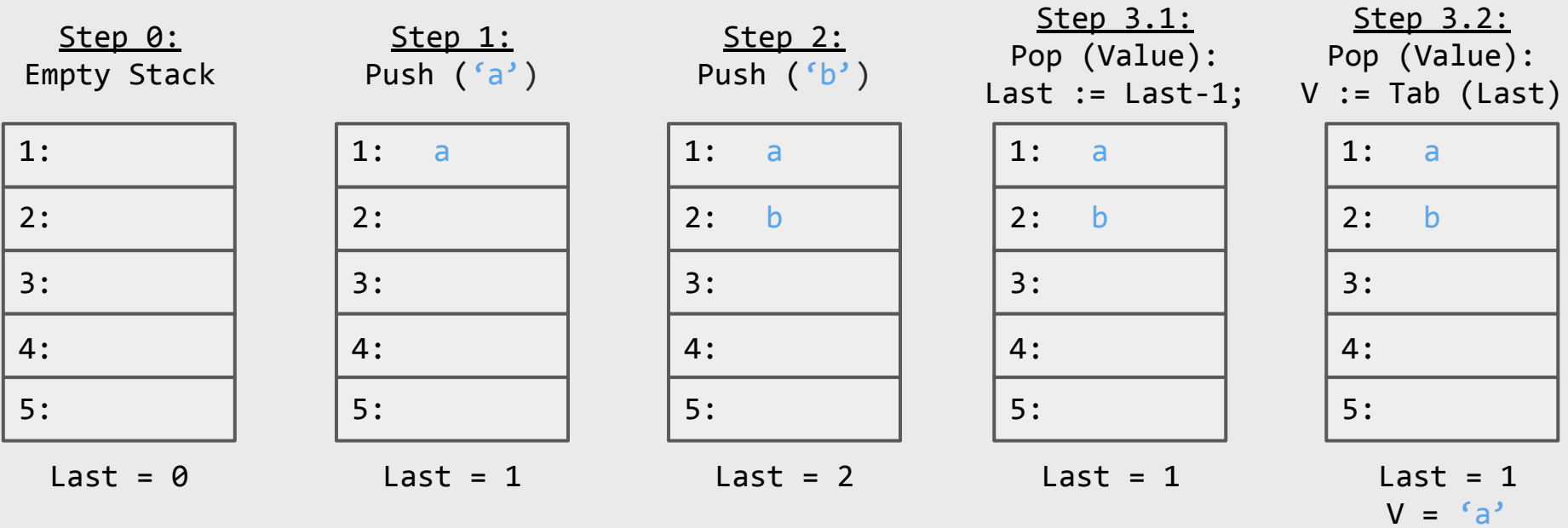


```
23
24  -----
25  -- Pop --
26  -----
27
28  procedure Pop (V : out Character)
29  is
30  begin
31      Last := Last - 1;
32      V := Tab (Last);
33  end Pop;
34
```

Can anyone spot the bug in this code?  
(**Hint:** *It's a logical bug, not a syntax bug*)

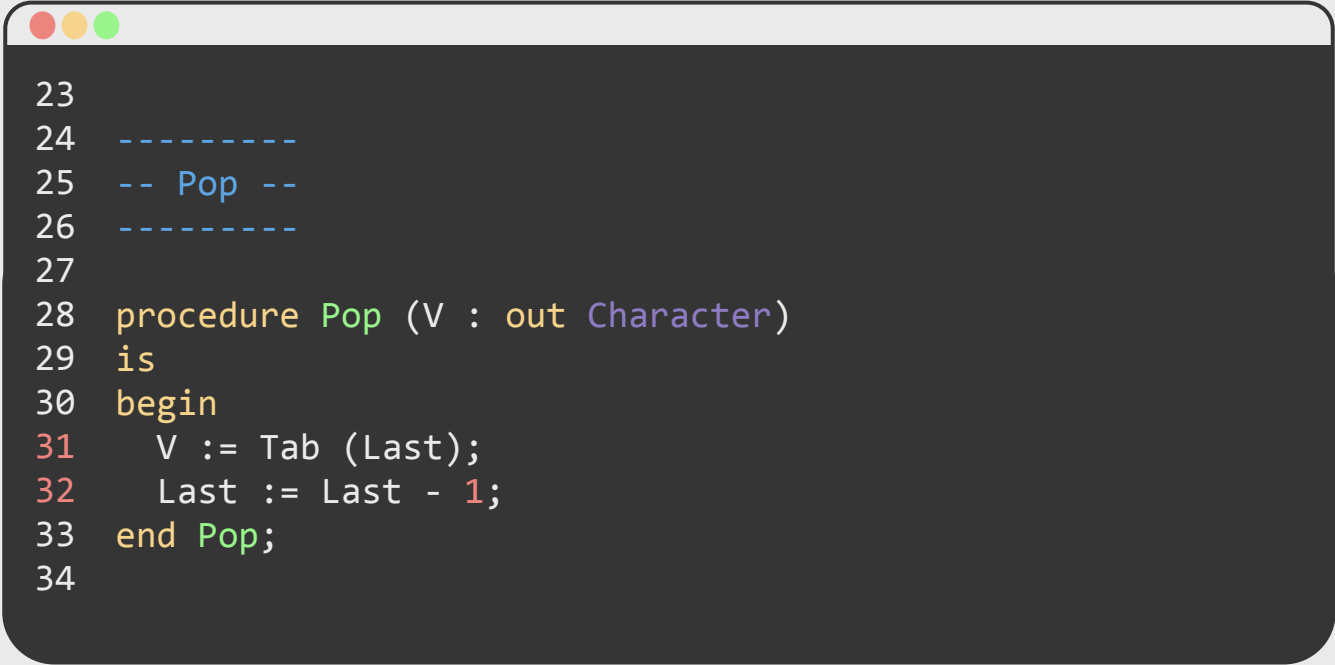
# Fixing the Pop procedure.

Let's draw out what happens when we **Push** a few values onto the stack & then **Pop** them off.



# Fixing the Pop procedure.

Swap lines 29 & 30 to set *V*'s value *before* we decrement *Last*.



```
23
24 -----
25 -- Pop --
26 -----
27
28 procedure Pop (V : out Character)
29 is
30 begin
>> 31   V := Tab (Last);
>> 32   Last := Last - 1;
33 end Pop;
34
```

What happens when you run the prover now?



# We're down to 3 warnings!

When we run the prover now, we only get 3 warnings. All three come from the Stack package's specification file ([stack.ads](#)).

```

33     end Pop;
34
35     -----
36     -- Top --
37     -----
38
39     function Top return Character
40     is
41     begin
42         return Tab (1);
43     end Top;
44
45 end Stack;

```

☐ Test against custom input

Reset Prove Run Submit

Console Output:

```

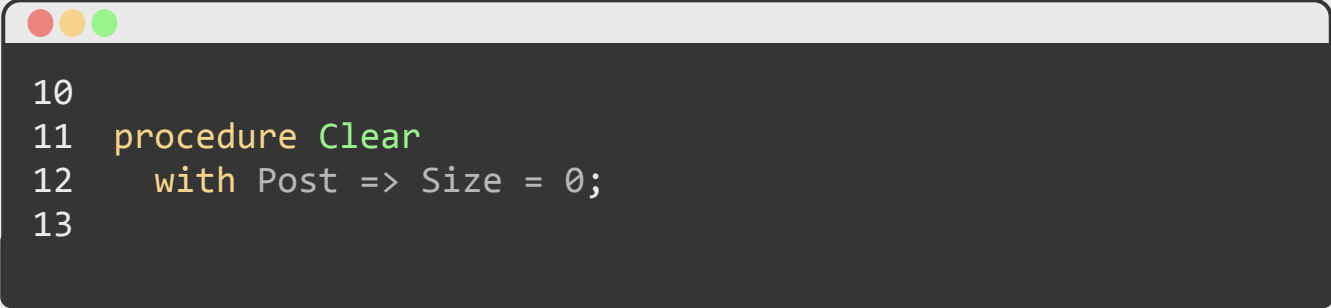
$ gnatprove -P main --checks-as-errors --level=0 --no-axiom-guard
gnatprove: unproved check messages considered as errors
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stack.ads:12:19: medium: postcondition might fail, cannot prove Size = 0 (e.g. when Last = 1)
stack.ads:15:19: medium: postcondition might fail, cannot prove Top'Result = Tab(Last) (e.g.
when Last = 0 and Tab = (1 => 'NUL', others => 'SOH') and Top'Result = 'NUL')
stack.ads:15:36: medium: array index check might fail
Summary logged in ./gnatprove/gnatprove.out

```

# Fixing the Clear procedure.

Click on the warning from line 12 in `stack.ads`.

`stack.ads:`



```
10
11 procedure Clear
12     with Post => Size = 0;
13
```

The SPARK Prover can't assert that `Size` will equal 0 after the `Clear` procedure has been run.

*Can anyone figure out why?*

# Fixing the Clear procedure.

The code on line 11 of `stack.ads` corresponds to the code on line 7 of `stack.adb`.

`stack.ads:`

```
10
11  procedure Clear
12    with Post => Size = 0;
13
```

`stack.adb:`

```
03  -----
04  -- Clear --
05  -----
06
07  procedure Clear
08  is
09  begin
10    Last := Tab'First;
11  end Clear;
```

# Fixing the Clear procedure.

The Size function is also defined in `stack.ads` on line 30.

stack.ads:

```
10
11 procedure Clear
12   with Post => Size = 0;
13
```

stack.ads:

```
29
30 function Size return Integer is (Last);
31
```

It simply returns the value of `Last`.  
So, when the stack is cleared, it should return 0.

# Fixing the Clear procedure.

Let's draw out what happens when we **Push** a few values onto the stack & then **Clear** it.

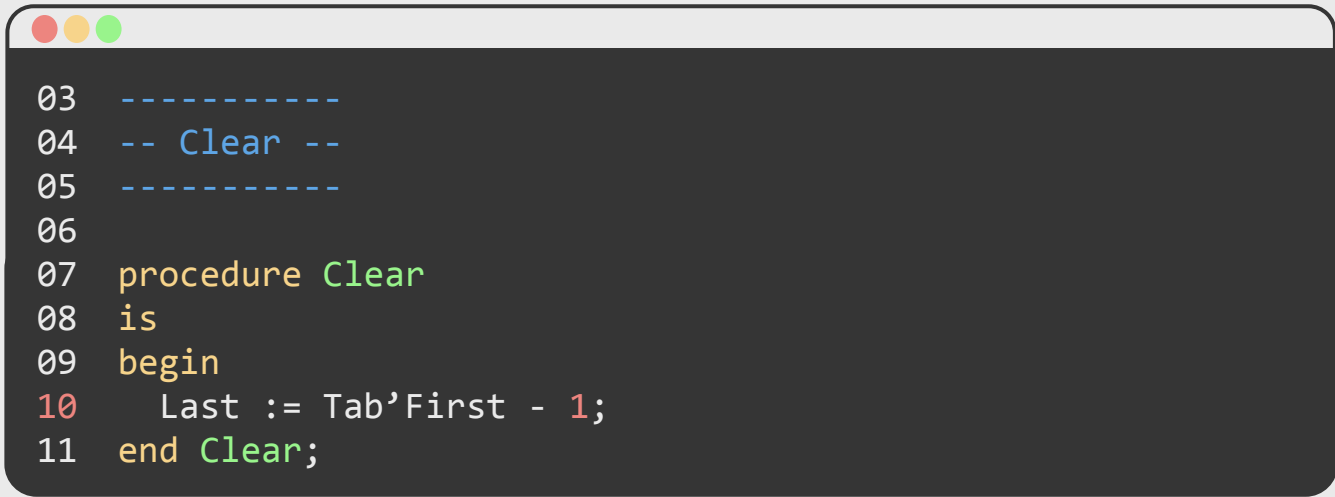
Step 0: Empty List	Step 1: Push ( 'a' )	Step 2: Push ( 'b' )	Step 3: Push ( 'c' )	Step 4: Clear
1:	1: a	1: a	1: a	1: a
2:	2:	2: b	2: b	2: b
3:	3:	3:	3: c	3: c
4:	4:	4:	4:	4:
5:	5:	5:	5:	5:
Last = 0	Last = 1	Last = 2	Last = 3	Last = 1

In Step 4, **Last** is set to **Tab'First**,  
which is the first index of the array - **1** not **0**!

# Fixing the Clear procedure.

The code on line 11 of `stack.ads` corresponds to the code on line 7 of `stack.adb`.

stack.adb:



```
03  -----
04  -- Clear --
05  -----
06
07  procedure Clear
08  is
09  begin
>> 10    Last := Tab'First - 1;
11  end Clear;
```

Now `Last` will be set to 0 when we `Clear` the stack.

# We're down to our last warnings!

When we run the prover now, we only get 2 warnings. Both point to line 15 of the specification - [stack.ads](#).

```
38
39     function Top return Character
40     is
41     begin
42         return Tab (1);
43     end Top;
44
45 end Stack;
```

☐ Test against custom input

Reset

Prove

Run

Submit

Console Output:

```
$ gnatprove -P main --checks-as-errors --level=0 --no-axiom-guard
gnatprove: unproved check messages considered as errors
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stack.ads:15:19: medium: postcondition might fail, cannot prove Top'Result = Tab(Last) (e.g.
when Last = 0 and Tab = (1 => 'NUL', others => 'SOH') and Top'Result = 'NUL')
stack.ads:15:36: medium: array index check might fail
Summary logged in ./gnatprove/gnatprove.out
```

# Fixing the Top function.

Top returns the item at the top of the stack without popping it.

*Can anyone spot the bug?*

stack.ads:

```
13
14 procedure Top
15     with Post => Top'Result = Tab>Last);
16
```

stack.adb:

```
35 -----
36 -- Top --
37 -----
38
39 function Top return Character
40 is
41 begin
42     return Tab (1);
43 end Top;
```



# Fixing the Top function.

Although **1** is technically the top (first element) of the **Tab** array, the top of the stack is actually **Last**.

stack.adb:

```
35  -----
36  -- Top --
37  -----
38
39  function Top return Character
40  is
41  begin
>> 42      return Tab (Last);
43  end Top;
```

*What happens when you run the prover now?  
We should be all out of warnings, right?*

# Surprise! We've got one more warning to fix.

The SPARK Prover recognizes that if `Last = 0`, `Top` would throw an index out of bounds error.

```
37
38
39   function Top return Character
40   is
41   begin
42     return Tab (Last);
43   end Top;
44
45 end Stack;
```

☐ Test against custom input

Reset Prove Run Submit

Console Output:

```
$ gnatprove -P main --checks-as-errors --level=0 --no-axiom-guard
gnatprove: unproved check messages considered as errors
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
stack.adb:42:19: medium: array index check might fail [possible explanation: subprogram at
stack.ads:14 should mention Last in a precondition]
Summary logged in ./gnatprove/gnatprove.out
```

**Hint:** If `Post` specifies conditions after a function runs, how would you specify conditions before?

# Fixing the final warning.

We can use **Pre** conditions to specify that **Top** can't be called when the stack is empty.

stack.ads:

```
13
14 function Top return Character
15     with Post => Top'Result = Tab(Last),
16     Pre  => not Empty;
17
```

stack.ads:

```
28
29 function Empty return Boolean is (Last < 1);
30
```

# Our code is bug-free!

Thanks to the SPARK Ada Prover, our code is proven to not raise exceptions, use uninitialized variables, and more!

```
25  -- The stack. We push and pop pointers to Values.  
26  
27  function Full return Boolean is (Last = Max_Size);  
28  
29  function Empty return Boolean is (Last < 1);  
30  
31  function Size return Integer is (Last);  
32  
33  end Stack;
```

☐ Test against custom input

Reset

Prove

Run

Submit

Console Output:

```
$ gnatprove -P main --checks-as-errors --level=0 --no-axiom-guard  
Phase 1 of 2: generation of Global contracts ...  
Phase 2 of 2: flow analysis and proof ...  
Summary logged in ./gnatprove/gnatprove.out
```

Run the prover on your code to see for yourself.

*Raise your hand if you need help or are still getting warnings!*

# Try submitting the program yourself!

Now that our program works,  
Try submitting it by clicking “Submit”.

```
29   function Empty return Boolean is (Last < 1);  
30  
31   function Size return Integer is (Last);  
32  
33 end Stack;
```

☐ Test against custom input

Reset

Prove

Run

Submit

Console Output:

```
$ gprbuild -q -P main -gnatwa  
Lab completed successfully.
```

Test Case #0 *Pass*

+


Test Case #1 *Pass*

+

Test Case #2 *Pass*

+

# Table of Contents

1. What are Ada & SPARK?
2. Intro to SPARK Ada
3. Set Up your Environment
4. Let's Build a Stack!
-  5. Review & Quiz
6. Next Steps



## *Let's recap quickly...*

- 1 SPARK Ada makes it easy for you to write bug-free programs!
- 2 Ada is a modern language that you can use on a variety of systems for a variety of purposes.
- 3 Employers are looking for developers that can engineer robust software & Ada enables that!


# What did you learn today?

We created a fun quiz to test your knowledge and see what you learned from this workshop.

[\*\*http://mlhlocal.host/quiz\*\*](http://mlhlocal.host/quiz)



# Table of Contents

1. What are Ada & SPARK?
2. Intro to SPARK Ada
3. Set Up your Environment
4. Let's Build a Stack!
5. Review & Quiz
-  6. Next Steps

The AdaCore logo is displayed inside a black rectangular border. The text "AdaCore" is in a blue, sans-serif font, with "Ada" in a slightly larger size than "Core".

## *Where to go from here...*

1

**Explore the differences between Ada, C++, & Java.**  
<http://mlhlocal.host/adacore-ada-vs-java>

2

**Keep learning Ada & SPARK.**  
<http://mlhlocal.host/learn-ada>

3

**Complete the additional practice problems.**  
*These problems & way more are in your email!*

# Keep Learning Ada and SPARK

## Introduction to Ada and Introduction to SPARK

Learn more about Ada and SPARK with the [Introduction to Ada](#) and [Introduction to SPARK](#) courses on [learn.adacore.com](https://learn.adacore.com). You can also try your new Ada and SPARK skills by visiting the [labs](#) section.

The screenshot shows a code editor window titled 'learn.adb' containing the following Ada code:

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Learn is
4     subtype Alphabet is Character range 'A' .. 'Z';
5
6 begin
7
8     Put_Line ("Learning Ada from " & Alphabet'First & " to " & Alphabet'Last);
9
10
11 end Learn;
```

Below the code editor are two orange buttons: 'Reset' and 'Run'.

Below the buttons are two blue boxes representing course options:

- COURSES Introduction to Ada**  
This course will teach you the basics of the Ada programming language and is intended for those who already have a basic understanding of programming techniques. You will learn how to apply those techniques to programming in Ada.
- COURSES Introduction to SPARK**  
This tutorial is an interactive introduction to the SPARK programming language and its formal verification tools. You will learn the difference between Ada and SPARK and how to use the various analysis tools that come with SPARK.

Navigation arrows (left and right) are visible on the sides of the course boxes, and a series of small dots at the bottom indicates the current selection.

# Learning shouldn't stop when the workshop ends...



**Check your email for access to:**

- These workshop slides
- Practice problems to keep learning
- Deeper dives into key topics
- Instructions to join the community
- More opportunities from MLH!

Workshop

# Bug-free Coding with SPARK Ada