Seattle Pacific University
Department of Engineering and Computer Science
Concepts of Programming Languages
Fall 2020

Implementation of Data Types in Programming Languages

Jonathan Hacker
Katie Honsinger
Andrew Mclain
Chris Moroney

Ada

Summary

Ada is a statically-typed, strongly-typed, imperative, object-oriented programming language [9]. Ada is primarily used in embedded and real-time systems, particularly in the aerospace and defense domains. First, we will be discussing **strong typing** in Ada, and then we will discuss **array types** and **string types**. We will be using the latest stable release, Ada 2012 TC1, and compiling with GNAT 7.5.0.

Strong typing

Ada is a strongly typed language, which means it enforces stricter typing rules at compilation than most other languages. For example, the following code will not compile because Ada does not do implicit type conversion in assignment operations [1].

Figure 1

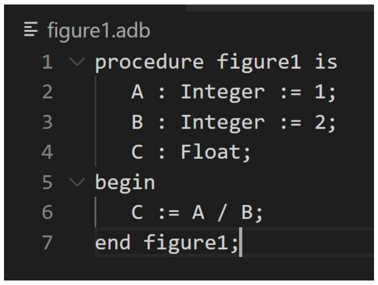
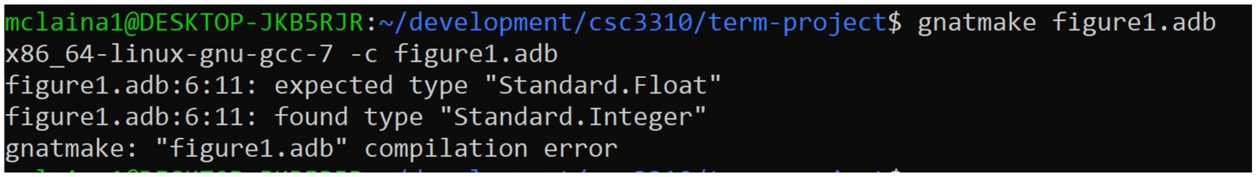


figure1.adb

```
procedure figure1 is
    A : Integer := 1;
    B : Integer := 2;
    C : Float;
begin
    C := A / B;
end figure1;
```

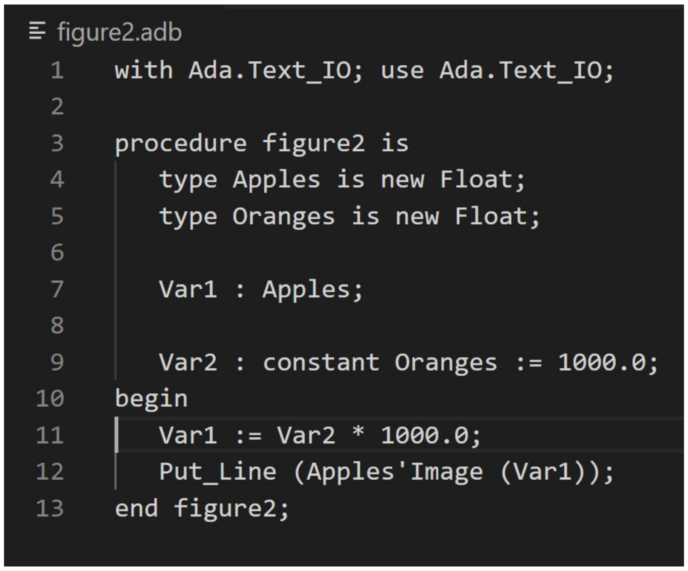


```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure1.adb
x86_64-linux-gnu-gcc-7 -c figure1.adb
figure1.adb:6:11: expected type "Standard.Float"
figure1.adb:6:11: found type "Standard.Integer"
gnatmake: "figure1.adb" compilation error
```

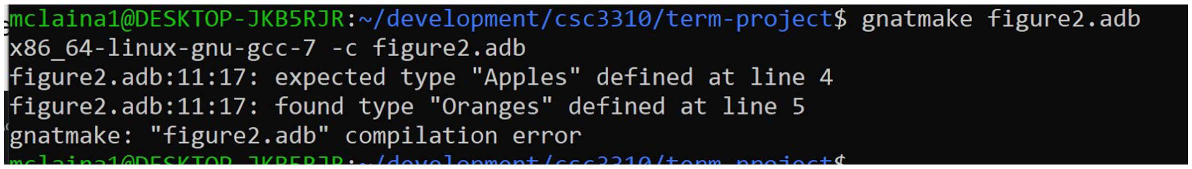
In order for the program to compile properly, the assignment to C must be changed to `C := Float (A) / Float (B);` This helps prevent unexpected behavior that may be caused by implicit type conversion, such as imprecise results due to rounding in integer division.

Ada's strong typing even extends to user-defined types which are defined using "built-in" primitive types. The code below will not compile because `Var1` is of the "Apples" data type and `Var2` is of the "Oranges" data type, even though the underlying primitive data type, Float, is the same.

Figure 2



```
figure2.adb
 1  with Ada.Text_IO; use Ada.Text_IO;
 2
 3  procedure figure2 is
 4     type Apples is new Float;
 5     type Oranges is new Float;
 6
 7     Var1 : Apples;
 8
 9     Var2 : constant Oranges := 1000.0;
10  begin
11     Var1 := Var2 * 1000.0;
12     Put_Line (Apples'Image (Var1));
13  end figure2;
```

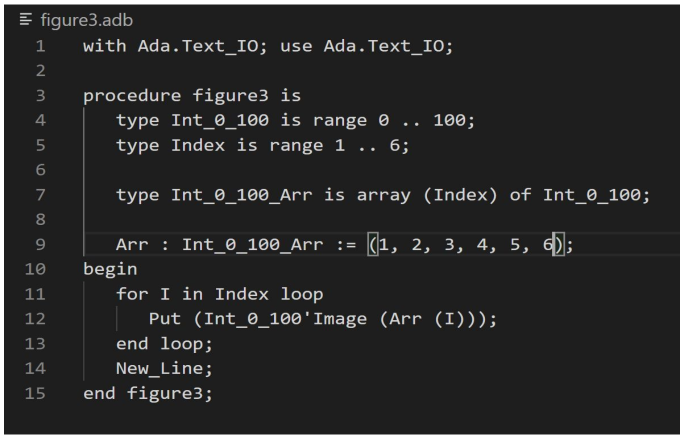


```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure2.adb
x86_64-linux-gnu-gcc-7 -c figure2.adb
figure2.adb:11:17: expected type "Apples" defined at line 4
figure2.adb:11:17: found type "Oranges" defined at line 5
gnatmake: "figure2.adb" compilation error
```

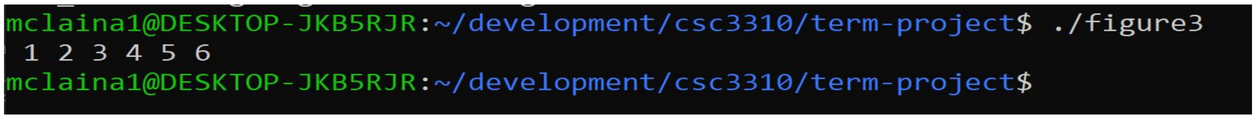
Array types

Below is an example of array declaration in Ada. Note how one can define a new array type by its index type and the type of its elements. Instances of this array type must conform to those specifications.

Figure3



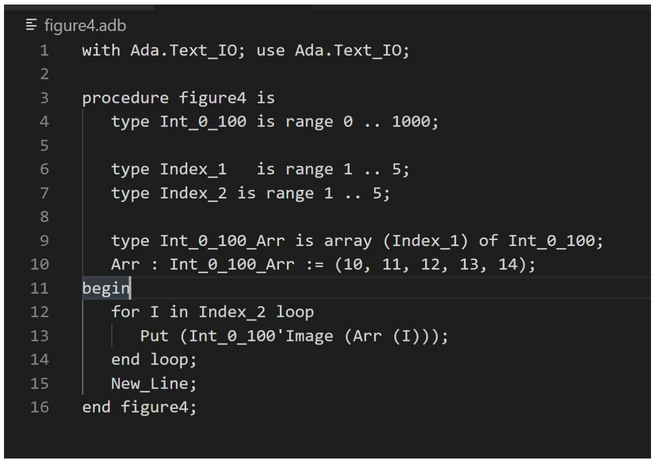
```
figure3.adb
with Ada.Text_IO; use Ada.Text_IO;

procedure figure3 is
   type Int_0_100 is range 0 .. 100;
   type Index is range 1 .. 6;

   type Int_0_100_Arr is array (Index) of Int_0_100;

   Arr : Int_0_100_Arr := (1, 2, 3, 4, 5, 6);
begin
   for I in Index loop
      Put (Int_0_100'Image (Arr (I)));
   end loop;
   New_Line;
end figure3;
```



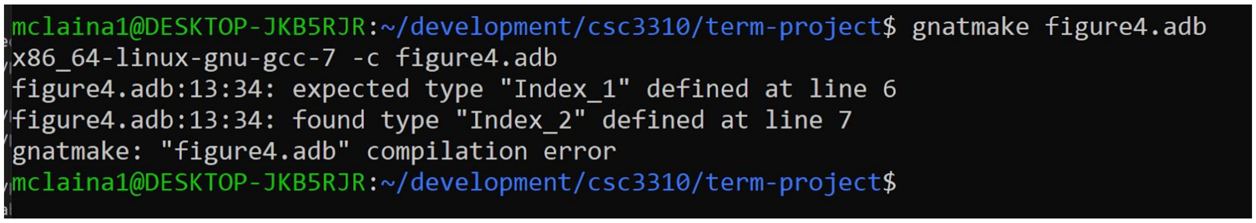
```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure3
 1 2 3 4 5 6
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

Array indexing in Ada is also strongly typed. For example, the following code will not compile:

Figure 4



```
figure4.adb
with Ada.Text_IO; use Ada.Text_IO;

procedure figure4 is
   type Int_0_100 is range 0 .. 1000;

   type Index_1   is range 1 .. 5;
   type Index_2 is range 1 .. 5;

   type Int_0_100_Arr is array (Index_1) of Int_0_100;
   Arr : Int_0_100_Arr := (10, 11, 12, 13, 14);
begin
   for I in Index_2 loop
      Put (Int_0_100'Image (Arr (I)));
   end loop;
   New_Line;
end figure4;
```

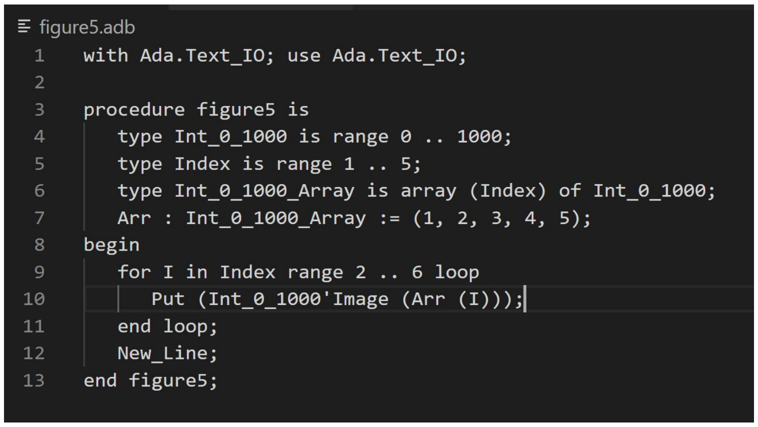


```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure4.adb
x86_64-linux-gnu-gcc-7 -c figure4.adb
figure4.adb:13:34: expected type "Index_1" defined at line 6
figure4.adb:13:34: found type "Index_2" defined at line 7
gnatmake: "figure4.adb" compilation error
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

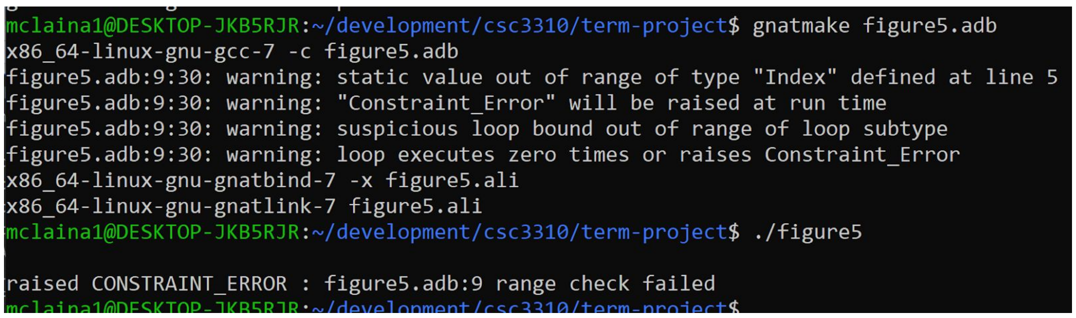
As you can see, this is because we are trying to access `Arr` with the type `Index_2` when the array's bounds were set with the type `Index_1`.

Ada is a memory safe language, so one cannot access elements outside the bounds of the array. If one attempts to do so, it will result in a runtime error. Below is an example of a program which will cause such an error:

Figure 5



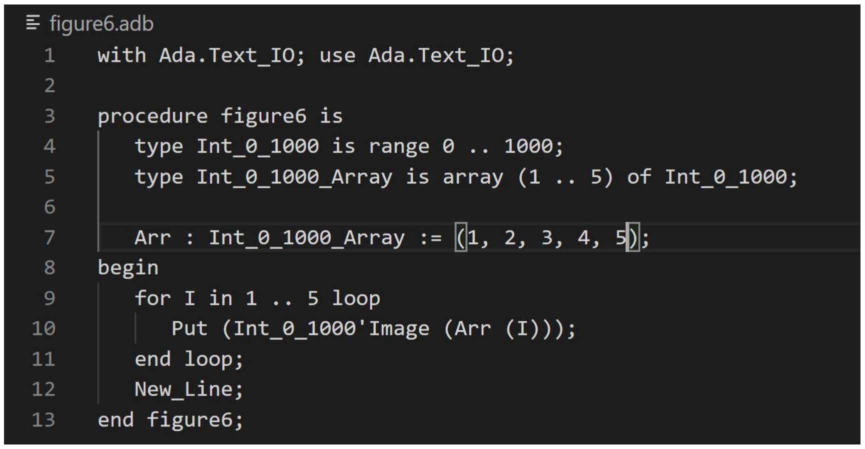
```
figure5.adb
with Ada.Text_IO; use Ada.Text_IO;

procedure figure5 is
   type Int_0_1000 is range 0 .. 1000;
   type Index is range 1 .. 5;
   type Int_0_1000_Array is array (Index) of Int_0_1000;
   Arr : Int_0_1000_Array := (1, 2, 3, 4, 5);
begin
   for I in Index range 2 .. 6 loop
      Put (Int_0_1000'Image (Arr (I)));
   end loop;
   New_Line;
end figure5;
```



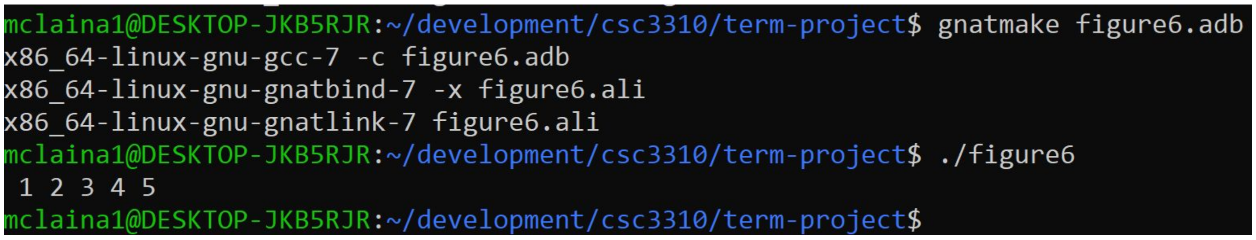
```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure5.adb
x86_64-linux-gnu-gcc-7 -c figure5.adb
figure5.adb:9:30: warning: static value out of range of type "Index" defined at line 5
figure5.adb:9:30: warning: "Constraint_Error" will be raised at run time
figure5.adb:9:30: warning: suspicious loop bound out of range of loop subtype
figure5.adb:9:30: warning: loop executes zero times or raises Constraint_Error
x86_64-linux-gnu-gnatbind-7 -x figure5.ali
x86_64-linux-gnu-gnatlink-7 figure5.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure5

raised CONSTRAINT_ERROR : figure5.adb:9 range check failed
```

The default index type for arrays in Ada is the Integer built-in type. This prevents one from needing to define an index type in order to initialize an array, as demonstrated in the program below:

Figure 6



```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure figure6 is
   type Int_0_1000 is range 0 .. 1000;
   type Int_0_1000_Array is array (1 .. 5) of Int_0_1000;

   Arr : Int_0_1000_Array := (1, 2, 3, 4, 5);
begin
   for I in 1 .. 5 loop
      Put (Int_0_1000'Image (Arr (I)));
   end loop;
   New_Line;
end figure6;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure6.adb
x86_64-linux-gnu-gcc-7 -c figure6.adb
x86_64-linux-gnu-gnatbind-7 -x figure6.ali
x86_64-linux-gnu-gnatlink-7 figure6.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure6
 1 2 3 4 5
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

Arrays in Ada also have the range attributes `‘Range`, `‘First`, and `‘Last` which allows one to iterate over the range of an array without hard-coding the bounds in the loop:

Figure 7

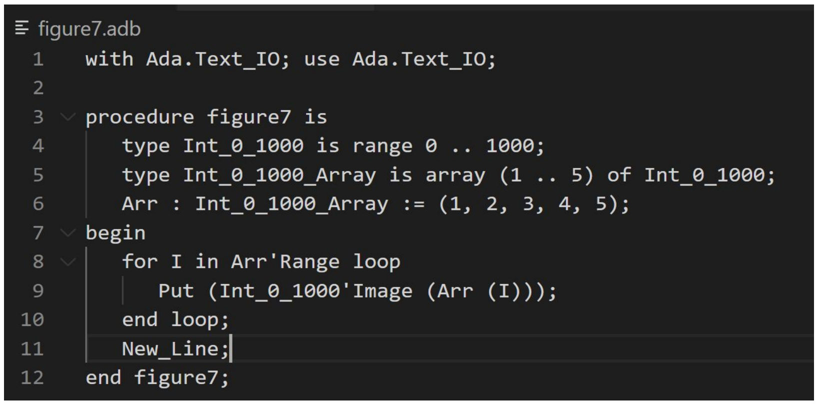


figure7.adb
```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure figure7 is
   type Int_0_1000 is range 0 .. 1000;
   type Int_0_1000_Array is array (1 .. 5) of Int_0_1000;
   Arr : Int_0_1000_Array := (1, 2, 3, 4, 5);
begin
   for I in Arr'Range loop
      Put (Int_0_1000'Image (Arr (I)));
   end loop;
   New_Line;
end figure7;
```

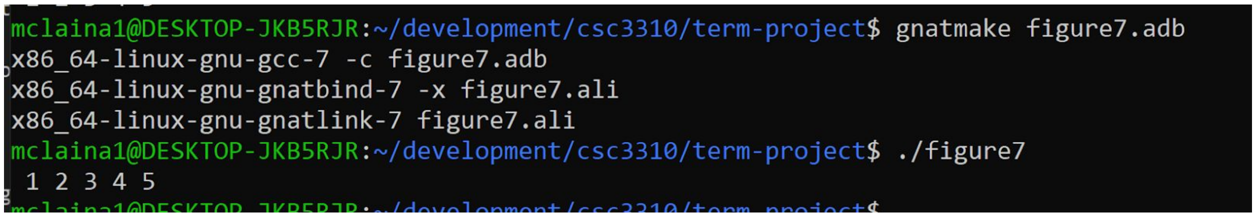

```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure7.adb
x86_64-linux-gnu-gcc-7 -c figure7.adb
x86_64-linux-gnu-gnatbind-7 -x figure7.ali
x86_64-linux-gnu-gnatlink-7 figure7.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure7
 1 2 3 4 5
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

Figure 8

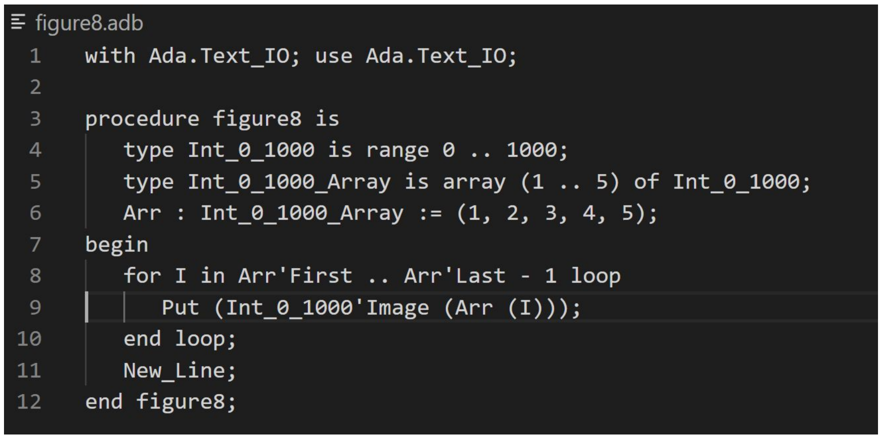
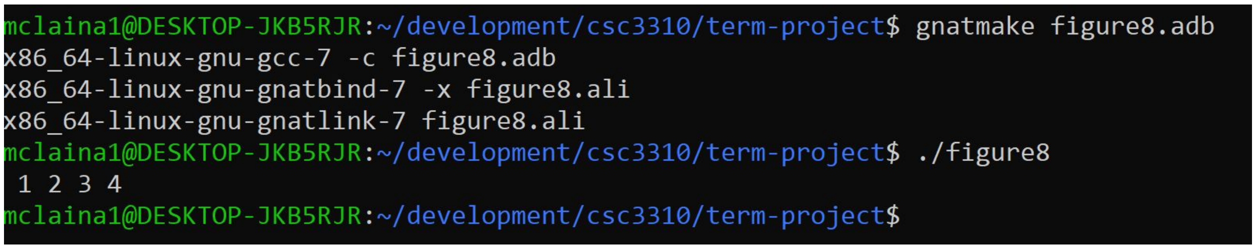


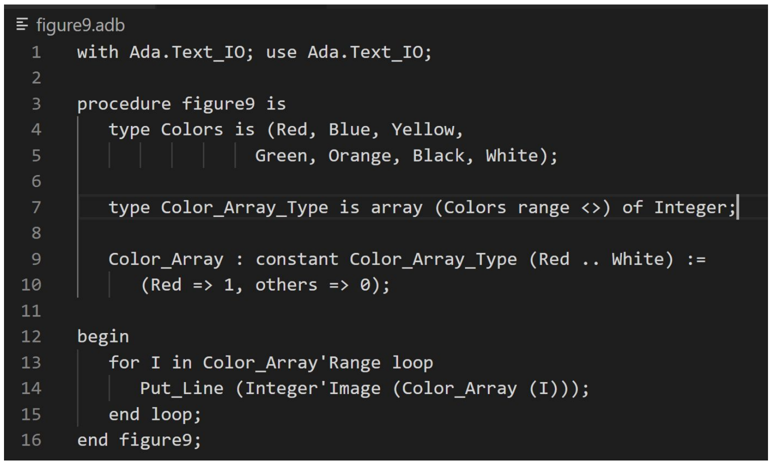
figure8.adb
```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure figure8 is
   type Int_0_1000 is range 0 .. 1000;
   type Int_0_1000_Array is array (1 .. 5) of Int_0_1000;
   Arr : Int_0_1000_Array := (1, 2, 3, 4, 5);
begin
   for I in Arr'First .. Arr'Last - 1 loop
      Put (Int_0_1000'Image (Arr (I)));
   end loop;
   New_Line;
end figure8;
```


```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure8.adb
x86_64-linux-gnu-gcc-7 -c figure8.adb
x86_64-linux-gnu-gnatbind-7 -x figure8.ali
x86_64-linux-gnu-gnatlink-7 figure8.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure8
 1 2 3 4
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

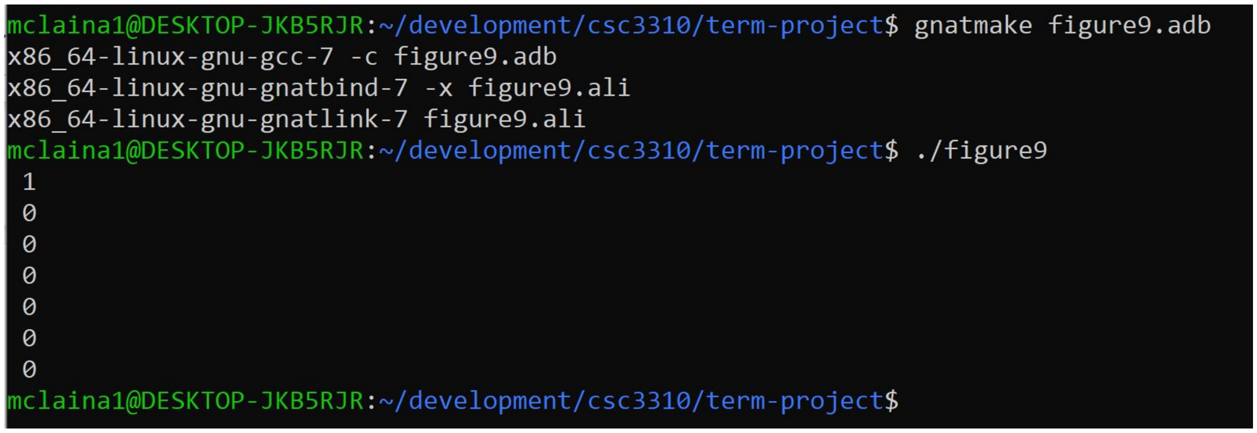
Unconstrained arrays

Ada also allows one to declare an unconstrained array type. Unconstrained array types do not have fixed bounds, so the bounds of an instance of an unconstrained array must be defined when the instance is defined. The "<>" symbol is used as a sort of wildcard or placeholder in Ada. Below is an example of an unconstrained array:

Figure 9



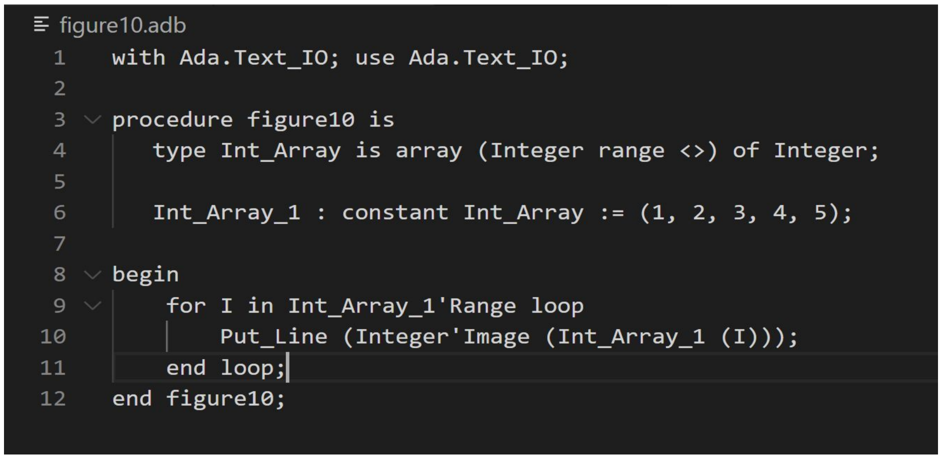
```
figure9.adb
with Ada.Text_IO; use Ada.Text_IO;

procedure figure9 is
   type Colors is (Red, Blue, Yellow,
                   Green, Orange, Black, White);

   type Color_Array_Type is array (Colors range <>) of Integer;

   Color_Array : constant Color_Array_Type (Red .. White) :=
      (Red => 1, others => 0);

begin
   for I in Color_Array'Range loop
      Put_Line (Integer'Image (Color_Array (I)));
   end loop;
end figure9;
```



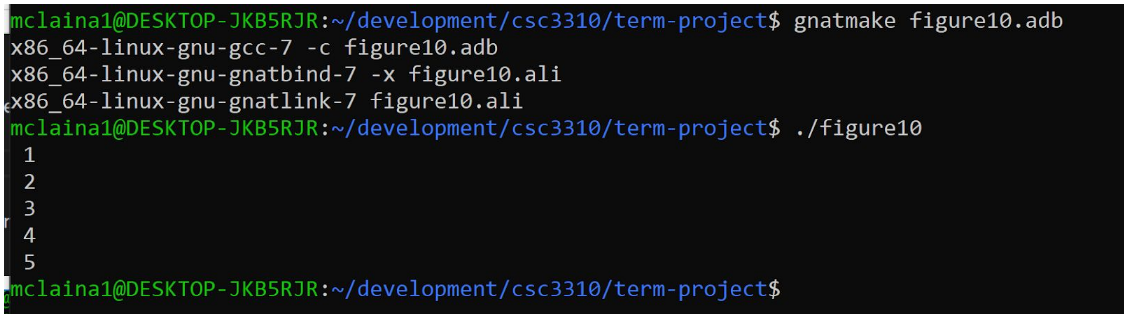
```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure9.adb
x86_64-linux-gnu-gcc-7 -c figure9.adb
x86_64-linux-gnu-gnatbind-7 -x figure9.ali
x86_64-linux-gnu-gnatlink-7 figure9.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure9
 1
 0
 0
 0
 0
 0
 0
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

Arrays initialized with values will have their bounds automatically computed from the total number of elements, which saves the programmer from explicitly stating the bounds at initialization:

Figure 10



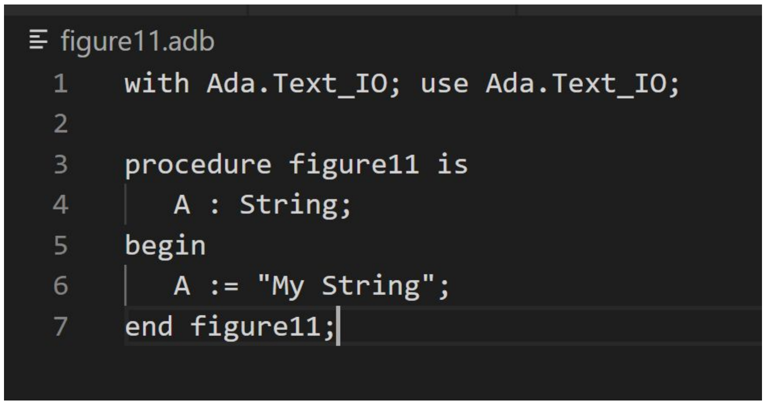
```
figure10.adb
with Ada.Text_IO; use Ada.Text_IO;

procedure figure10 is
   type Int_Array is array (Integer range <>) of Integer;

   Int_Array_1 : constant Int_Array := (1, 2, 3, 4, 5);

begin
    for I in Int_Array_1'Range loop
        Put_Line (Integer'Image (Int_Array_1 (I)));
    end loop;
end figure10;
```



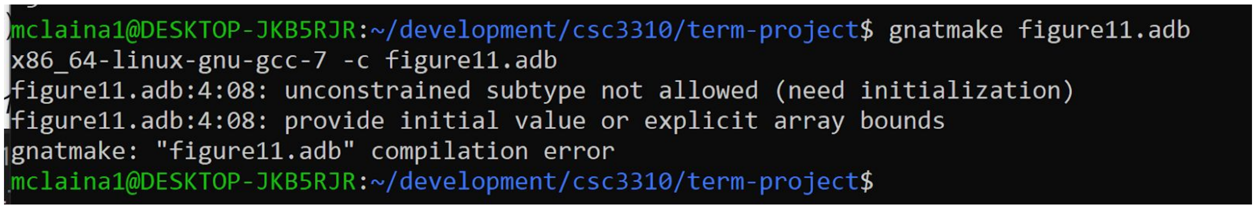
```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure10.adb
x86_64-linux-gnu-gcc-7 -c figure10.adb
x86_64-linux-gnu-gnatbind-7 -x figure10.ali
x86_64-linux-gnu-gnatlink-7 figure10.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure10
 1
 2
 3
 4
 5
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

The values in the array must be declared at initialization in order for this to work. An instance of an array cannot be created without specifying the bounds in some way - either explicitly when defining the array type, or implicitly when initializing the values of an array instance. For example, this program would generate an error:

Figure 11



```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure11 is
   A : String;
begin
   A := "My String";
end figure11;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure11.adb
x86_64-linux-gnu-gcc-7 -c figure11.adb
figure11.adb:4:08: unconstrained subtype not allowed (need initialization)
figure11.adb:4:08: provide initial value or explicit array bounds
gnatmake: "figure11.adb" compilation error
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

The size of an instance of an array is also statically bound, so an array cannot change size during run-time. For example, this program would generate a run-time error:

Figure 12

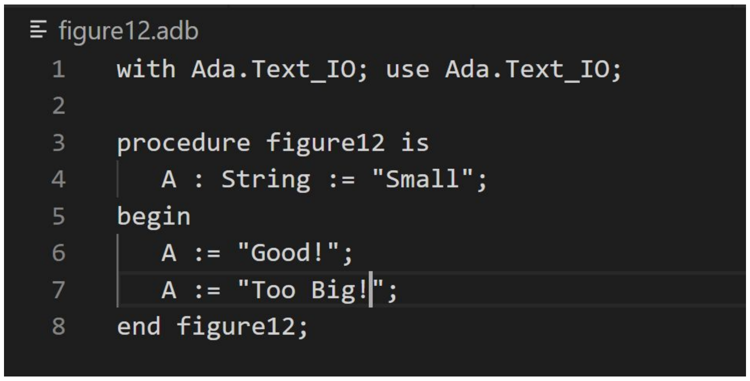
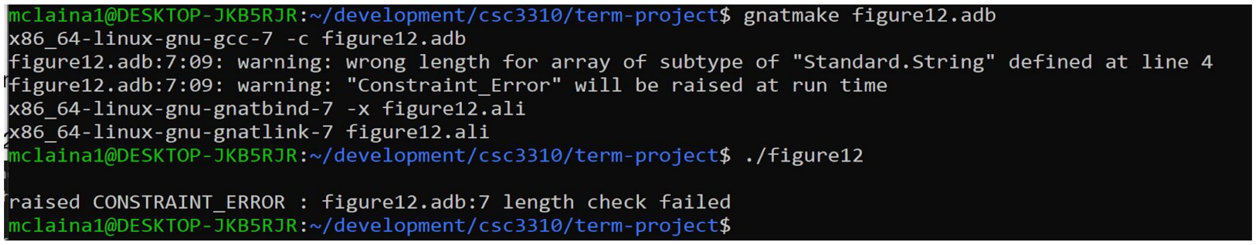


figure12.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure figure12 is
   A : String := "Small";
begin
   A := "Good!";
   A := "Too Big!";
end figure12;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure12.adb
x86_64-linux-gnu-gcc-7 -c figure12.adb
figure12.adb:7:09: warning: wrong length for array of subtype of "Standard.String" defined at line 4
figure12.adb:7:09: warning: "Constraint_Error" will be raised at run time
x86_64-linux-gnu-gnatbind-7 -x figure12.ali
x86_64-linux-gnu-gnatlink-7 figure12.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure12

raised CONSTRAINT_ERROR : figure12.adb:7 length check failed
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

Character string types

Ada’s built-in String type

The built-in String type in Ada is just a simple fixed-length array defined as followed: `type String is array (Positive range <>) of Character;`
This is an example of a **static length string** [8], because the length of the string cannot change during run time. As you can see, the array bounds in this definition are undefined, so the range of the String instance must be specified when it is created.
To make String declarations easier, there are syntactic shortcuts which can be used:

Figure 13

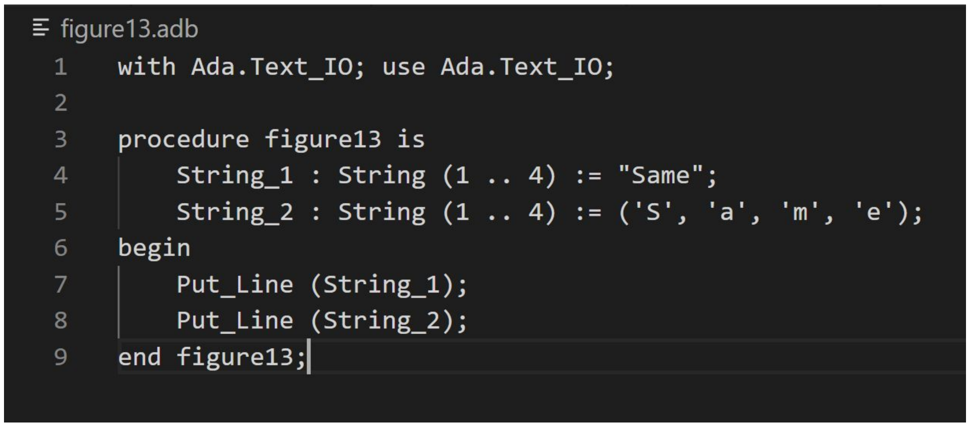
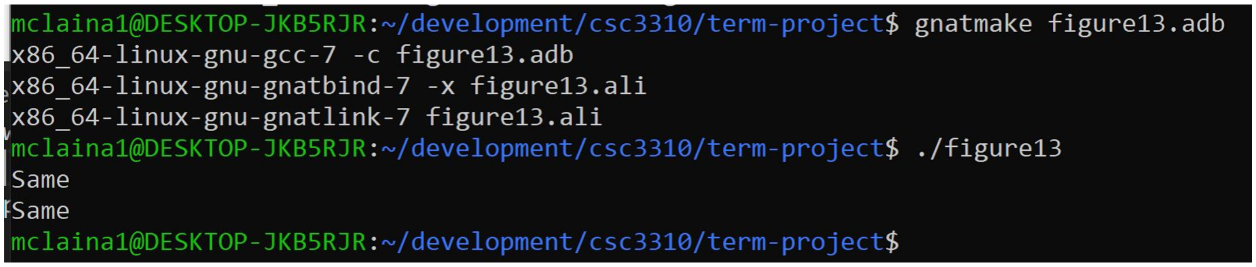


figure13.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure figure13 is
    String_1 : String (1 .. 4) := "Same";
    String_2 : String (1 .. 4) := ('S', 'a', 'm', 'e');
begin
    Put_Line (String_1);
    Put_Line (String_2);
end figure13;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure13.adb
x86_64-linux-gnu-gcc-7 -c figure13.adb
x86_64-linux-gnu-gnatbind-7 -x figure13.ali
x86_64-linux-gnu-gnatlink-7 figure13.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure13
Same
Same
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

As with any array, as long as you specify the initial values of the String the bounds will be automatically assigned. Additionally, strings also have the same attributes as arrays. This frees one from needing to count the number of elements in the String to create a loop or having to change the bounds if they change the initial String value:

Figure 14

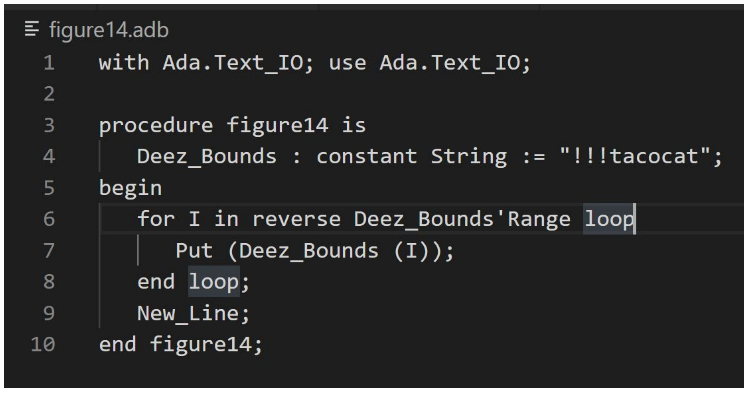
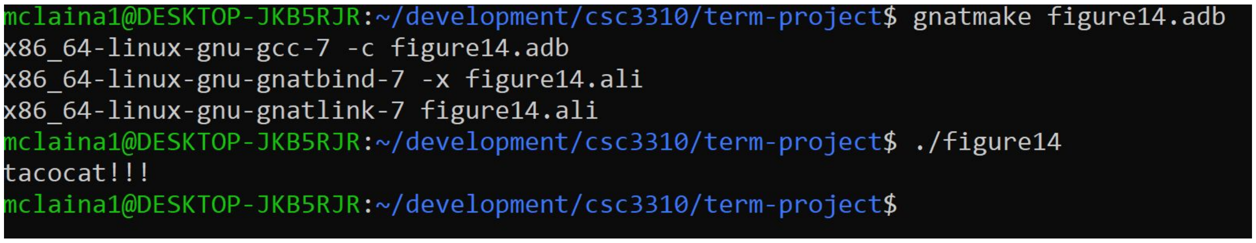


figure14.adb

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure figure14 is
   Deez_Bounds : constant String := "!!!tacocat";
begin
   for I in reverse Deez_Bounds'Range loop
      Put (Deez_Bounds (I));
   end loop;
   New_Line;
end figure14;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure14.adb
x86_64-linux-gnu-gcc-7 -c figure14.adb
x86_64-linux-gnu-gnatbind-7 -x figure14.ali
x86_64-linux-gnu-gnatlink-7 figure14.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure14
tacocat!!!
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

The package `Ada.Strings.Fixed` provides operations on fixed-length character strings. Some available operations are `Count`, `Index`, `Insert`, `Overwrite`, `Delete`, and `Trim`.

`Count` accepts a source string and a pattern string and returns the number of times the pattern string appears in the source string:

Figure 15

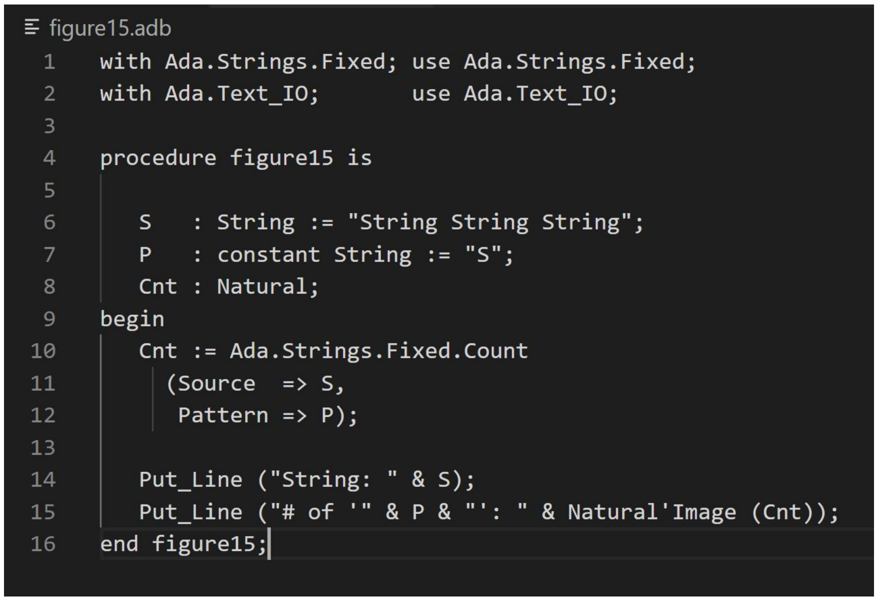
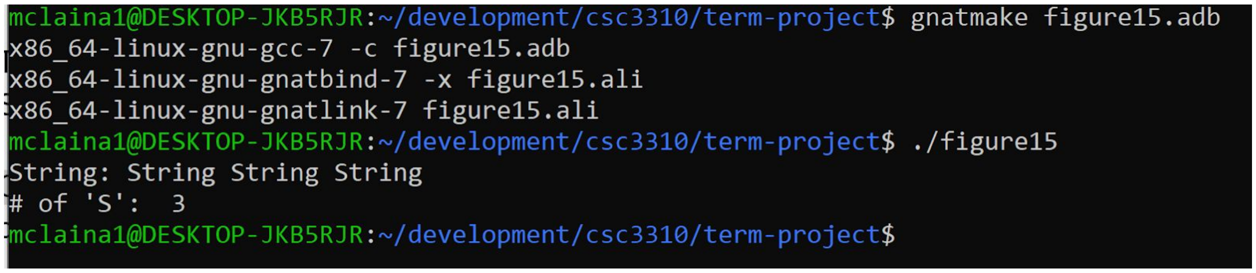


figure15.adb

```
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

procedure figure15 is

   S   : String := "String String String";
   P   : constant String := "S";
   Cnt : Natural;
begin
   Cnt := Ada.Strings.Fixed.Count
     (Source  => S,
      Pattern => P);

   Put_Line ("String: " & S);
   Put_Line ("# of '" & P & "': " & Natural'Image (Cnt));
end figure15;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure15.adb
x86_64-linux-gnu-gcc-7 -c figure15.adb
x86_64-linux-gnu-gnatbind-7 -x figure15.ali
x86_64-linux-gnu-gnatlink-7 figure15.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure15
String: String String String
# of 'S':  3
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

`Index` accepts a source string, a pattern string, and a “from” index and returns the index of the first occurrence of the pattern string in the source string starting at the “from” index.

Figure 16

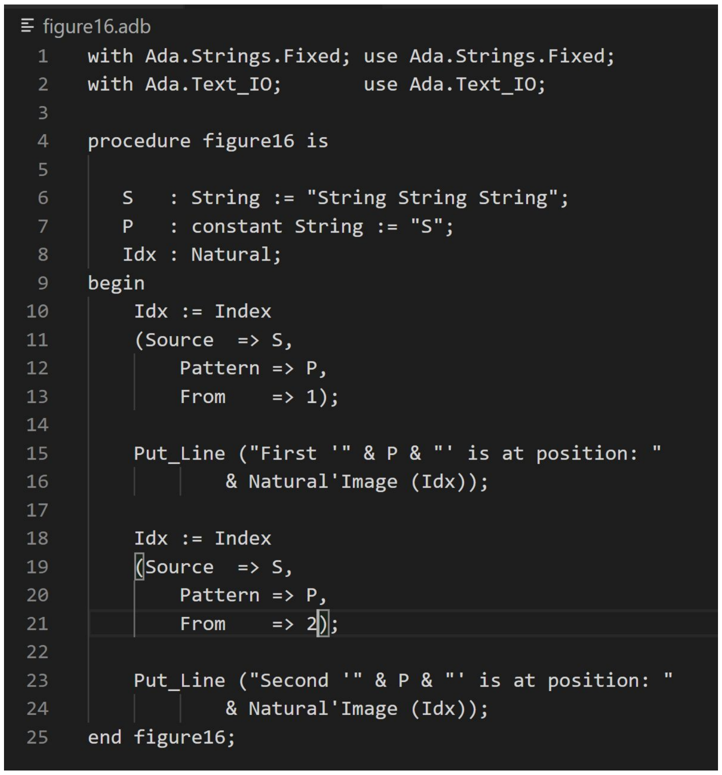
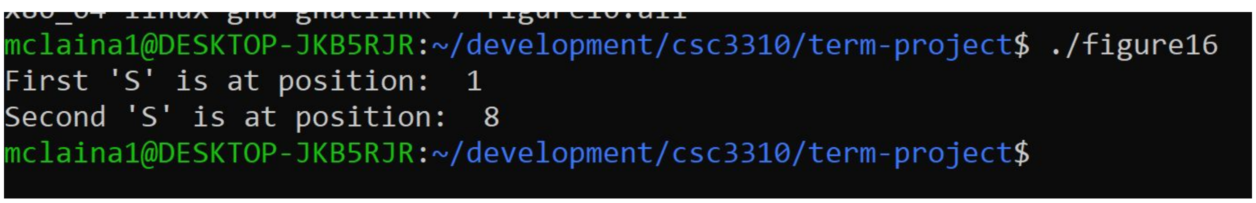


figure16.adb

```
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

procedure figure16 is

   S   : String := "String String String";
   P   : constant String := "S";
   Idx : Natural;
begin
    Idx := Index
    (Source  => S,
        Pattern => P,
        From    => 1);

    Put_Line ("First '" & P & "' is at position: "
            & Natural'Image (Idx));

    Idx := Index
    (Source  => S,
        Pattern => P,
        From    => 2);

    Put_Line ("Second '" & P & "' is at position: "
            & Natural'Image (Idx));
end figure16;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure16
First 'S' is at position:  1
Second 'S' is at position:  8
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

Below is a demonstration of the `Insert`, `Overwrite`, `Trim`, and `Delete` functions from the `Ada.Strings.Fixed` library.

Figure 17

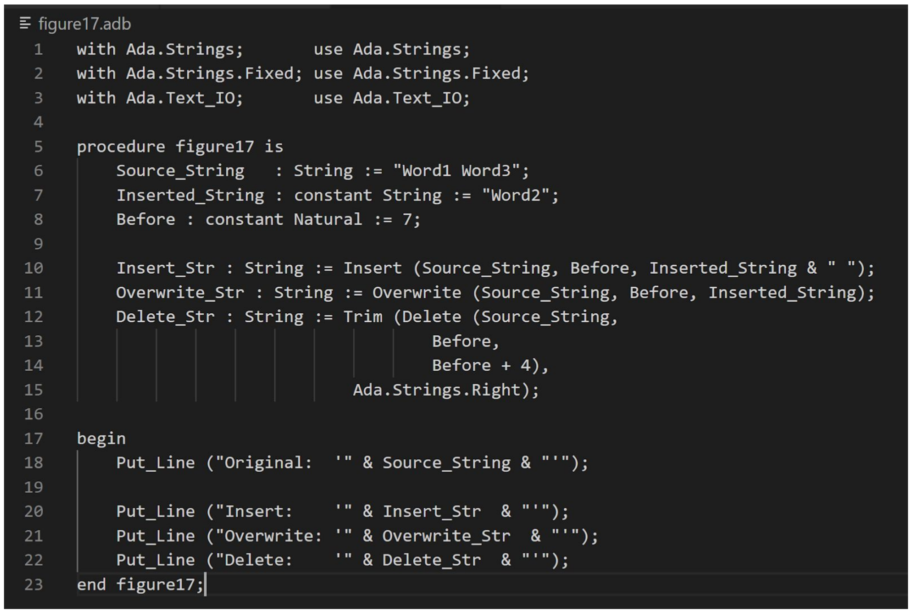
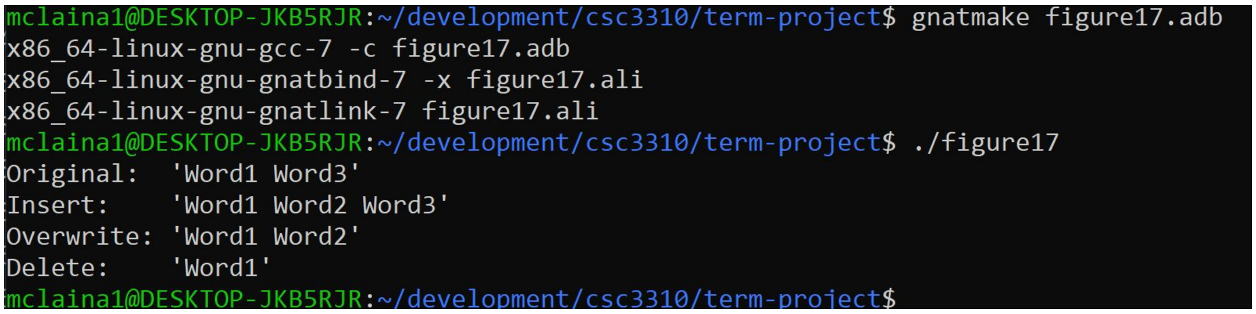


figure17.adb

```ada
with Ada.Strings;       use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

procedure figure17 is
    Source_String   : String := "Word1 Word3";
    Inserted_String : constant String := "Word2";
    Before : constant Natural := 7;

    Insert_Str : String := Insert (Source_String, Before, Inserted_String & " ");
    Overwrite_Str : String := Overwrite (Source_String, Before, Inserted_String);
    Delete_Str : String := Trim (Delete (Source_String,
                                Before,
                                Before + 4),
                        Ada.Strings.Right);

begin
    Put_Line ("Original:  '" & Source_String & "'");

    Put_Line ("Insert:    '" & Insert_Str  & "'");
    Put_Line ("Overwrite: '" & Overwrite_Str  & "'");
    Put_Line ("Delete:    '" & Delete_Str  & "'");
end figure17;
```


```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure17.adb
x86_64-linux-gnu-gcc-7 -c figure17.adb
x86_64-linux-gnu-gnatbind-7 -x figure17.ali
x86_64-linux-gnu-gnatlink-7 figure17.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure17
Original:  'Word1 Word3'
Insert:    'Word1 Word2 Word3'
Overwrite: 'Word1 Word2'
Delete:    'Word1'
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

<u>Strings in Ada's standard library</u>

Ada's String library provides two categories of Strings which address some of the limitations of Ada's fixed-length built-in String type. They are *not* arrays of characters.

Bounded Strings:

Bounded Strings have a maximum length which is fixed at initialization but can hold any string at or below that length during run time. Thus, they are **limited dynamic-length strings** [8]. Below is a demonstration of how to define bounded strings using the `Ada.Strings.Bounded.Generic_Bounded_Length` package. Bounded strings do not have a length attribute, so the length function must be used to return their length.

Figure 18

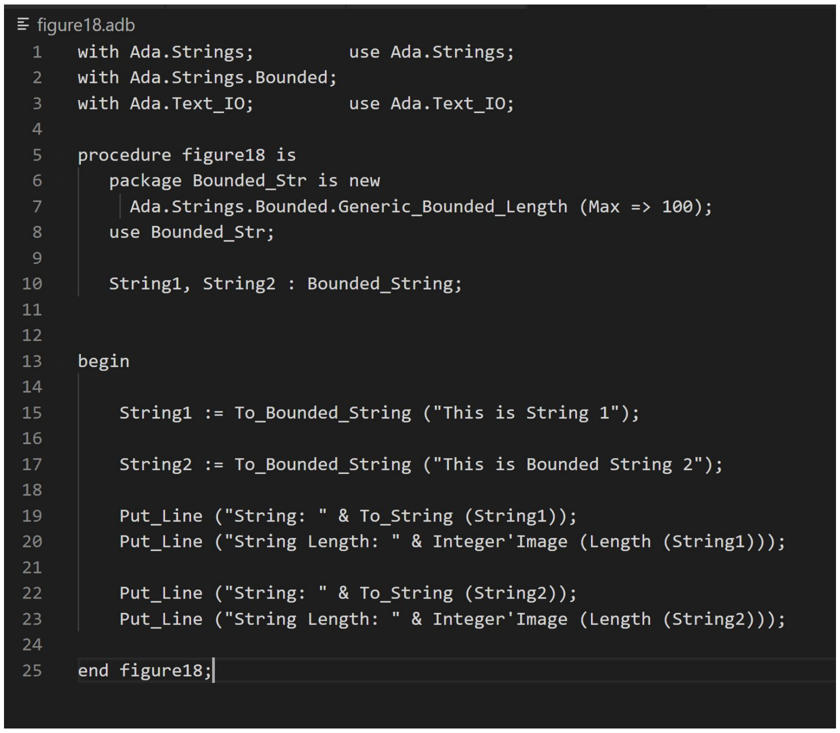
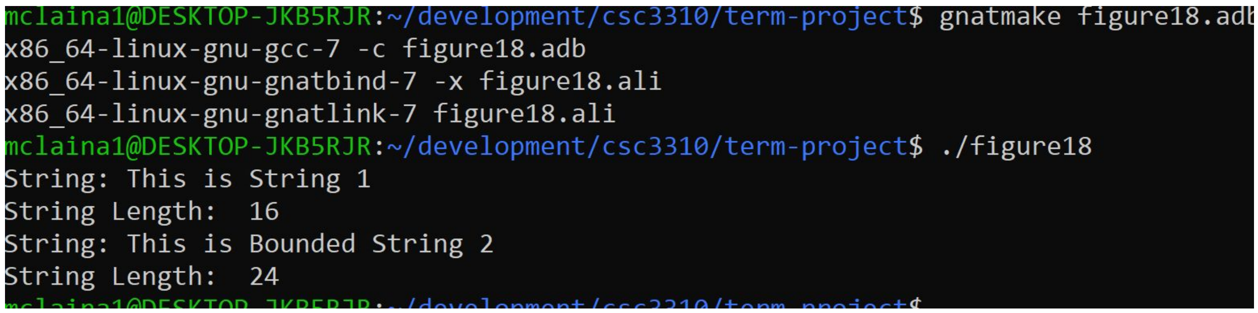


figure18.adb

```
with Ada.Strings;         use Ada.Strings;
with Ada.Strings.Bounded;
with Ada.Text_IO;         use Ada.Text_IO;

procedure figure18 is
   package Bounded_Str is new
     Ada.Strings.Bounded.Generic_Bounded_Length (Max => 100);
   use Bounded_Str;

   String1, String2 : Bounded_String;


begin

    String1 := To_Bounded_String ("This is String 1");

    String2 := To_Bounded_String ("This is Bounded String 2");

    Put_Line ("String: " & To_String (String1));
    Put_Line ("String Length: " & Integer'Image (Length (String1)));

    Put_Line ("String: " & To_String (String2));
    Put_Line ("String Length: " & Integer'Image (Length (String2)));

end figure18;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure18.adb
x86_64-linux-gnu-gcc-7 -c figure18.adb
x86_64-linux-gnu-gnatbind-7 -x figure18.ali
x86_64-linux-gnu-gnatlink-7 figure18.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure18
String: This is String 1
String Length:  16
String: This is Bounded String 2
String Length:  24
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

Unbounded Strings:

Unbounded Strings are dynamically-allocated and can hold a string of any length at run-time. Thus, they are **dynamic-length strings** [8]. Below is an example of how to define unbounded strings using the `Ada.Strings.Unbounded`, note that the length of the unbounded string can grow from its initialized value during run time.

Figure 19

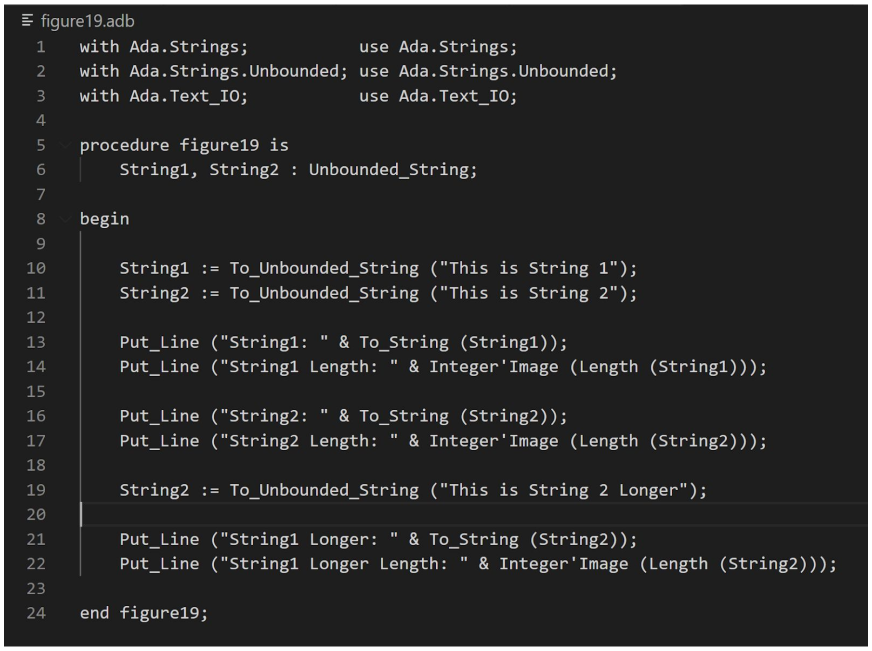
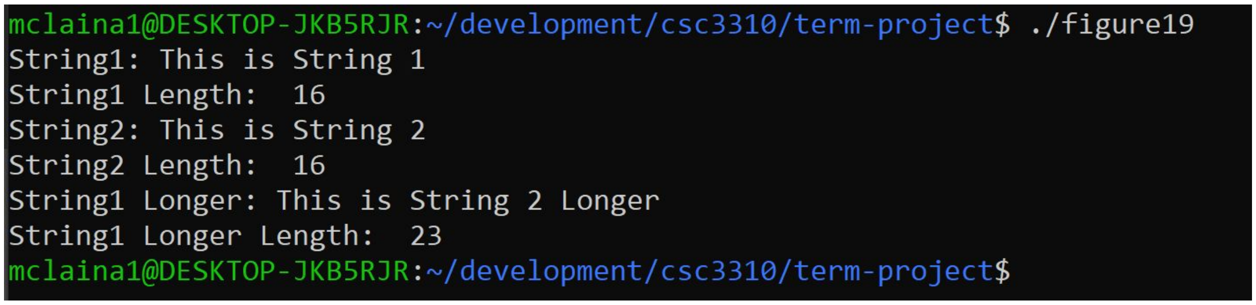


figure19.adb

```ada
with Ada.Strings;           use Ada.Strings;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO;           use Ada.Text_IO;

procedure figure19 is
    String1, String2 : Unbounded_String;

begin

    String1 := To_Unbounded_String ("This is String 1");
    String2 := To_Unbounded_String ("This is String 2");

    Put_Line ("String1: " & To_String (String1));
    Put_Line ("String1 Length: " & Integer'Image (Length (String1)));

    Put_Line ("String2: " & To_String (String2));
    Put_Line ("String2 Length: " & Integer'Image (Length (String2)));

    String2 := To_Unbounded_String ("This is String 2 Longer");

    Put_Line ("String1 Longer: " & To_String (String2));
    Put_Line ("String1 Longer Length: " & Integer'Image (Length (String2)));

end figure19;
```

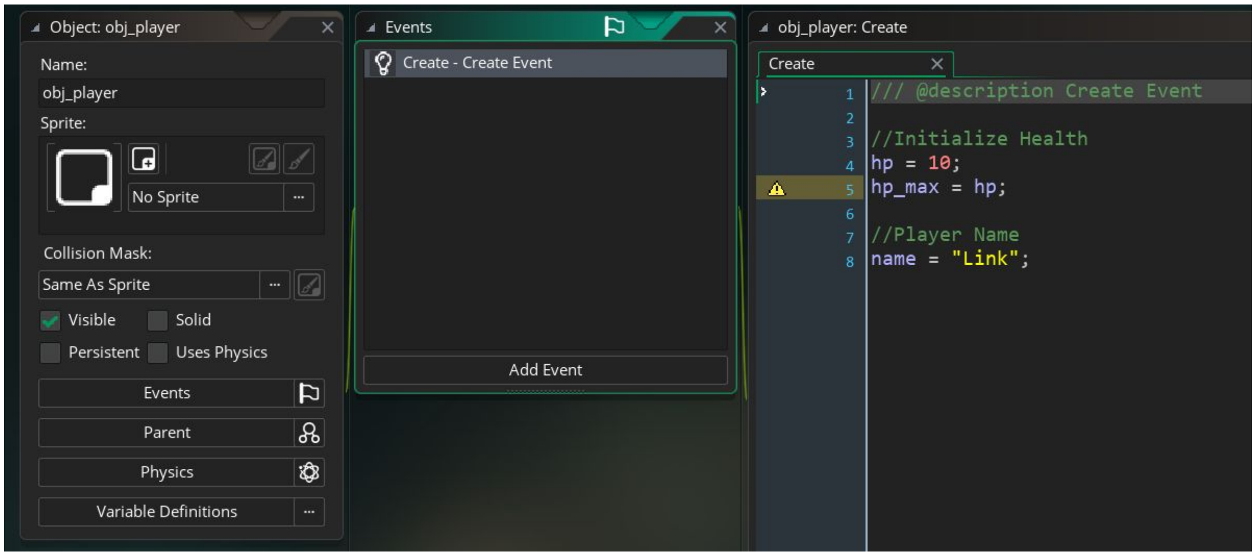


```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure19
String1: This is String 1
String1 Length:  16
String2: This is String 2
String2 Length:  16
String1 Longer: This is String 2 Longer
String1 Longer Length:  23
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

GML

GameMaker Language (GML) is a high-level scripting language included with the GameMaker Studio game engine. It is an imperative, dynamically typed language. It is designed to be easy to use, and the engine allows a new user to get a 2D game running with very few lines of code. This allows for the rapid development of simple and complex video games.

Introduction
GML code is primarily attached to game objects. A GameMaker object contains events, and the code attached to each event will run when the event triggers. The image below shows the object `obj_player` and the code which is in the Create Event, which will run once when the object is instantiated.



Name: obj_player
Sprite: No Sprite
Collision Mask: Same As Sprite
Visible [✓] Solid [ ]
Persistent [ ] Uses Physics [ ]

```
/// @description Create Event

//Initialize Health
hp = 10;
hp_max = hp;

//Player Name
name = "Link";
```

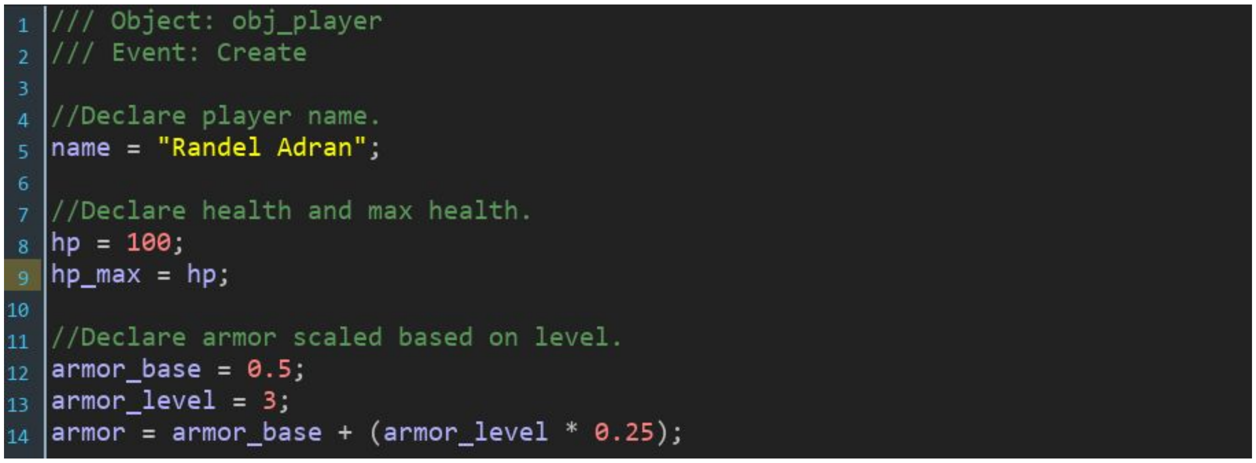
Objects are base templates for instances. Instances are created in a room (the active game environment) according to the properties defined in the object. Each instance operates independently and has its own variables and will run its own code.

The most essential events are the following:

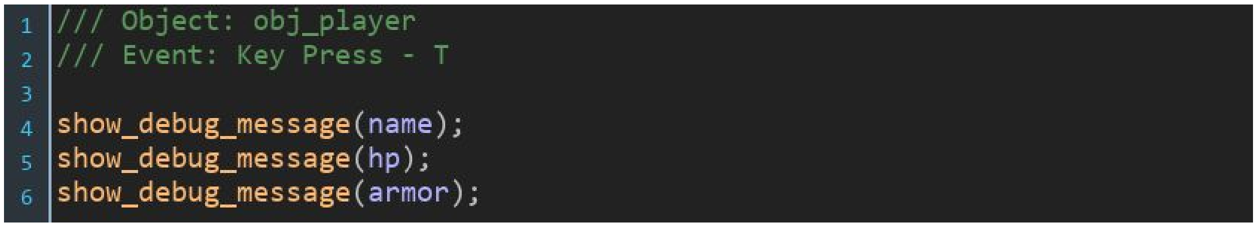
- **Create:** Runs once when an instance is first created.
- **Step:** Runs once every frame that the game is running.
- **Draw:** Performs draw functions that render the instance in the game.

There are many other events, which handle user input, interactions between instances (collisions), miscellaneous purposes. A complete list of events can be seen here: Object Events. The example code here will show a comment to indicate the object and the event which contains the code.

Variable Declaration
GML variables are implicitly declared, and can contain either a number or a string. A new variable is declared by giving it a name and assigning it to a value. Variables are dynamically typed, and will become whatever type they are assigned at runtime.

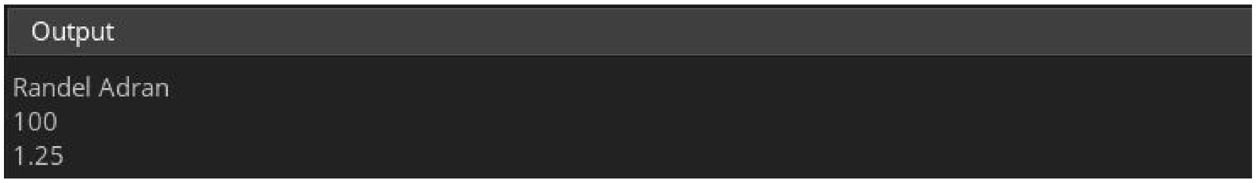


```
/// Object: obj_player
/// Event: Create

//Declare player name.
name = "Randel Adran";

//Declare health and max health.
hp = 100;
hp_max = hp;

//Declare armor scaled based on level.
armor_base = 0.5;
armor_level = 3;
armor = armor_base + (armor_level * 0.25);
```



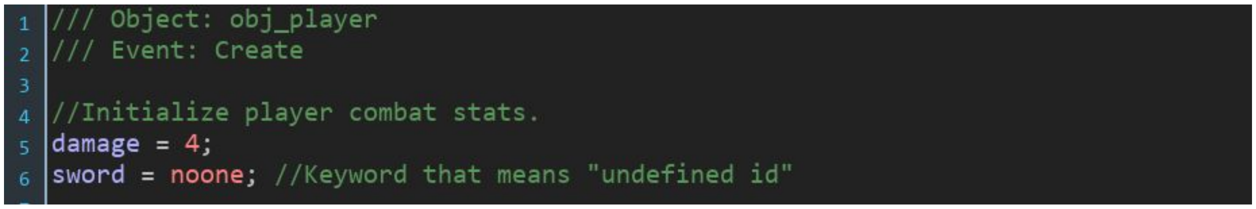
```
/// Object: obj_player
/// Event: Key Press - T

show_debug_message(name);
show_debug_message(hp);
show_debug_message(armor);
```

In this example we declare several variables in the Create event, both numbers and strings, and output their values to the debug console in the Key Press - T event. The output is as follows, and should be pretty unexpected:

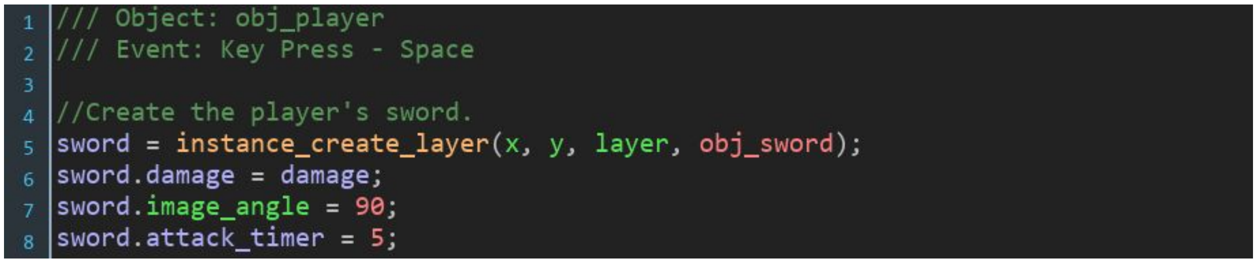


```
Output

Randel Adran
100
1.25
```

Variable Scope
The variables in the first example could be referenced from the Key Press - T event, even though they were created in the Create event. This is because they were declared as instance variables, which can be accessed by any code performed by that instance, or through dot notation: `instanceID.variable`.



```
1  /// Object: obj_player
2  /// Event: Create
3
4  //Initialize player combat stats.
5  damage = 4;
6  sword = noone; //Keyword that means "undefined id"
```



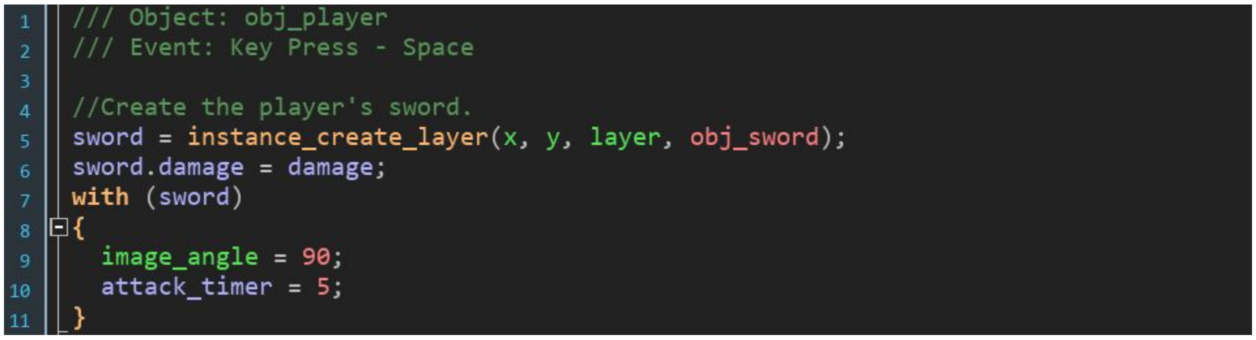
```
1  /// Object: obj_player
2  /// Event: Key Press - Space
3
4  //Create the player's sword.
5  sword = instance_create_layer(x, y, layer, obj_sword);
6  sword.damage = damage;
7  sword.image_angle = 90;
8  sword.attack_timer = 5;
```

This example demonstrates the use of instance variables, and how they can be referenced between two different instances. First, the player initializes its own damage value, and a placeholder variable for the sword. Then, when pressing Space, a new instance of `obj_sword` is created, and the id is stored in the variable `sword`.

Note: The green highlighted variables, `x`, `y`, `layer`, and `image angle` are built-in variables which are declared by default for all newly created instances. They describe properties such as the position of the instance in the room, and required information about how it should be rendered by the game engine.

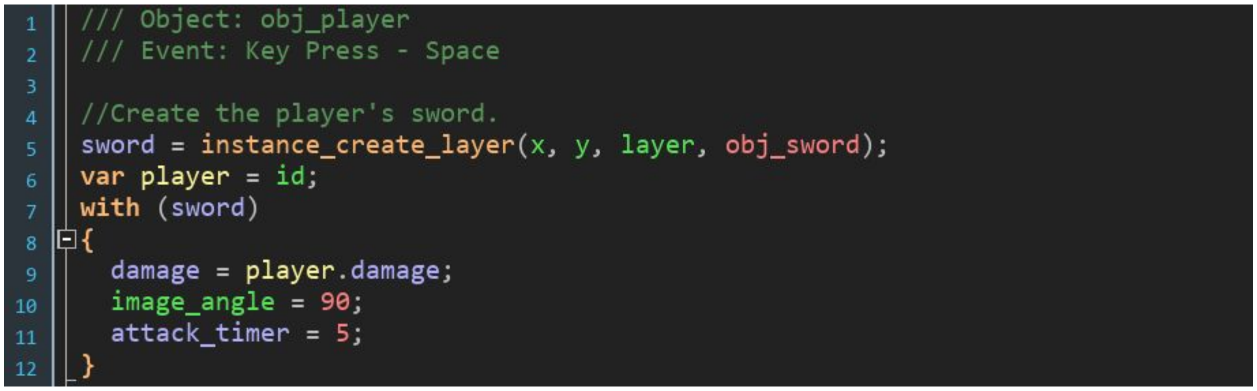
Dot notation is used in lines 6-8 to assign some initial variables for the newly created instance `sword`. Note that `sword.damage` and `damage` have the same name, but they are different variables because they are assigned to different instances.

This code can be modified to use the `with` construct instead. The `with` construct changes the scope of the code to another instance. Additionally, if it is provided an object index instead, it will run that code on every active instance of the object. To reduce the use of dot notation we can write the second part using `with` instead.



```
1   /// Object: obj_player
2   /// Event: Key Press - Space
3
4   //Create the player's sword.
5   sword = instance_create_layer(x, y, layer, obj_sword);
6   sword.damage = damage;
7   with (sword)
8   {
9     image_angle = 90;
10    attack_timer = 5;
11  }
```

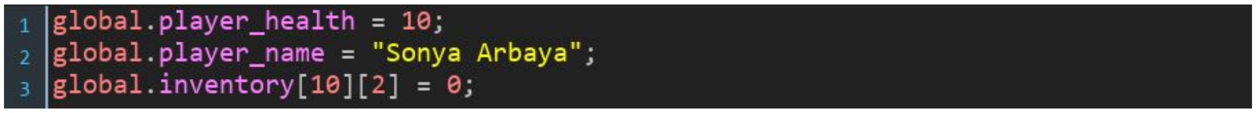
Local variables can be declared as well, using the `var` declaration. Local variables are in scope for the duration of an event or a function, and will be discarded at the end. Additionally, local variables are global within their lifetime, meaning any instance can reference them, even if the scope changes using `with`.



```
/// Object: obj_player
/// Event: Key Press - Space

//Create the player's sword.
sword = instance_create_layer(x, y, layer, obj_sword);
var player = id;
with (sword)
{
  damage = player.damage;
  image_angle = 90;
  attack_timer = 5;
}
```

This third version uses a local variable to store the id of the player instance, and pass it to the sword using `with` to assign the damage variable. This example is unnecessary, but the form of it is useful in more complex situations.

Finally, global variables can be declared with the `global` prefix, and can be referenced by every function and instance in the game. They must be accessed using the `global` prefix as well.



```
global.player_health = 10;
global.player_name = "Sonya Arbaya";
global.inventory[10][2] = 0;
```

Type Checking

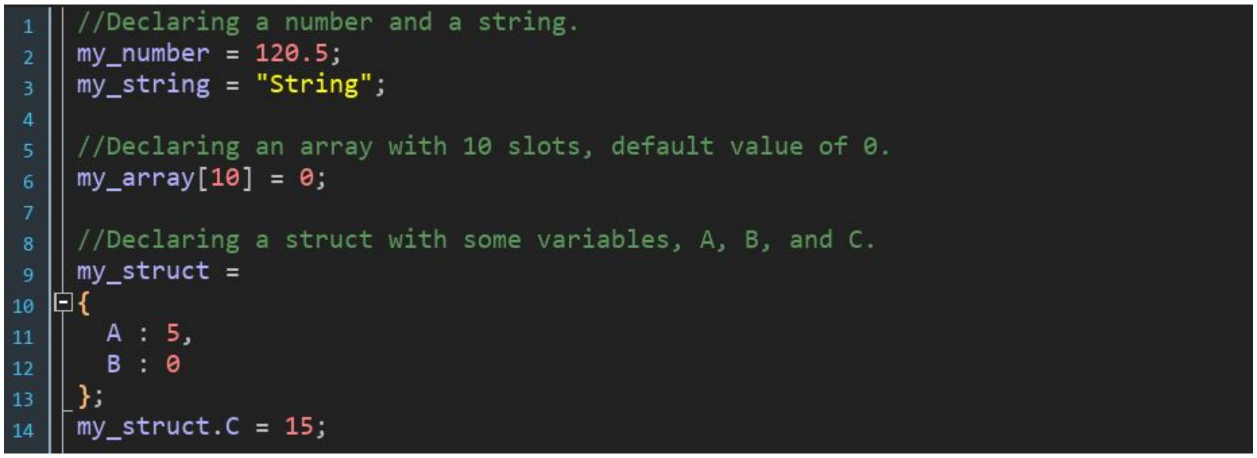
There are four primary data types in GML. They are as follows:

- **Number:** Stores a real number (integer or floating point).
- **String:** Stores a string of characters.
- **Array:** A multidimensional list of values.
- **Struct:** An abstract data type which can contain variables, including methods.

With these few data types, there are a few things to keep in mind.

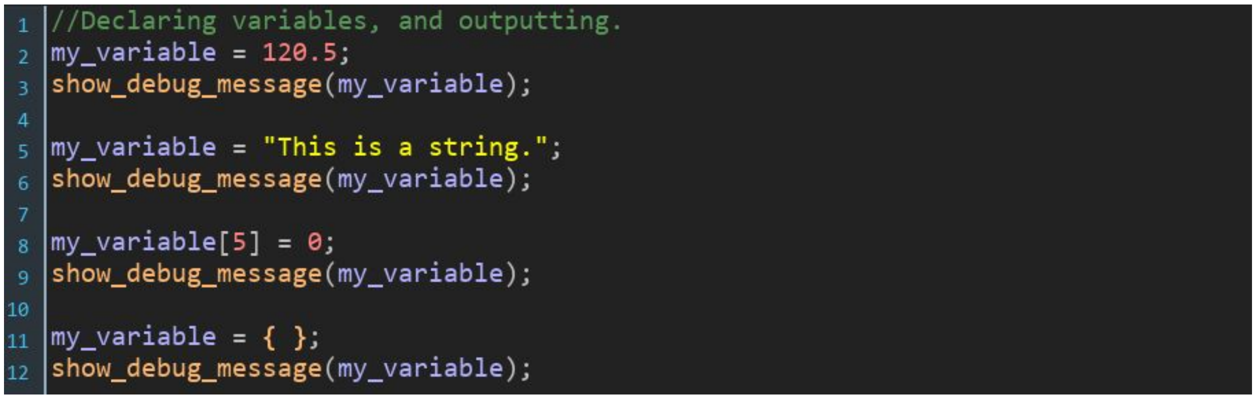
- Boolean values (true and false) are represented using numbers.
- An expression evaluates to “true” if it is greater than 0.5.
- Characters are only represented as a string with a length of one.
- ID values are represented using numbers.

This example shows a valid declaration for each of the data types.

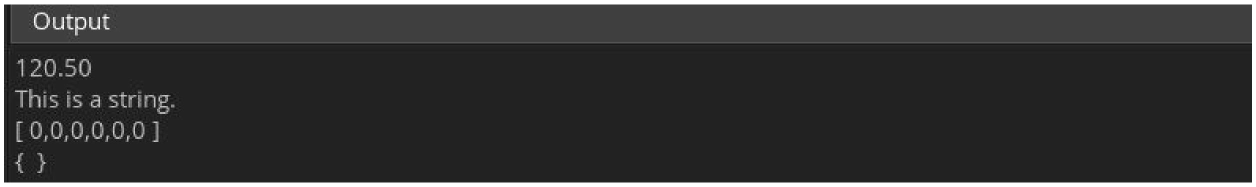


```
//Declaring a number and a string.
my_number = 120.5;
my_string = "String";

//Declaring an array with 10 slots, default value of 0.
my_array[10] = 0;

//Declaring a struct with some variables, A, B, and C.
my_struct =
{
    A : 5,
    B : 0
};
my_struct.C = 15;
```

As GML variables are dynamically typed, GML must also use dynamic type checking. Type-checking is only performed at runtime, the compiler is unable to distinguish between variable types. A variable may be defined as a new type at any time during the program execution.



```
//Declaring variables, and outputting.
my_variable = 120.5;
show_debug_message(my_variable);

my_variable = "This is a string.";
show_debug_message(my_variable);

my_variable[5] = 0;
show_debug_message(my_variable);

my_variable = { };
show_debug_message(my_variable);
```

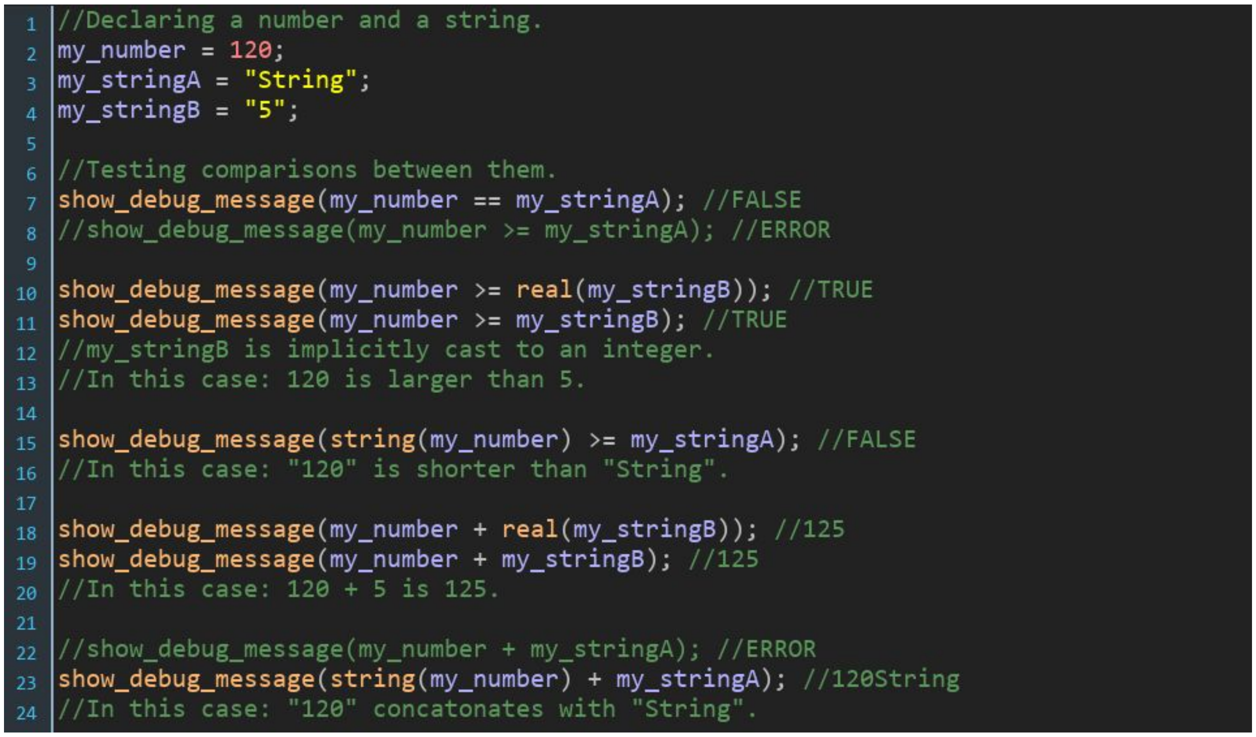


Output

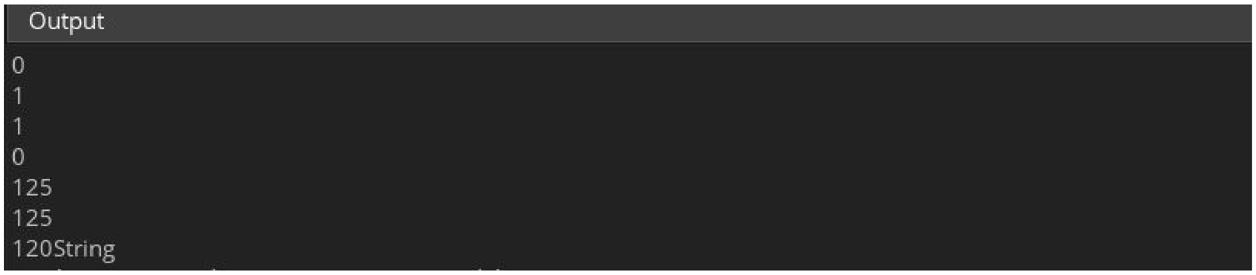
```
120.50
This is a string.
[ 0,0,0,0,0,0 ]
{ }
```

When comparing and operating on variables of different types, GML will implicitly cast values to like types when possible, and throw errors otherwise.

- Equality comparisons between mismatched types evaluate as false.
- Inequality comparisons between mismatched types will throw a type error.
- Operations between mismatched types will throw a type error.
- Variables can cast between number and string using `real` and `string`.
- Strings can be implicitly cast to numbers if they are exclusively numeric.



```
//Declaring a number and a string.
my_number = 120;
my_stringA = "String";
my_stringB = "5";

//Testing comparisons between them.
show_debug_message(my_number == my_stringA); //FALSE
//show_debug_message(my_number >= my_stringA); //ERROR

show_debug_message(my_number >= real(my_stringB)); //TRUE
show_debug_message(my_number >= my_stringB); //TRUE
//my_stringB is implicitly cast to an integer.
//In this case: 120 is larger than 5.

show_debug_message(string(my_number) >= my_stringA); //FALSE
//In this case: "120" is shorter than "String".

show_debug_message(my_number + real(my_stringB)); //125
show_debug_message(my_number + my_stringB); //125
//In this case: 120 + 5 is 125.

//show_debug_message(my_number + my_stringA); //ERROR
show_debug_message(string(my_number) + my_stringA); //120String
//In this case: "120" concatonates with "String".
```

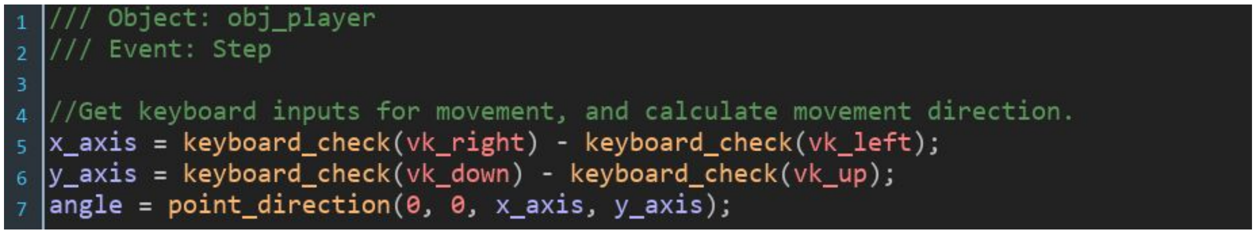


Output

```
0
1
1
0
125
125
120String
```

This example demonstrates a few cases where comparisons between a string and a number variable type may occur. This shows potential type errors that will be caught at runtime, and ways to cast between variables to avoid those errors. Type-checking becomes more complicated with arrays and structs, but the first three general rules above will apply in all scenarios.

Dynamic type checking in GML allows the code to be incredibly flexible. Variables can be easily reused, and many types of data can be stored in the same data structures. (Useful for game save data, or an inventory system). This additionally allows for shortcuts and useful tricks such as the following:



```
/// Object: obj_player
/// Event: Step

//Get keyboard inputs for movement, and calculate movement direction.
x_axis = keyboard_check(vk_right) - keyboard_check(vk_left);
y_axis = keyboard_check(vk_down) - keyboard_check(vk_up);
angle = point_direction(0, 0, x_axis, y_axis);
```

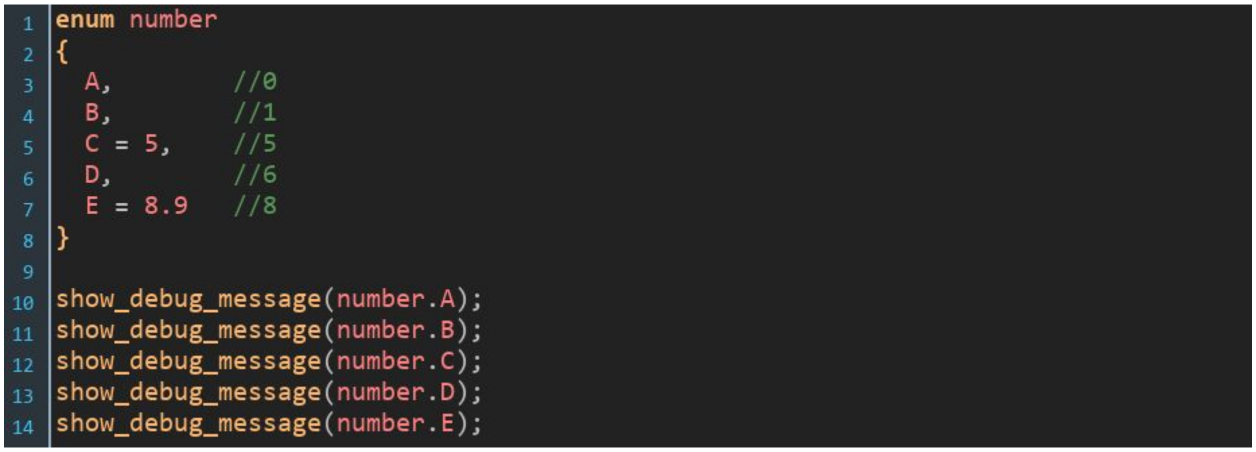
This example adds the expected boolean outputs of the `keyboard_check` function, then calculates a final movement angle. In a strongly typed language, a type cast may be necessary to perform this calculation. Another example in GML would be the ease of creating game save data, where a large variety of potential data would need to be stored in the same data structure.

However, it is not all better, as dynamic type checking has some downsides.

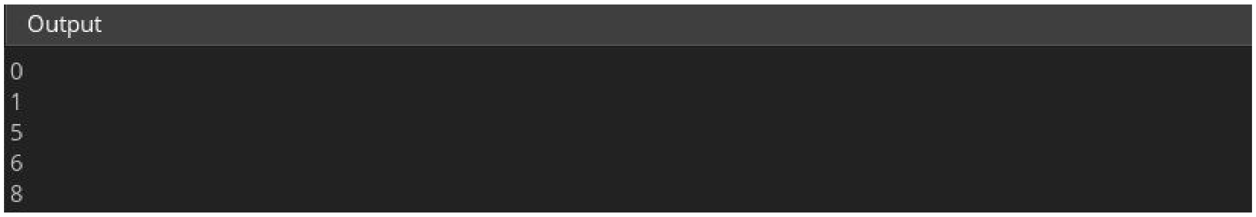
- GML is more prone to type-checking errors, as they cannot be identified at compile time. Errors may occur if a variable is an unexpected type.
- The code is less readable because it is harder to tell what value a particular variable represents at a point in the code.
- Without boolean and integer types, floating point errors or logical errors can occur if a value is out of the bounds of what the programmer expects.

Enumerators

GML enumerators are a collection of named macro constants. An enumerator value must be a numeric integer, other types are not allowed. A floating point number will be rounded down to the nearest integer. Each enumerator term by default will increment from the term before it (starting at zero), but each term can also be defined to any value manually. The syntax to declare an enumerator is as follows:

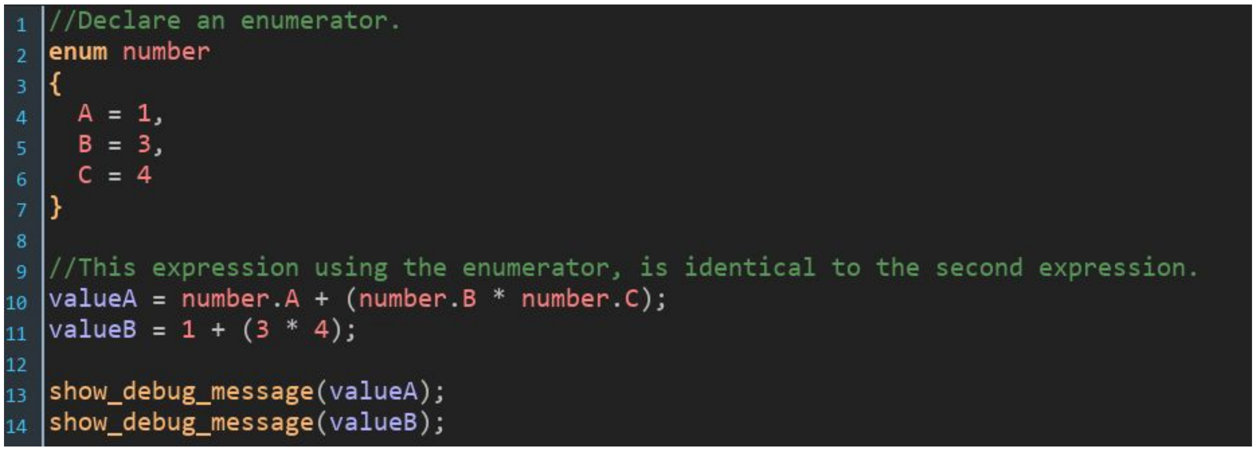


```
enum number
{
  A,        //0
  B,        //1
  C = 5,    //5
  D,        //6
  E = 8.9   //8
}

show_debug_message(number.A);
show_debug_message(number.B);
show_debug_message(number.C);
show_debug_message(number.D);
show_debug_message(number.E);
```



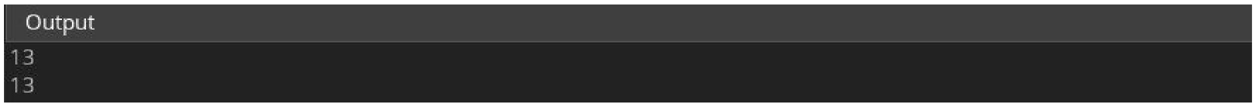
```
Output
0
1
5
6
8
```

GML enumerators are macros. Each term is evaluated to a numerical constant at compile time, then every occurrence of the enumerator is replaced with the value. An enumerator reference is thus equivalent to a numerical literal, just with a name to improve readability. Therefore, these two code snippets are equivalent:



```
//Declare an enumerator.
enum number
{
  A = 1,
  B = 3,
  C = 4
}

//This expression using the enumerator, is identical to the second expression.
valueA = number.A + (number.B * number.C);
valueB = 1 + (3 * 4);

show_debug_message(valueA);
show_debug_message(valueB);
```

Both expressions evaluate to the same value, so the output is identical.



```
13
13
```

Enumerators are extremely useful. Some examples of uses include an inventory system, where an enumerator can be used to represent item IDs. Many systems can be implemented with an enumerator and an associated data structure (look up table) which relates data to each enumerator value. I have used this extensively for inventories, status effects, combat stats, and magic systems.

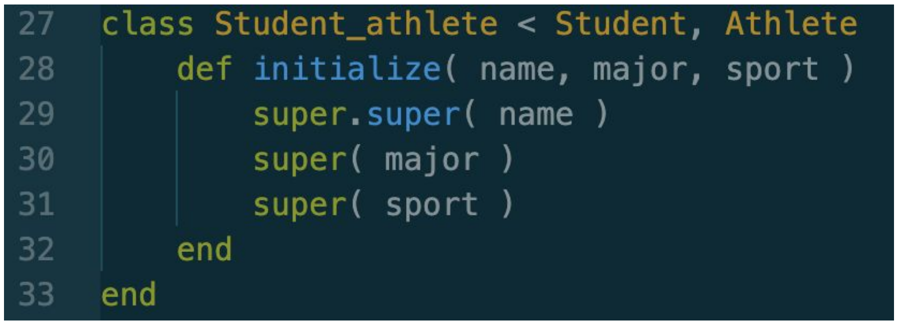
One downside to the GML implementation of enumerators is that they are unable to perform type-checking directly. Every value must be a numeric integer, but there is no way to detect type errors beyond the ordinary dynamic type checking. For example, an error could occur if an inventory system is provided an item ID outside the range of expected IDs. Other languages make enumerators their own type, which is less flexible but also able to define more strict rules for how an enumerator can be used at runtime.

Ruby

Introduction
We used Ruby version 2.6.3p62 for all example code.

Ruby is a high-level, object-oriented scripting language. It was created to be syntactically simple yet powerful, and is a blend of its designer's favorite languages, including Perl, Smalltalk, and Ada. We will discuss class inheritance, including single inheritance and modules, and Ruby's implementation of constant variables.

Class Inheritance
Ruby uses single inheritance, unlike other common object-oriented programming languages like C++ and Python where a class has multiple parent classes. Single inheritance allows classes to inherit from only one parent class. For example, we may want to define a `Student_athlete` class as a child of an existing `Student` class. It may seem most intuitive to make an `Athlete` child classes and then combine them into a new `Student_athlete` class:

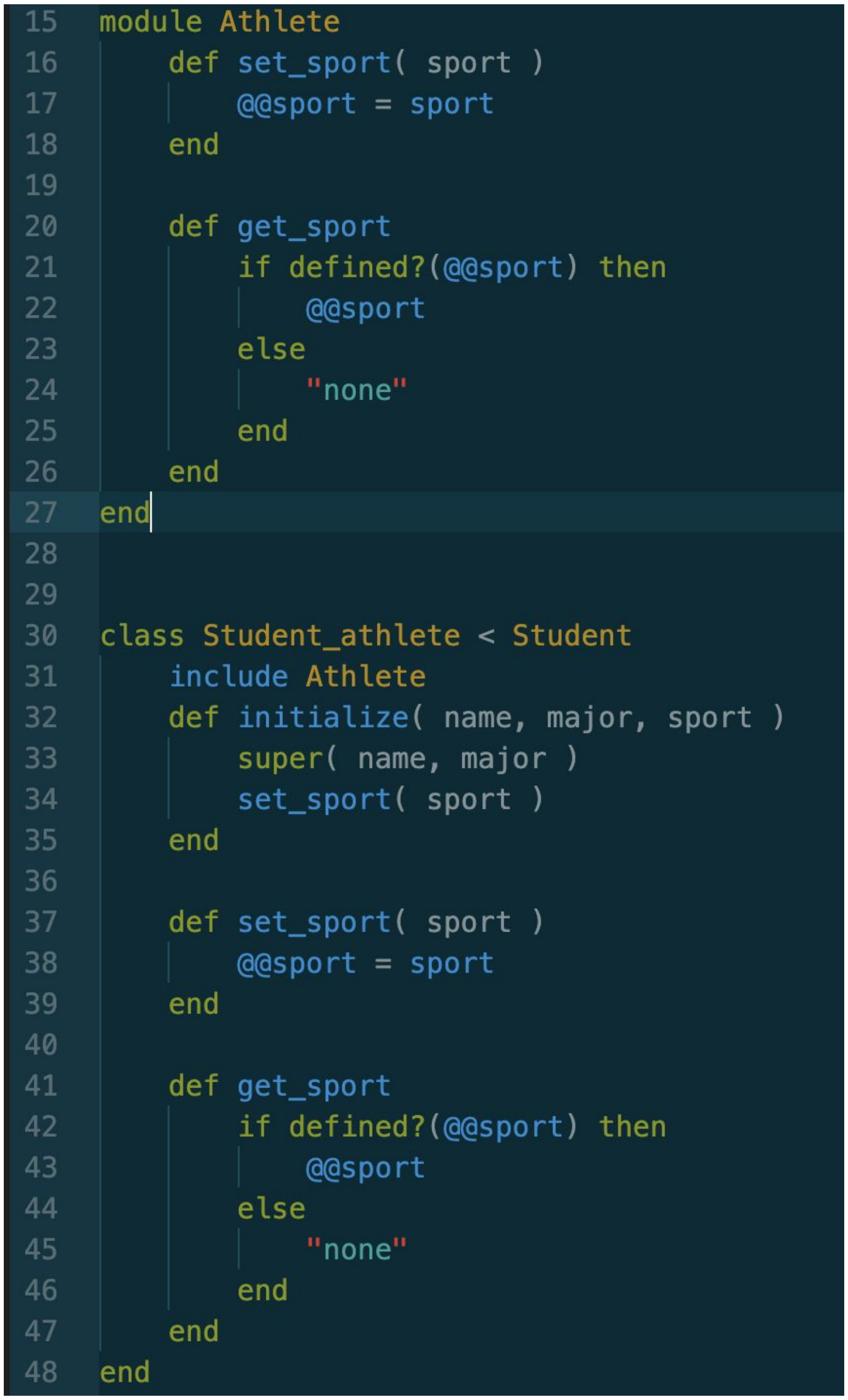


```
class Student_athlete < Student, Athlete
    def initialize( name, major, sport )
        super.super( name )
        super( major )
        super( sport )
    end
end
```

Figure 3.1

The above would be the multiple inheritance version of `Student_athlete`, which Ruby doesn't allow. Instead, we can use modules—collections of methods that can be included in a class. Modules are declared and defined like a class, and when a class includes a module it has access to all the functionality and variables defined in that module.

This is the Ruby module version of creating the `Student_athlete` class:



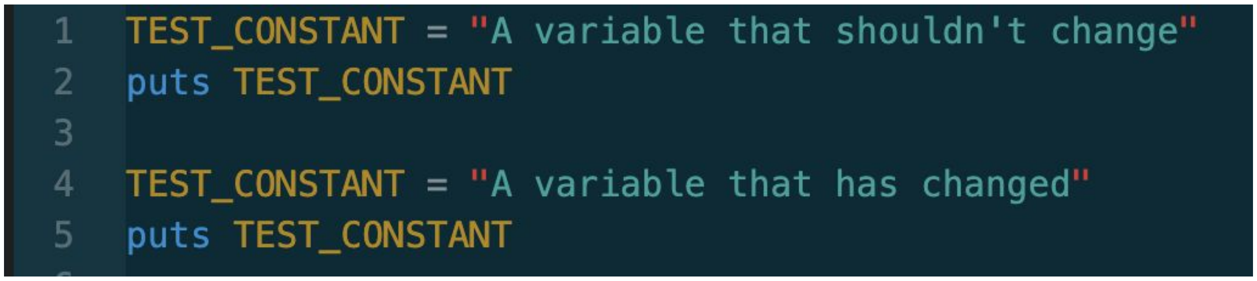
```ruby
module Athlete
    def set_sport( sport )
        @@sport = sport
    end

    def get_sport
        if defined?(@@sport) then
            @@sport
        else
            "none"
        end
    end
end


class Student_athlete < Student
    include Athlete
    def initialize( name, major, sport )
        super( name, major )
        set_sport( sport )
    end

    def set_sport( sport )
        @@sport = sport
    end

    def get_sport
        if defined?(@@sport) then
            @@sport
        else
            "none"
        end
    end
end
```

Figure 3.2

The module acts like an abstract class that can't be used alone but is designed to add functionality to other classes. It provides a namespace for methods and variables to avoid collisions. Any variable conflicts are resolved by hierarchy; the variable closest to the current scope is the one used. Modules can also be more flexible than multiple inheritance.

Constants

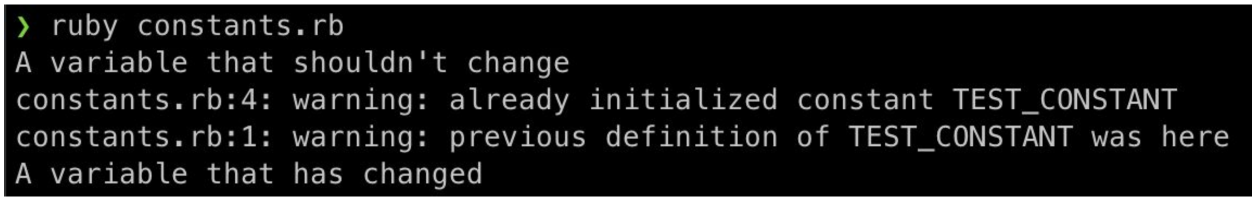
Ruby constants are declared by beginning the variable name with a capital letter. By convention, classes (which are constants) have the first letter capitalized and constant variables are in all caps. In Ruby, constant variables are not enforced at runtime, as shown by:



```ruby
TEST_CONSTANT = "A variable that shouldn't change"
puts TEST_CONSTANT

TEST_CONSTANT = "A variable that has changed"
puts TEST_CONSTANT
```

Figure 3.3

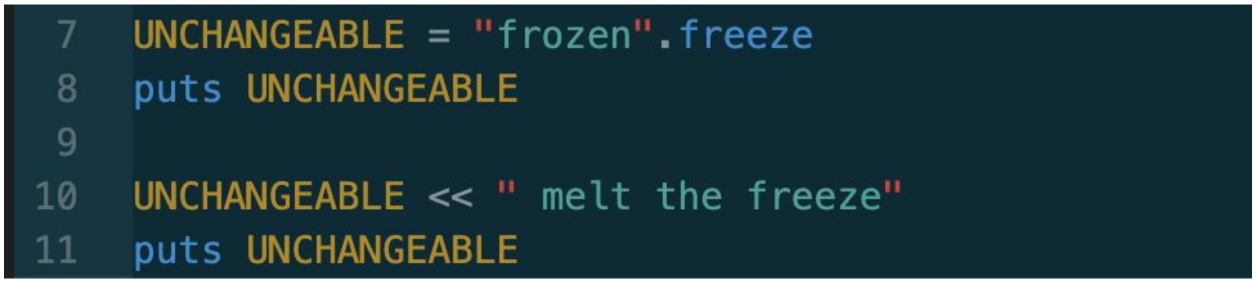
Output:



```
❯ ruby constants.rb
A variable that shouldn't change
constants.rb:4: warning: already initialized constant TEST_CONSTANT
constants.rb:1: warning: previous definition of TEST_CONSTANT was here
A variable that has changed
```

Figure 3.4

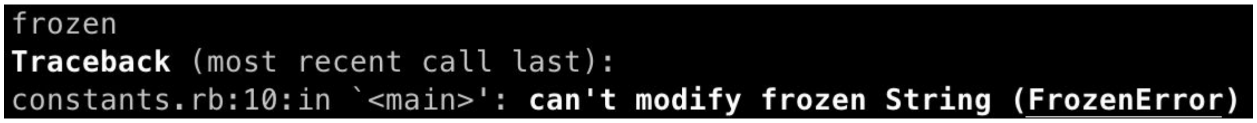
The output shows a warning that we are changing a constant, but does not generate any runtime errors. This is because in Ruby, all variables are pointers to objects rather than containers for an object. We can use immutable objects to keep whatever the variable is pointing at from changing, but still cannot prevent the variable from pointing at something else. The following code will generate a runtime error because we aren't allowed to modify the frozen object:



```ruby
UNCHANGEABLE = "frozen".freeze
puts UNCHANGEABLE

UNCHANGEABLE << " melt the freeze"
puts UNCHANGEABLE
```

Figure 3.5

Output:



```
frozen
Traceback (most recent call last):
constants.rb:10:in `<main>': can't modify frozen String (FrozenError)
```

Figure 3.6

However, if we were to reassign `UNCHANGEABLE` like we reassigned `TEST_CONSTANT` in the previous example, it would generate a similar warning to the first output.

The changeability of "constants" in Ruby is a side effect from how the language implements its object-orientedness, which makes the language less safe but at the same time more consistent.

Conclusion

Our group altogether had a very good experience with this project. We all learned a lot about new languages and different features that have different benefits and drawbacks. With learning these new languages and working on this project, we also learned how to improve our ability to properly follow the syntax of each language. We believe that this attention to detail will allow us to improve our ability to program with languages that we already know. Something that our group struggled with was understanding the purpose of a lot of features in languages. However, we were able to overcome this struggle by comparing and contrasting with languages we already knew. Rather than just trying to pick a favorite, such as C++ or Ruby, we recognized each of the different features in all of the languages, and determined different advantages and disadvantages for each language. From there, we were able to determine different applications and uses for the various languages. While we are still most comfortable and familiar with our existing languages, learning these languages allows us to reflect and understand the capabilities each programming language provides. Below is an evaluation of the 3 programming languages that we studied and a reflection of what we learned.

Ada was the language that our group took the most time learning and understanding. Ada is a very interesting language because of the strong-type aspect and data types (specifically, string and array types). While some of the syntax was difficult to understand at first, there were some aspects of casting and declaring that were relatively similar to Rust. One aspect of Ada that took a while for all of us to comprehend was the strong typing aspect. While a lot of languages we know are generally strongly typed, Ada is definitely “stronger” in terms of its typing. In the first example, where we take two integers and divide them, Ada could not accept the value cast as a Float. We would either need to convert both A and B to a Float, or convert C to an Integer. However, if we ran this code in Java, the answer would result in 0.0. This doesn’t mean that Java is weakly typed, but rather that Ada is very strongly typed, so much so that in order to actually compile our code, we would need an explicit statement to change our data types. Our group was able to identify strengths in this type of strong typed aspect, such as in preventing errors and inconsistencies across operations. However, there are definitely inconveniences with this strong typing language if we need to reference a variable several times in different data forms. We would require an external statement every time we call on a variable that is being assigned to a different data type.

Ada’s array and string types are relatively similar to those of C++. Both Ada and C++ have a runtime error if a loop falls out of range while iterating, and also iterate through the arrays in very similar ways with Ada's built-in functions. One distinct difference with Ada, however, is how array lengths(or char arrays with strings) are statically bound. This means that we can't necessarily depend on arrays if we have to collect a series of values that is greater than the existing array size. We may require a different data type.

While GML seems decently applicable towards games, our group thought that GML had interesting ideas that helped us expand our knowledge of programming languages. When first learning this language as a group, we felt like this language looked very similar to Javascript. We thought that the `keyboard_check` related pretty closely to an "event handler" in that whenever a button, such as the space button was clicked, a certain event or function would occur. What our group struggled the most to learn and comprehend was the `with` construct. We thought it was very interesting to see how this construct changed the scope of variables temporarily. We also thought it was very interesting that global variables had to be declared and referenced explicitly with a `global` statement. We thought this could provide a lot of clarity in the code to show that certain variables could be accessed from different scopes and sections. However, we also thought that there could be some bugs and confusion with the code because normally, global variables are declared outside of a function, but if they are being declared and referenced inside of a function, there could potentially be some scoping issues.

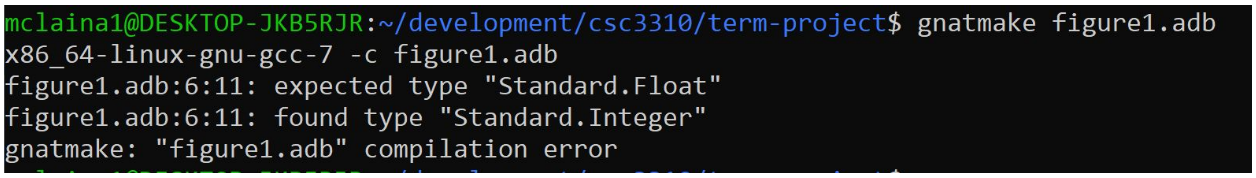
Overall, our group felt like Ruby was the most similar to all of the previous programming languages we knew. The most interesting aspect about Ruby that our group found was the idea of having single inheritance. The majority of our group is most familiar with C++, which actually supports multiple inheritance, which changed our thinking about parent classes and how we would create a class to fit underneath multiple classes if possible. We figured out that instead of creating multiple parent classes whose attributes can be combined, we would create a class with only one parent class, and define unique attributes and variables to the newly created class. We can see that from single inheritance, while more inconvenient than multiple inheritance, single inheritance actually allows us to avoid confusion with scope and changing default values. Other than this difference, we thought that Ruby was a relatively straight-forward language compared to the languages we already know, such as C++ and Java. Our group felt like we were able to understand Ruby quickly and could see how Ruby would be a convenient programming language.

Appendix

Ada

1.1

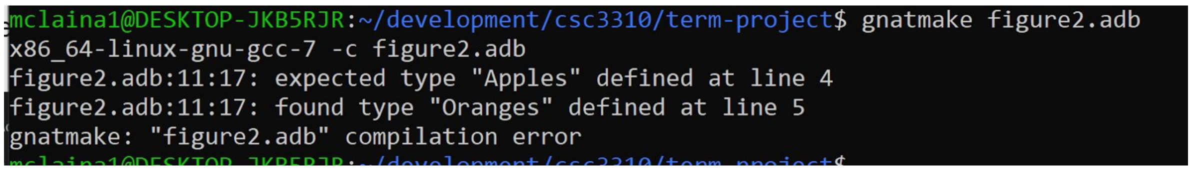
```
procedure figure1 is
   A : Integer := 1;
   B : Integer := 2;
   C : Float;
begin
   C := A / B;
end figure1;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure1.adb
x86_64-linux-gnu-gcc-7 -c figure1.adb
figure1.adb:6:11: expected type "Standard.Float"
figure1.adb:6:11: found type "Standard.Integer"
gnatmake: "figure1.adb" compilation error
```

1.2

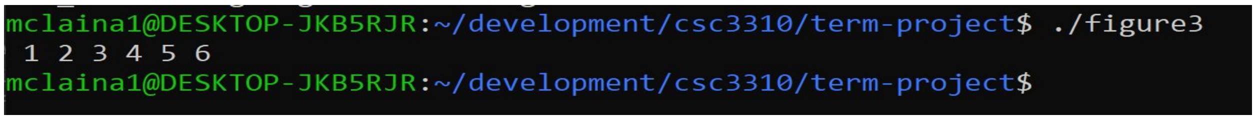
```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure2 is
   type Apples is new Float;
   type Oranges is new Float;

   Var1 : Apples;

   Var2 : constant Oranges := 1000.0;
begin
   Var1 := Var2 * 1000.0;
   Put_Line (Apples'Image (Var1));
end figure2;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure2.adb
x86_64-linux-gnu-gcc-7 -c figure2.adb
figure2.adb:11:17: expected type "Apples" defined at line 4
figure2.adb:11:17: found type "Oranges" defined at line 5
gnatmake: "figure2.adb" compilation error
```

1.3

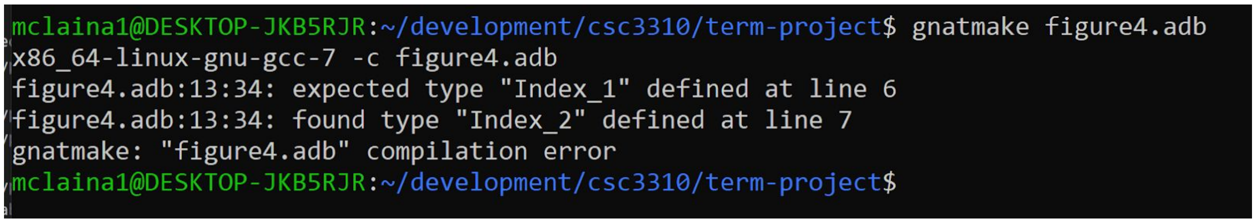
```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure3 is
   type Int_0_100 is range 0 .. 100;
   type Index is range 1 .. 6;

   type Int_0_100_Arr is array (Index) of Int_0_100;

   Arr : Int_0_100_Arr := (1, 2, 3, 4, 5, 6);
begin
   for I in Index loop
      Put (Int_0_100'Image (Arr (I)));
   end loop;
   New_Line;
end figure3;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure3
 1 2 3 4 5 6
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.4

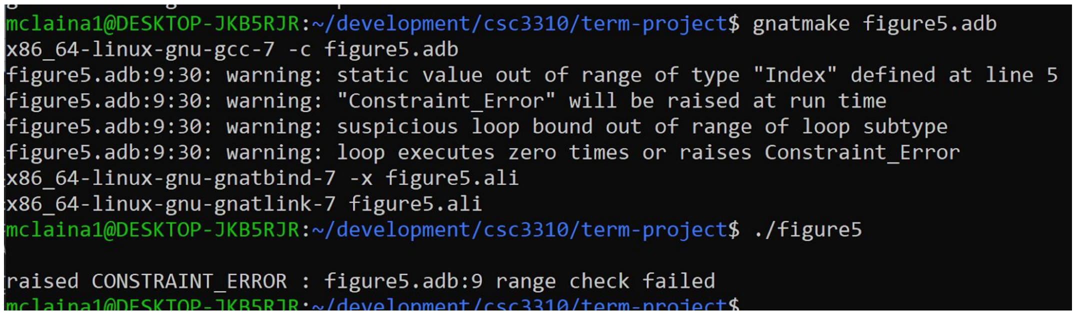
```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure4 is
   type Int_0_100 is range 0 .. 1000;

   type Index_1   is range 1 .. 5;
   type Index_2 is range 1 .. 5;

   type Int_0_100_Arr is array (Index_1) of Int_0_100;
   Arr : Int_0_100_Arr := (10, 11, 12, 13, 14);
begin
   for I in Index_2 loop
      Put (Int_0_100'Image (Arr (I)));
   end loop;
   New_Line;
end figure4;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure4.adb
x86_64-linux-gnu-gcc-7 -c figure4.adb
figure4.adb:13:34: expected type "Index_1" defined at line 6
figure4.adb:13:34: found type "Index_2" defined at line 7
gnatmake: "figure4.adb" compilation error
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.5

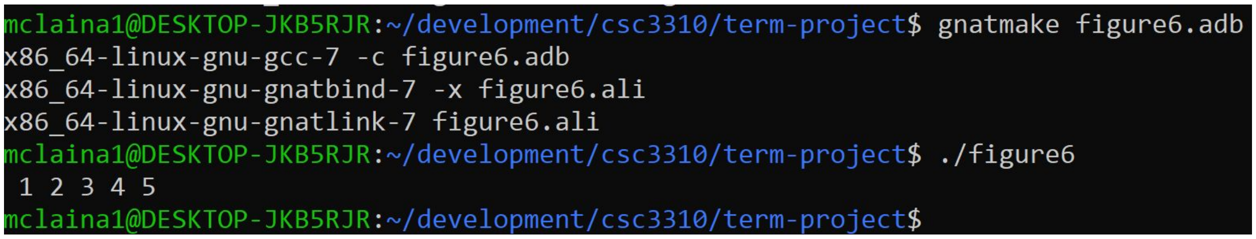
```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure figure5 is
   type Int_0_1000 is range 0 .. 1000;
   type Index is range 1 .. 5;
   type Int_0_1000_Array is array (Index) of Int_0_1000;
   Arr : Int_0_1000_Array := (1, 2, 3, 4, 5);
begin
   for I in Index range 2 .. 6 loop
      Put (Int_0_1000'Image (Arr (I)));
   end loop;
   New_Line;
end figure5;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure5.adb
x86_64-linux-gnu-gcc-7 -c figure5.adb
figure5.adb:9:30: warning: static value out of range of type "Index" defined at line 5
figure5.adb:9:30: warning: "Constraint_Error" will be raised at run time
figure5.adb:9:30: warning: suspicious loop bound out of range of loop subtype
figure5.adb:9:30: warning: loop executes zero times or raises Constraint_Error
x86_64-linux-gnu-gnatbind-7 -x figure5.ali
x86_64-linux-gnu-gnatlink-7 figure5.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure5

raised CONSTRAINT_ERROR : figure5.adb:9 range check failed
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.6

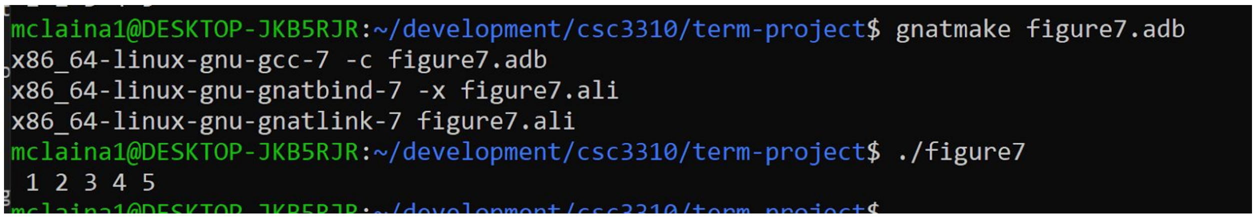
```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure figure6 is
   type Int_0_1000 is range 0 .. 1000;
   type Int_0_1000_Array is array (1 .. 5) of Int_0_1000;

   Arr : Int_0_1000_Array := (1, 2, 3, 4, 5);
begin
   for I in 1 .. 5 loop
      Put (Int_0_1000'Image (Arr (I)));
   end loop;
   New_Line;
end figure6;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure6.adb
x86_64-linux-gnu-gcc-7 -c figure6.adb
x86_64-linux-gnu-gnatbind-7 -x figure6.ali
x86_64-linux-gnu-gnatlink-7 figure6.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure6
 1 2 3 4 5
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.7

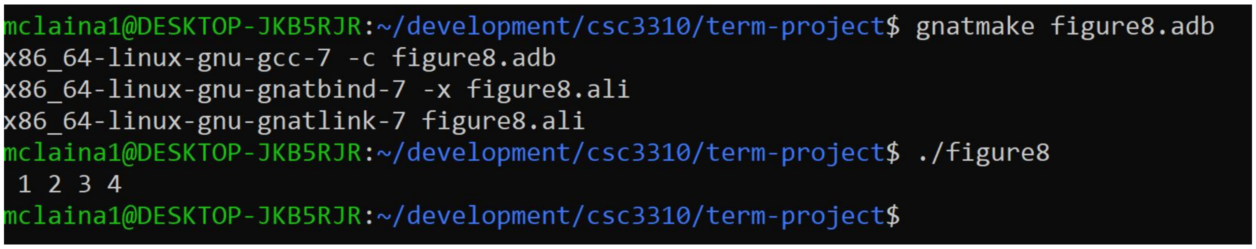
```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure7 is
   type Int_0_1000 is range 0 .. 1000;
   type Int_0_1000_Array is array (1 .. 5) of Int_0_1000;
   Arr : Int_0_1000_Array := (1, 2, 3, 4, 5);
begin
   for I in Arr'Range loop
      Put (Int_0_1000'Image (Arr (I)));
   end loop;
   New_Line;
end figure7;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure7.adb
x86_64-linux-gnu-gcc-7 -c figure7.adb
x86_64-linux-gnu-gnatbind-7 -x figure7.ali
x86_64-linux-gnu-gnatlink-7 figure7.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure7
 1 2 3 4 5
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.8

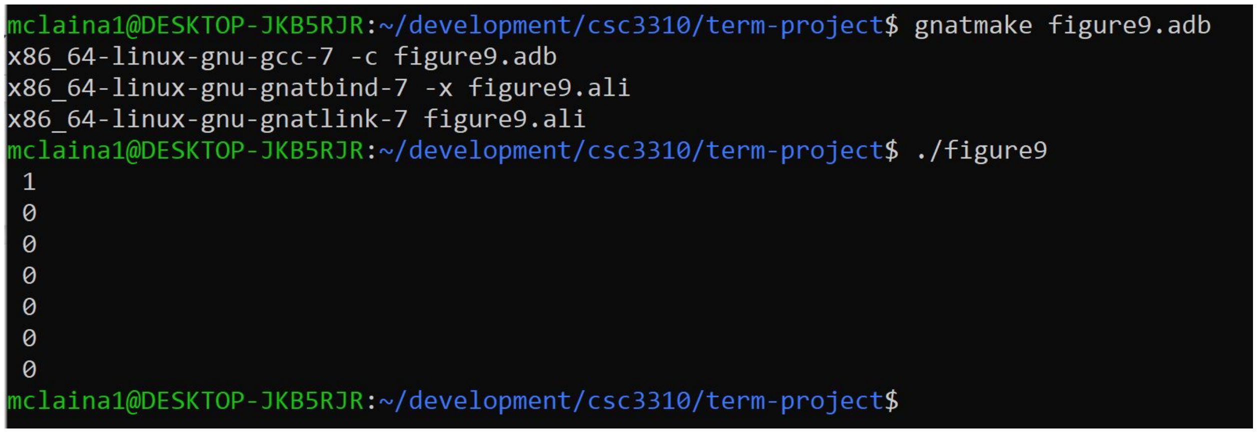
```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure8 is
   type Int_0_1000 is range 0 .. 1000;
   type Int_0_1000_Array is array (1 .. 5) of Int_0_1000;
   Arr : Int_0_1000_Array := (1, 2, 3, 4, 5);
begin
   for I in Arr'First .. Arr'Last - 1 loop
      Put (Int_0_1000'Image (Arr (I)));
   end loop;
   New_Line;
end figure8;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure8.adb
x86_64-linux-gnu-gcc-7 -c figure8.adb
x86_64-linux-gnu-gnatbind-7 -x figure8.ali
x86_64-linux-gnu-gnatlink-7 figure8.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure8
 1 2 3 4
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.9

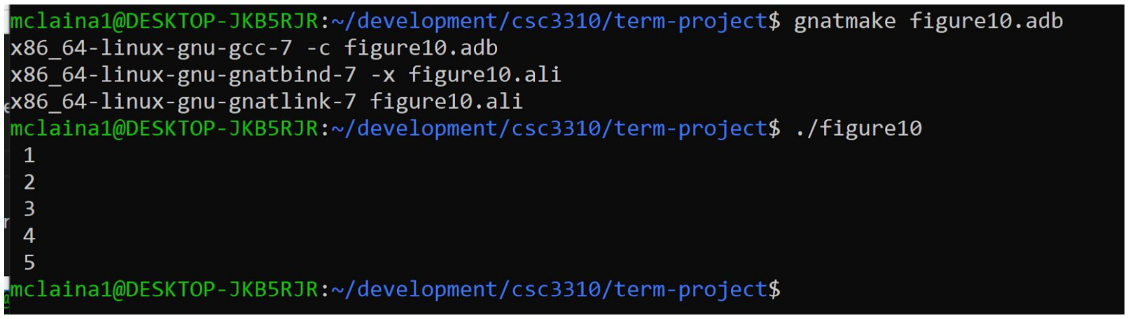
```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure9 is
   type Colors is (Red, Blue, Yellow,
                 Green, Orange, Black, White);

   type Color_Array_Type is array (Colors range <>) of Integer;

   Color_Array : constant Color_Array_Type (Red .. White) :=
      (Red => 1, others => 0);

begin
   for I in Color_Array'Range loop
      Put_Line (Integer'Image (Color_Array (I)));
   end loop;
end figure9;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure9.adb
x86_64-linux-gnu-gcc-7 -c figure9.adb
x86_64-linux-gnu-gnatbind-7 -x figure9.ali
x86_64-linux-gnu-gnatlink-7 figure9.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure9
 1
 0
 0
 0
 0
 0
 0
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.10

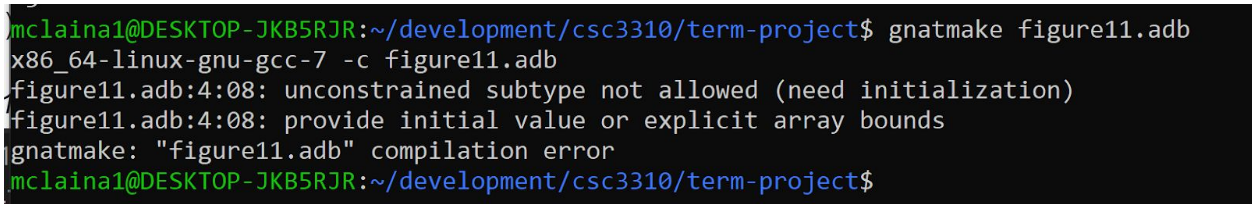
```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure10 is
   type Int_Array is array (Integer range <>) of Integer;

   Int_Array_1 : constant Int_Array := (1, 2, 3, 4, 5);

begin
    for I in Int_Array_1'Range loop
        Put_Line (Integer'Image (Int_Array_1 (I)));
    end loop;
end figure10;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure10.adb
x86_64-linux-gnu-gcc-7 -c figure10.adb
x86_64-linux-gnu-gnatbind-7 -x figure10.ali
x86_64-linux-gnu-gnatlink-7 figure10.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure10
 1
 2
 3
 4
 5
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.11

```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure11 is
   A : String;
begin
   A := "My String";
end figure11;
```

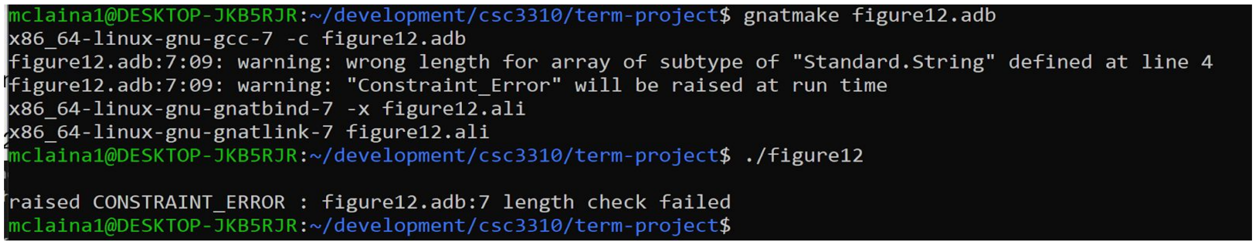


```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure11.adb
x86_64-linux-gnu-gcc-7 -c figure11.adb
figure11.adb:4:08: unconstrained subtype not allowed (need initialization)
figure11.adb:4:08: provide initial value or explicit array bounds
gnatmake: "figure11.adb" compilation error
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.12

```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure12 is
```

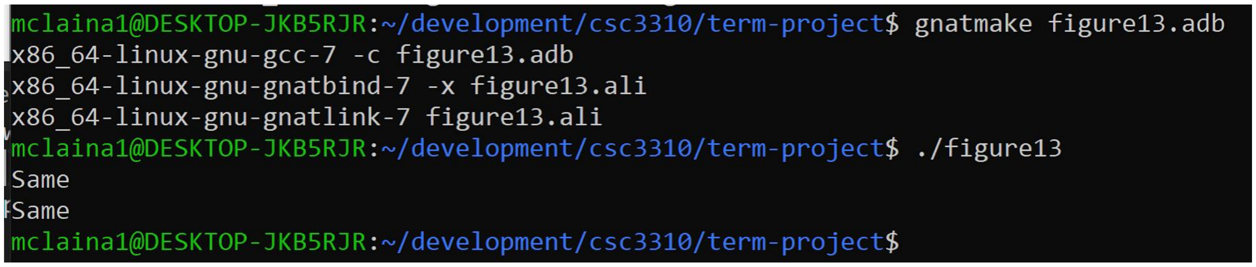
```
   A : String := "Small";
begin
   A := "Good!";
   A := "Too Big!";
end figure12;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure12.adb
x86_64-linux-gnu-gcc-7 -c figure12.adb
figure12.adb:7:09: warning: wrong length for array of subtype of "Standard.String" defined at line 4
figure12.adb:7:09: warning: "Constraint_Error" will be raised at run time
x86_64-linux-gnu-gnatbind-7 -x figure12.ali
x86_64-linux-gnu-gnatlink-7 figure12.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure12

raised CONSTRAINT_ERROR : figure12.adb:7 length check failed
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.13

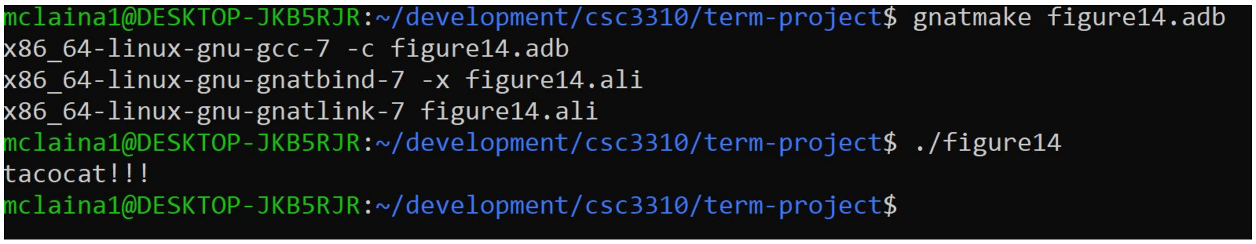
```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure13 is
    String_1 : String (1 .. 4) := "Same";
    String_2 : String (1 .. 4) := ('S', 'a', 'm', 'e');
begin
    Put_Line (String_1);
    Put_Line (String_2);
end figure13;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure13.adb
x86_64-linux-gnu-gcc-7 -c figure13.adb
x86_64-linux-gnu-gnatbind-7 -x figure13.ali
x86_64-linux-gnu-gnatlink-7 figure13.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure13
Same
Same
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.14

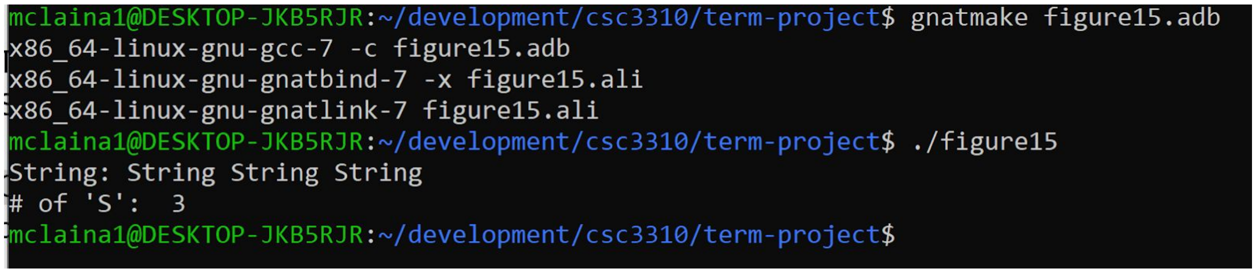
```
with Ada.Text_IO; use Ada.Text_IO;

procedure figure14 is
   Deez_Bounds : constant String := "!!!tacocat";
begin
   for I in reverse Deez_Bounds'Range loop
       Put (Deez_Bounds (I));
   end loop;
   New_Line;
end figure14;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure14.adb
x86_64-linux-gnu-gcc-7 -c figure14.adb
x86_64-linux-gnu-gnatbind-7 -x figure14.ali
x86_64-linux-gnu-gnatlink-7 figure14.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure14
tacocat!!!
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.15

```
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

procedure figure15 is

   S   : String := "String String String";
   P   : constant String := "S";
   Cnt : Natural;
begin
   Cnt := Ada.Strings.Fixed.Count
     (Source  => S,
      Pattern => P);

   Put_Line ("String: " & S);
   Put_Line ("# of '" & P & "': " & Natural'Image (Cnt));
end figure15;
```

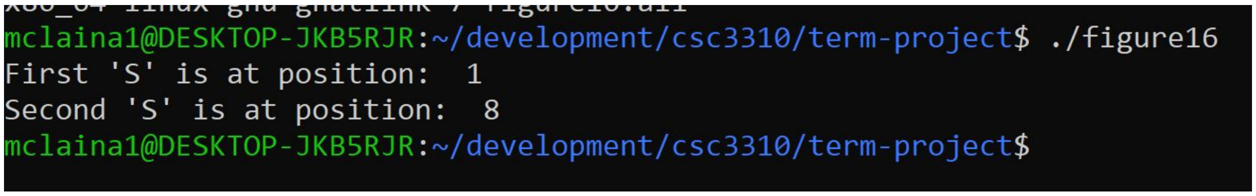


```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure15.adb
x86_64-linux-gnu-gcc-7 -c figure15.adb
x86_64-linux-gnu-gnatbind-7 -x figure15.ali
x86_64-linux-gnu-gnatlink-7 figure15.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure15
String: String String String
# of 'S':  3
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.16

```
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

procedure figure16 is
```

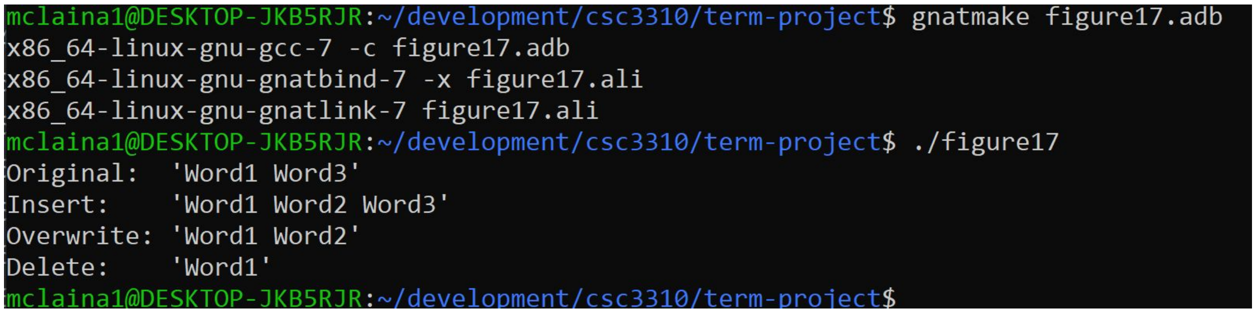
```
   S   : String := "String String String";
   P   : constant String := "S";
   Idx : Natural;
begin
    Idx := Index
    (Source  => S,
        Pattern => P,
        From    => 1);

    Put_Line ("First '" & P & "' is at position: "
            & Natural'Image (Idx));

    Idx := Index
   (Source  => S,
        Pattern => P,
        From    => 2);

    Put_Line ("Second '" & P & "' is at position: "
            & Natural'Image (Idx));
end figure16;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure16
First 'S' is at position:  1
Second 'S' is at position:  8
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.17

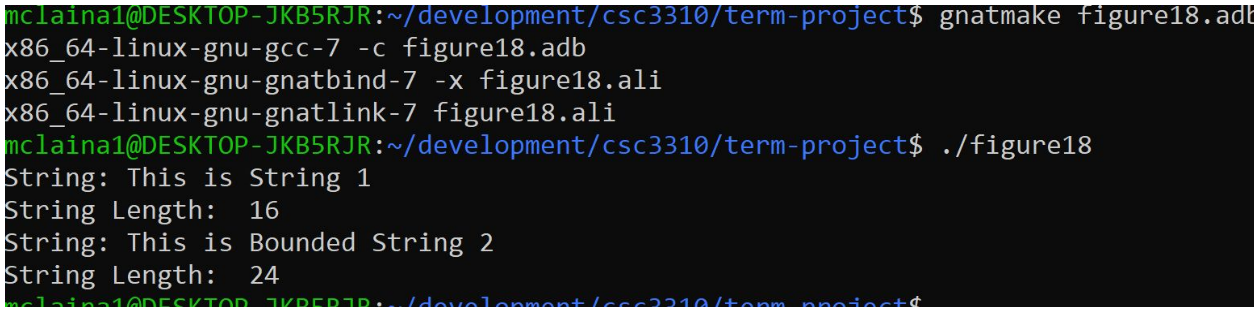
```
with Ada.Strings;       use Ada.Strings;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Text_IO;       use Ada.Text_IO;

procedure figure17 is
    Source_String   : String := "Word1 Word3";
    Inserted_String : constant String := "Word2";
    Before : constant Natural := 7;

    Insert_Str : String := Insert (Source_String, Before,
Inserted_String & " ");
    Overwrite_Str : String := Overwrite (Source_String, Before,
Inserted_String);
    Delete_Str : String := Trim (Delete (Source_String,
                                    Before,
                                    Before + 4),
                           Ada.Strings.Right);

begin
    Put_Line ("Original:  '" & Source_String & "'");

    Put_Line ("Insert:    '" & Insert_Str  & "'");
    Put_Line ("Overwrite: '" & Overwrite_Str  & "'");
    Put_Line ("Delete:    '" & Delete_Str  & "'");
end figure17;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure17.adb
x86_64-linux-gnu-gcc-7 -c figure17.adb
x86_64-linux-gnu-gnatbind-7 -x figure17.ali
x86_64-linux-gnu-gnatlink-7 figure17.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure17
Original:  'Word1 Word3'
Insert:    'Word1 Word2 Word3'
Overwrite: 'Word1 Word2'
Delete:    'Word1'
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

1.18

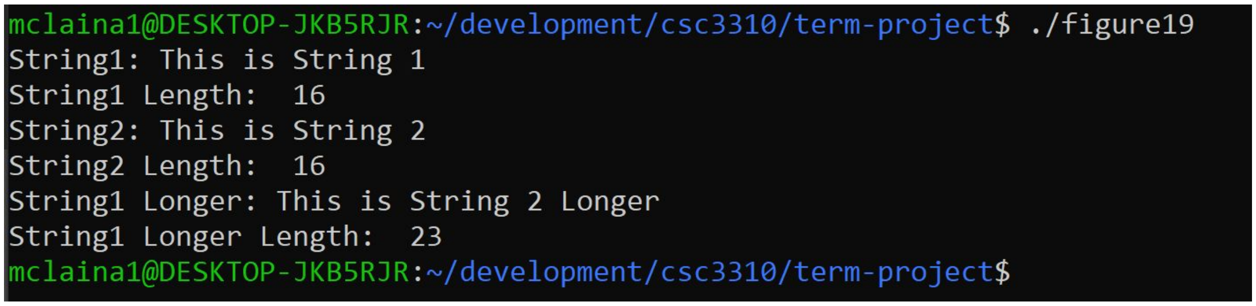
```
with Ada.Strings;          use Ada.Strings;
with Ada.Strings.Bounded;
with Ada.Text_IO;          use Ada.Text_IO;

procedure figure18 is
   package Bounded_Str is new
     Ada.Strings.Bounded.Generic_Bounded_Length (Max => 100);
   use Bounded_Str;

   String1, String2 : Bounded_String;


begin

    String1 := To_Bounded_String ("This is String 1");

    String2 := To_Bounded_String ("This is Bounded String 2");

    Put_Line ("String: " & To_String (String1));
    Put_Line ("String Length: " & Integer'Image (Length (String1)));

    Put_Line ("String: " & To_String (String2));
    Put_Line ("String Length: " & Integer'Image (Length (String2)));

end figure18;
```



```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ gnatmake figure18.adb
x86_64-linux-gnu-gcc-7 -c figure18.adb
x86_64-linux-gnu-gnatbind-7 -x figure18.ali
x86_64-linux-gnu-gnatlink-7 figure18.ali
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure18
String: This is String 1
String Length:  16
String: This is Bounded String 2
String Length:  24
```

1.19

```
with Ada.Strings;            use Ada.Strings;
with Ada.Strings.Unbounded;  use Ada.Strings.Unbounded;
with Ada.Text_IO;            use Ada.Text_IO;

procedure figure19 is
    String1, String2 : Unbounded_String;

begin

    String1 := To_Unbounded_String ("This is String 1");
    String2 := To_Unbounded_String ("This is String 2");

    Put_Line ("String1: " & To_String (String1));
    Put_Line ("String1 Length: " & Integer'Image (Length (String1)));

    Put_Line ("String2: " & To_String (String2));
    Put_Line ("String2 Length: " & Integer'Image (Length (String2)));

    String2 := To_Unbounded_String ("This is String 2 Longer");

    Put_Line ("String1 Longer: " & To_String (String2));
    Put_Line ("String1 Longer Length: " & Integer'Image (Length
(String2)));

end figure19;
```



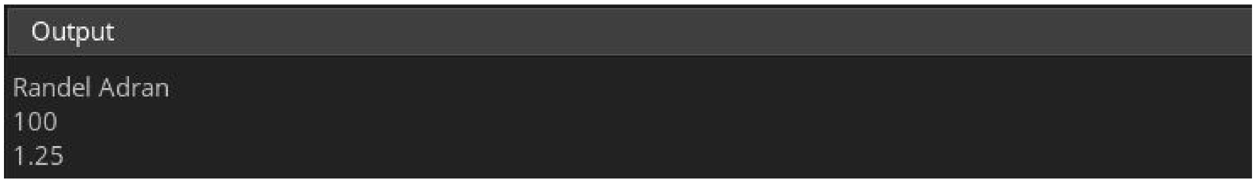
```
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$ ./figure19
String1: This is String 1
String1 Length:  16
String2: This is String 2
String2 Length:  16
String1 Longer: This is String 2 Longer
String1 Longer Length:  23
mclaina1@DESKTOP-JKB5RJR:~/development/csc3310/term-project$
```

GML

2.1

```
/// Object: obj_player
/// Event: Create

//Declare player name.
name = "Randel Adran";

//Declare health and max health.
hp = 100;
hp_max = hp;

//Declare armor scaled based on level.
armor_base = 0.5;
armor_level = 3;
armor = armor_base + (armor_level * 0.25);
```

```
/// Object: obj_player
/// Event: Key Press - T

show_debug_message(name);
show_debug_message(hp);
show_debug_message(armor);
```



```
Randel Adran
100
1.25
```

2.2

```
/// Object: obj_player
/// Event: Create

//Initialize player combat stats.
damage = 4;
sword = noone; //Keyword that means "undefined id"
```

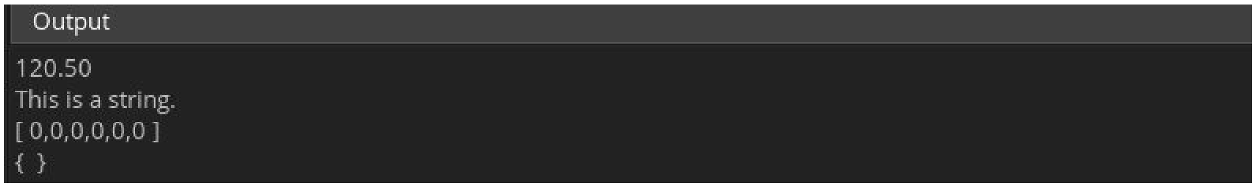
```
/// Object: obj_player
/// Event: Key Press - Space
```

```
//Create the player's sword.
sword = instance_create_layer(x, y, layer, obj_sword);
var player = id;
with (sword)
{
      damage = player.damage;
      image_angle = 90;
      attack_timer = 5;
}
```

No output.

2.3

```
//Declaring a number and a string.
my_number = 120;
my_stringA = "String";
my_stringB = "5";

//Testing comparisons between them.
show_debug_message(my_number == my_stringA); //FALSE
//show_debug_message(my_number >= my_stringA); //ERROR

show_debug_message(my_number >= real(my_stringB)); //TRUE
show_debug_message(my_number >= my_stringB); //TRUE
//my_stringB is implicitly cast to an integer.
//In this case: 120 is larger than 5.

show_debug_message(string(my_number) >= my_stringA); //FALSE
//In this case: "120" is shorter than "String".

show_debug_message(my_number + real(my_stringB)); //125
show_debug_message(my_number + my_stringB); //125
//In this case: 120 + 5 is 125.

//show_debug_message(my_number + my_stringA); //ERROR
show_debug_message(string(my_number) + my_stringA); //120String
//In this case: "120" concatenated with "String".
```



```
120.50
This is a string.
[ 0,0,0,0,0,0 ]
{ }
```

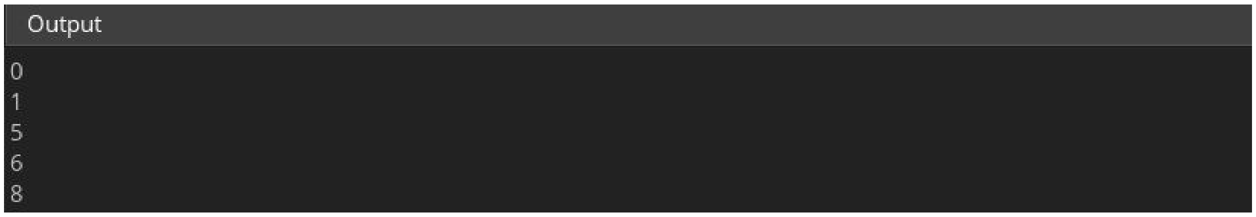
2.4

```
/// Object: obj_player
/// Event: Step

//Get keyboard inputs for movement, and calculate movement direction.
x_axis = keyboard_check(vk_right) - keyboard_check(vk_left);
y_axis = keyboard_check(vk_down) - keyboard_check(vk_up);
angle = point_direction(0, 0, x_axis, y_axis);
```

No output.

2.5

```
enum number
{
      A,          //0
      B,          //1
      C = 5,      //5
      D,          //6
      E = 8.9     //8
}

show_debug_message(number.A);
show_debug_message(number.B);
show_debug_message(number.C);
show_debug_message(number.D);
show_debug_message(number.E);
```

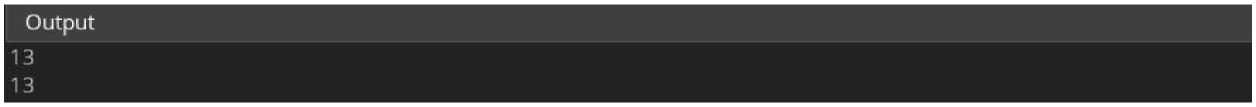


```
0
1
5
6
8
```

2.6

```
//Declare an enumerator.
enum number
{
      A = 1,
      B = 3,
      C = 4
}
```

```
//This expression using the enumerator is identical to the second
expression.
valueA = number.A + (number.B * number.C);
valueB = 1 + (3 * 4);

show_debug_message(valueA);
show_debug_message(valueB);
```



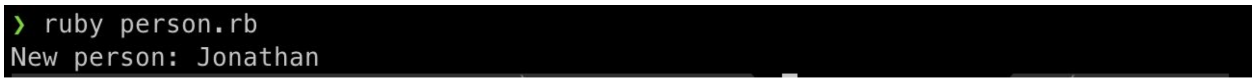
```
13
13
```

Ruby

3.1 person.rb

```
class Person
    def initialize ( name )
        @@name = name
        puts "New person: " +  name
    end

    def get_name
        @@name
    end
end

hacker = Person.new( "Jonathan" )
```

Output:

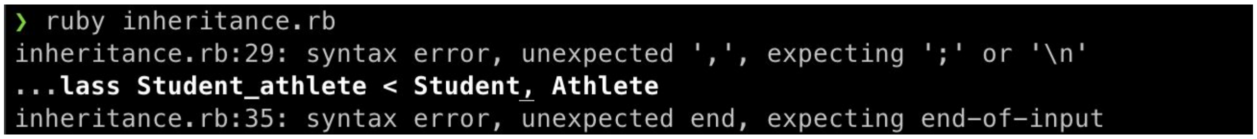


```
❯ ruby person.rb
New person: Jonathan
```

3.2 inheritance.rb

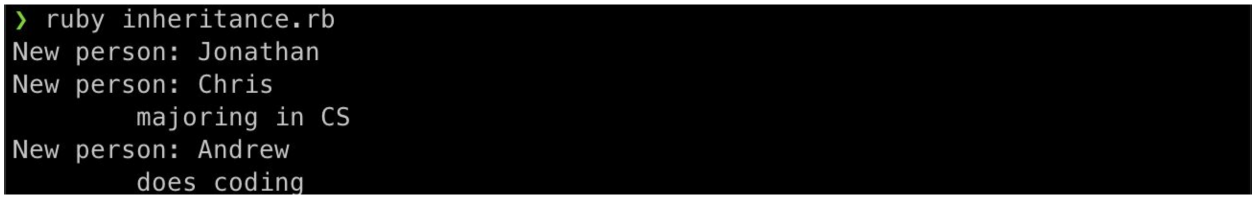
```
require_relative "person"

class Athlete < Person
    def initialize( name, sport )
        super( name )
        @@sport = sport
        puts "does " + sport
    end

    def get_sport
        @@sport
    end
end

class Student < Person
    def initialize( name, major )
        super( name )
        @@major = major
        puts "majoring in " + major
    end
    def get_major
        @@major
    end
end

# Faulty multiple-inheritance class definition
class Student_athlete < Student, Athlete
    def initialize( name, major, sport )
        super.super( name )
        super( major )
        super( sport )
    end
end

moroney = Student.new( "Chris", "CS" )
mclain = Student.new( "Andrew", "coding")
```

Output:



```
❯ ruby inheritance.rb
inheritance.rb:29: syntax error, unexpected ',', expecting ';' or '\n'
...lass Student_athlete < Student, Athlete
inheritance.rb:35: syntax error, unexpected end, expecting end-of-input
```

Output without faulty definition:



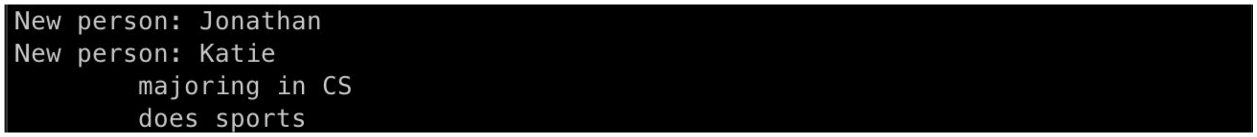
```
❯ ruby inheritance.rb
New person: Jonathan
New person: Chris
        majoring in CS
New person: Andrew
        does coding
```

3.3 mixins.rb

```
require_relative "person"

class Student < Person
    def initialize( name, major )
        super( name )
        @@major = major
        puts "majoring in " + major
    end

    def get_major
        @@major
    end
end
```

```
module Athlete
    def set_sport( sport )
        @@sport = sport
    end

    def get_sport
        if defined?(@@sport) then
            @@sport
        else
            "none"
        end
    end
end


class Student_athlete < Student
    include Athlete
    def initialize( name, major, sport )
        super( name, major )
        set_sport( sport )
    end

    def set_sport( sport )
        @@sport = sport
    end

    def get_sport
        if defined?(@@sport) then
            @@sport
        else
            "none"
        end
    end
end

me = Student_athlete.new("Katie", "CS", "sports")
```

Output:

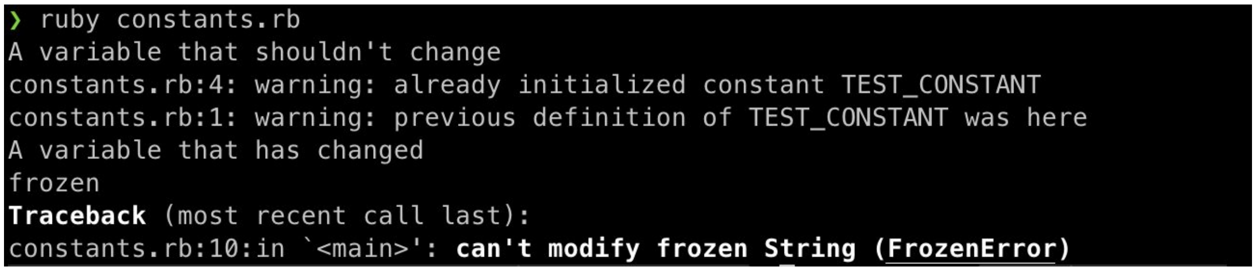


```
New person: Jonathan
New person: Katie
        majoring in CS
        does sports
```

3.5 constants.rb

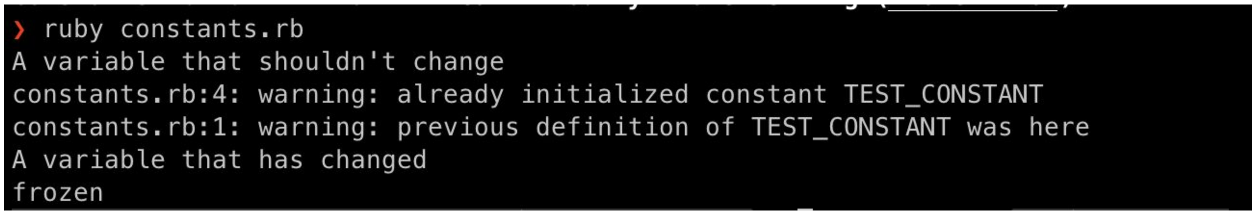
```
TEST_CONSTANT = "A variable that shouldn't change"
puts TEST_CONSTANT

TEST_CONSTANT = "A variable that has changed"
puts TEST_CONSTANT

UNCHANGEABLE = "frozen".freeze
puts UNCHANGEABLE

# trying to modify immutable
UNCHANGEABLE << " melt the freeze"
puts UNCHANGEABLE
```

Output:



```
❯ ruby constants.rb
A variable that shouldn't change
constants.rb:4: warning: already initialized constant TEST_CONSTANT
constants.rb:1: warning: previous definition of TEST_CONSTANT was here
A variable that has changed
frozen
Traceback (most recent call last):
constants.rb:10:in `<main>': can't modify frozen String (FrozenError)
```

Output without trying to modify the immutable object:



```
❯ ruby constants.rb
A variable that shouldn't change
constants.rb:4: warning: already initialized constant TEST_CONSTANT
constants.rb:1: warning: previous definition of TEST_CONSTANT was here
A variable that has changed
frozen
```

Bibliography

[1] *Adacore* [online]. 2020 [accessed. 13 . November 2020]. Retrieved z: https://learn.adacore.com

[2] Castello, Jesus. *Everything You Need to Know About Ruby Constants* [online]. [accessed. 14 . November 2020]. Retrieved z: https://www.rubyguides.com/2017/07/ruby-constants

[3] *GML Objects Documentation* [online]. [accessed. 14 . November 2020]. Retrieved z: https://docs2.yoyogames.com/source/_build/3_scripting/4_gml_reference/objects/index.html

[4] *GML Scope Documentation* [online]. [accessed. 14 . November 2020]. Retrieved z: https://docs2.yoyogames.com/source/_build/3_scripting/3_gml_overview/6_scope.html

[5] *GML Data Types Documentation* [online]. [accessed. 14 . November 2020]. Retrieved z: https://docs2.yoyogames.com/source/_build/3_scripting/3_gml_overview/9_data_types.html

[6] Perrotta, Paolo. *Why Inheritance Sucks (in Ruby, at least)* [online]. [accessed. 14 . November 2020]. Retrieved z: http://ducktypo.blogspot.com/2010/08/why-inheritance-sucks.html

[7] *Ruby Modules Documentation* [online]. [accessed. 14 . November 2020]. Retrieved z: https://www.tutorialspoint.com/ruby/ruby_modules.htm

[8] Sebesta, Robert. *Concepts of Programming Languages*. Pearson. 2015 [accessed. 13 . November 2020].

[9] *Wikipedia* [online]. 2020 [accessed. 13 . November 2020]. Retrieved z: https://en.wikipedia.org/wiki/Ada_(programming_language)