last update: 2nd February 2024

Session: 2023-24

COMP281 Lecture 4

Principles of C and Memory Management

Phil Jimmieson

UNIVERSITY OF LIVERPOOL

Department of Computer Science

Last Lecture

- C Language Basics
  - Basic I/O
  - Operators
  - Decision Making

Today

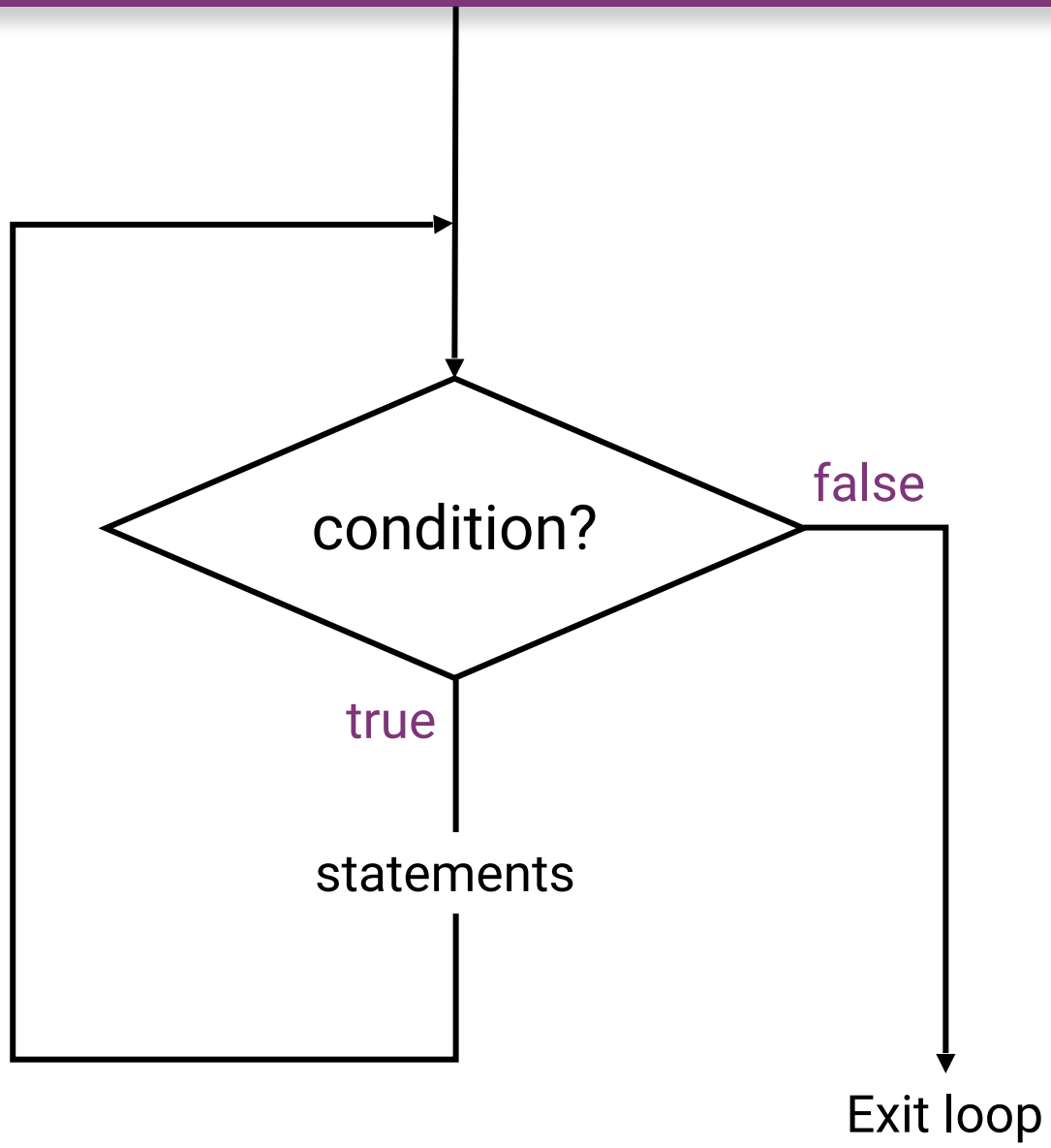
- C Language Basics
  - Basic I/O
  - Operators
  - Decision Making
  - **Loops**
- CodeGrade

Loops

Loops

- `while loop`
- `do… while Loop`
- `for Loop`
- `continue, break, goto`

while Loop



The `while` loop evaluates the test expression before every loop, so it can execute zero times if the condition is initially false. It requires parentheses like the `if`.

```
while (<expression>)
{
    <statements>
}
```

Example 1.1:

```
#include <stdio.h>
int main(void)
{
    int count = 1;
    while ( count <= 3 )
    {
        printf( "%d\n", count );
        count++;
    }
    return 0;
}
```

Output ?

```
Phils-iMac:c-progs phil$
```

Example 1.1:

```
#include <stdio.h>
int main(void)
{
    int count = 1;
    while ( count <= 3 )
    {
        printf( "%d\n", count );
        count++;
    }
    return 0;
}
```

Output ?

```
Phils-iMac:c-progs phil$
```

Example 1.3:

```
#include <stdio.h>
int main(void)
{
    int count = 6;
    while ( count >= 5 )
    {
        printf( “%d\n”, count );
        count ++;
    }
    return 0;
}
```

Output ?

```
Phils-iMac:c-progs phils
```

INFINITE LOOP!

OR IS IT…?

Example 1.3:

```
#include <stdio.h>
int main(void)
{
    int count = 6;
    while ( count >= 5 )
    {
        printf( “%d\n”, count );
        count ++;
    }
    return 0;
}
```

Output ?

```
2147222927
2147222928
2147222929
2147222930
2147222931
2147222932
2147222933
2147222934
2147222935
2147222936
2147222937
2147222938
2147222939
2147222940
2147222941
2147222942
2147222943
2147222944
2147222945
2147222946
2147222947
2147222948
2147222949
```

Example 1.3B:

```
#include <stdio.h>

#include <limits.h>

int main(void)

{

    int theValue = INT_MAX;

    printf("INT_MAX = %d\n”,theValue);

    return 0;

}
```

Example 1.4:

```
#include <stdio.h>
int main(void)
{
    int count = 5;
    while ( count <= 6 )
    {
        printf( “%d ", count );
        count --;
    }
    return 0;
}
```

Output ?

```
Phils-iMac:~ phil$
```

Use of Logical operators in `while` loops

```
while( num1 <= 10 && num2 <= 10 )

while( num1 <= 10 || num2 <= 10 )

while( num1 != num2 && num1 <= num2 )

while( num1 != 10 || num2 >= num1 )
```

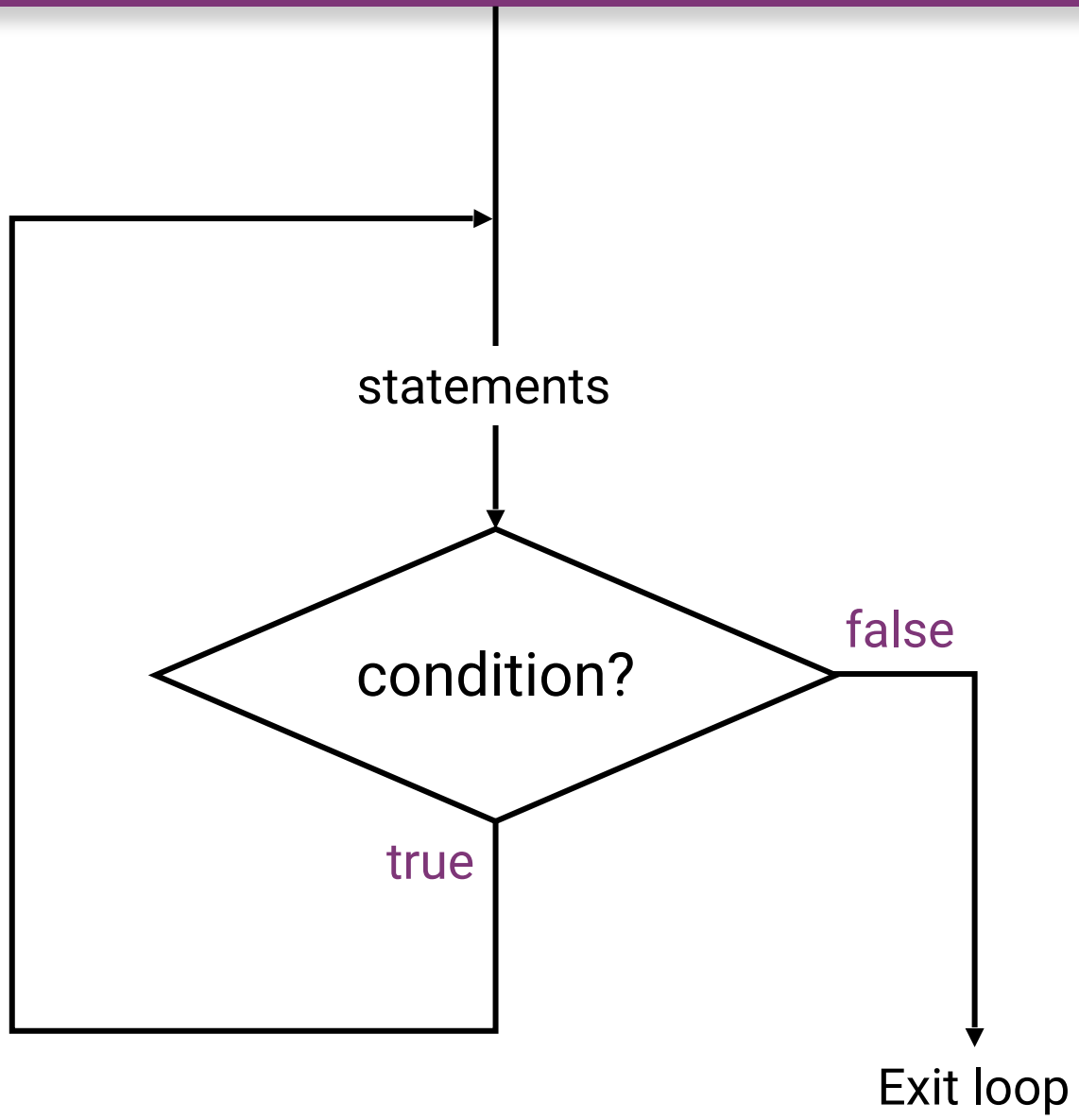
Example 1.5

```c
#include <stdio.h>
int main(void)
{
    int i=1, j=1;
    while ( i <= 4 || j <= 3 )
    {
        printf( "%d %d\n", i, j );
        i++;
        j++;
    }
    return 0;
}
```

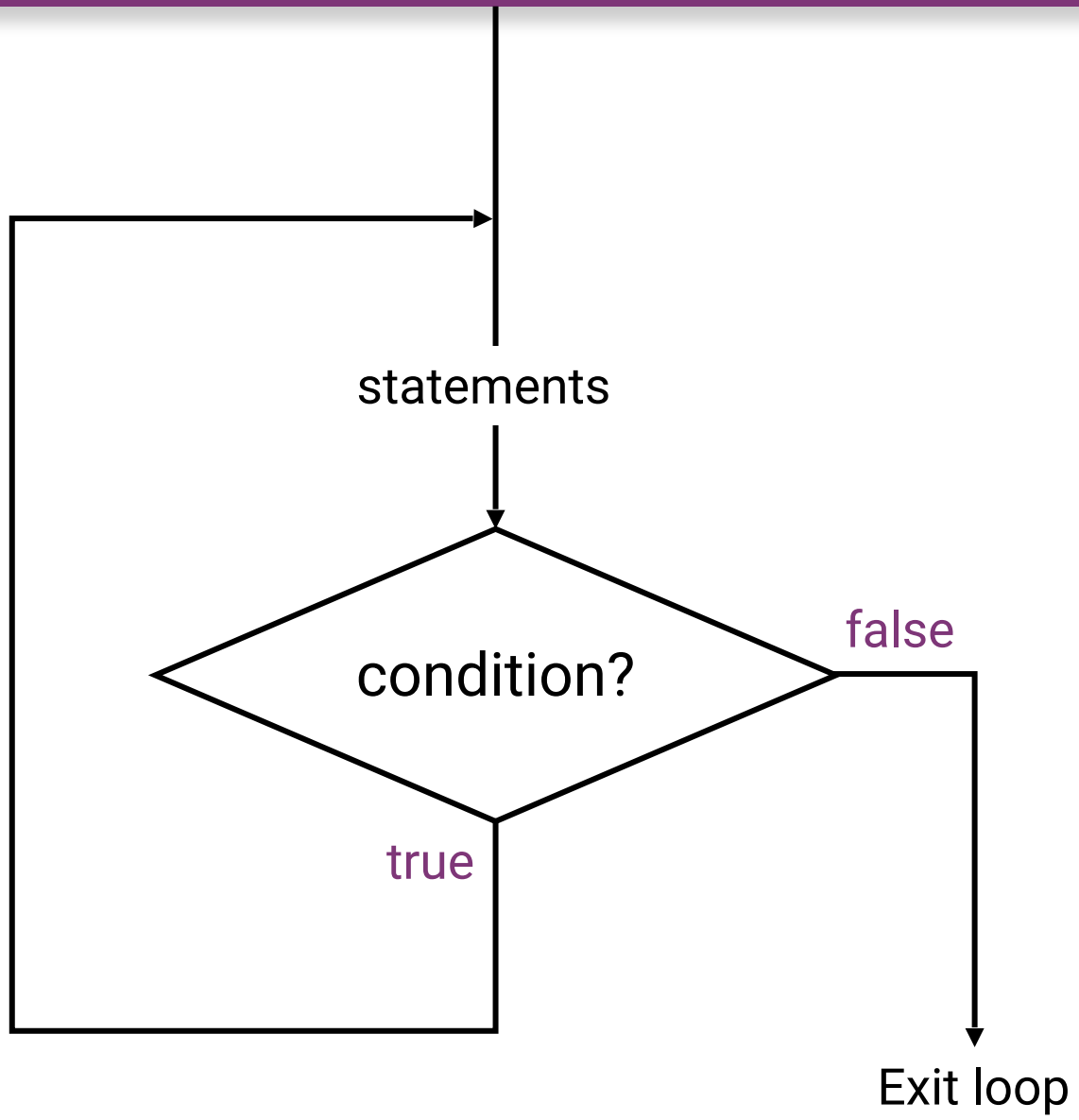
Output ?

```
Phils-iMac:C programs phil$
```

do… while Loop



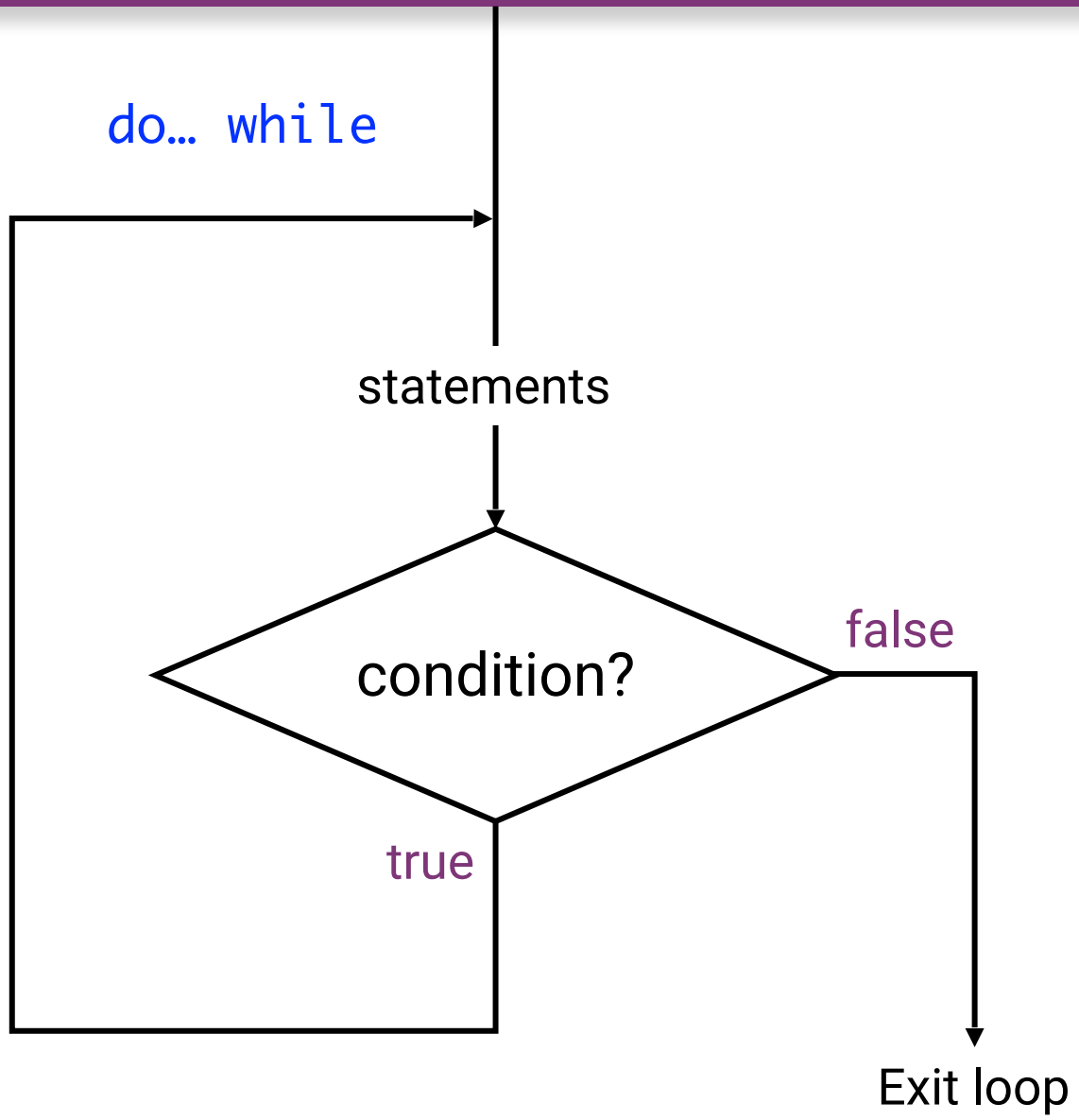
do… while Loop

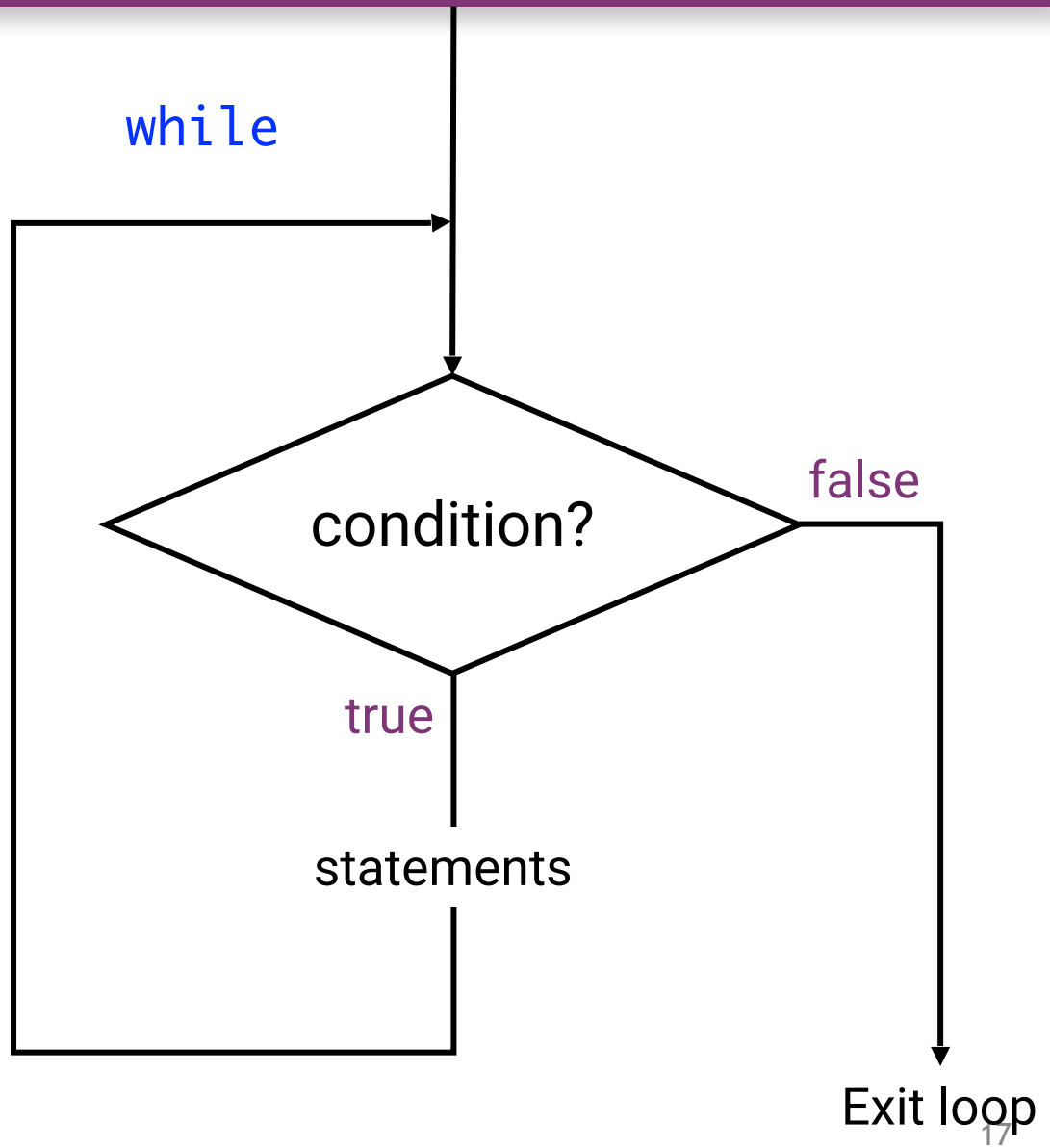


Like a `while`, but with the test condition at the bottom of the loop. The loop body will always execute at least once. The `do-while` is an unpopular area of the language, most everyone tries to use the straight `while` if at all possible.

```
do
{
    <statements>
} while (<expression>)
```

do… while vs while





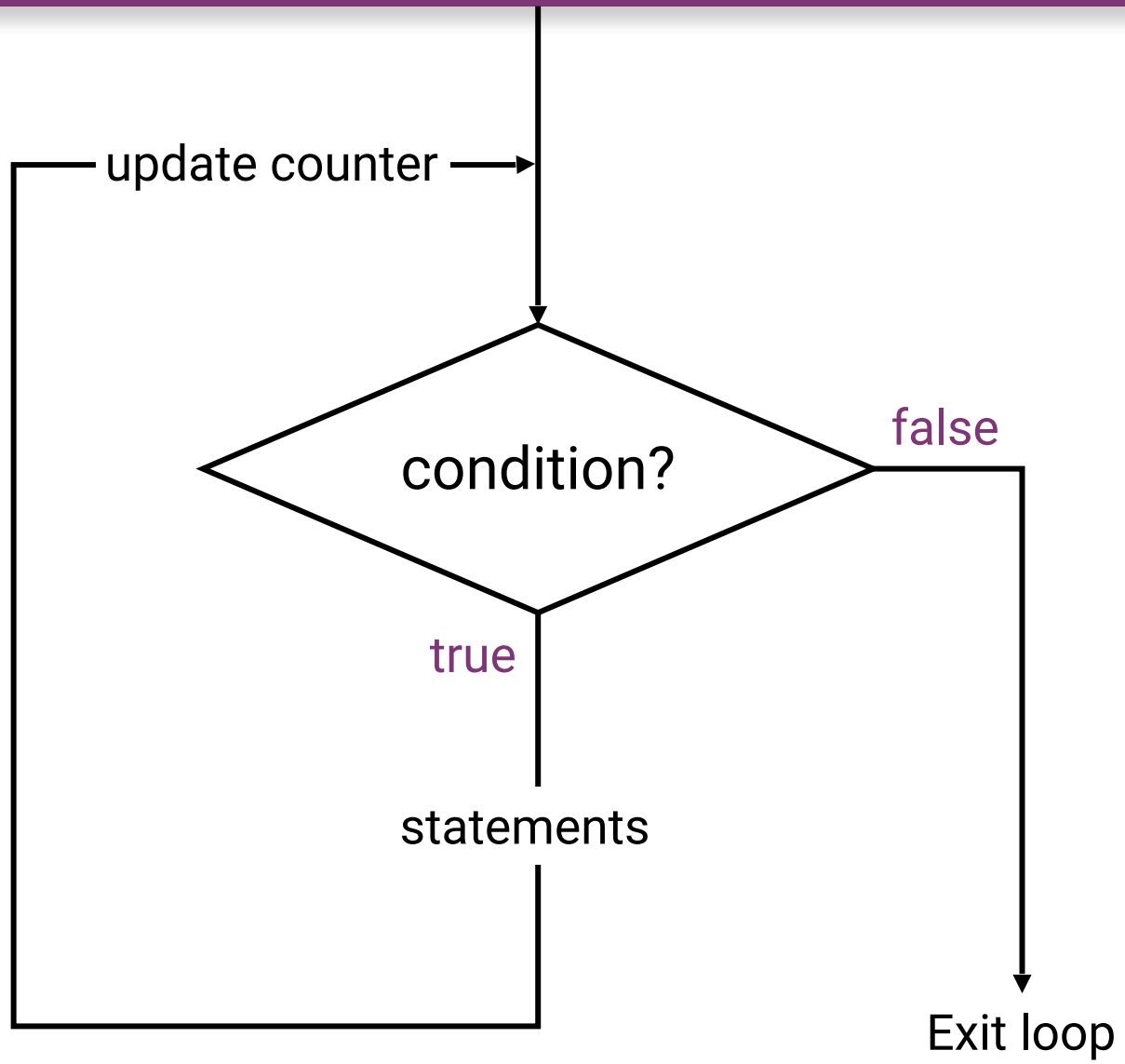
Example 2:

```
#include <stdio.h>
int main(void)
{
    int i = 0;
    do
    {
        printf( "%d\n", i);
        i++;
    } while( i<=3 );
    return 0;
}
```

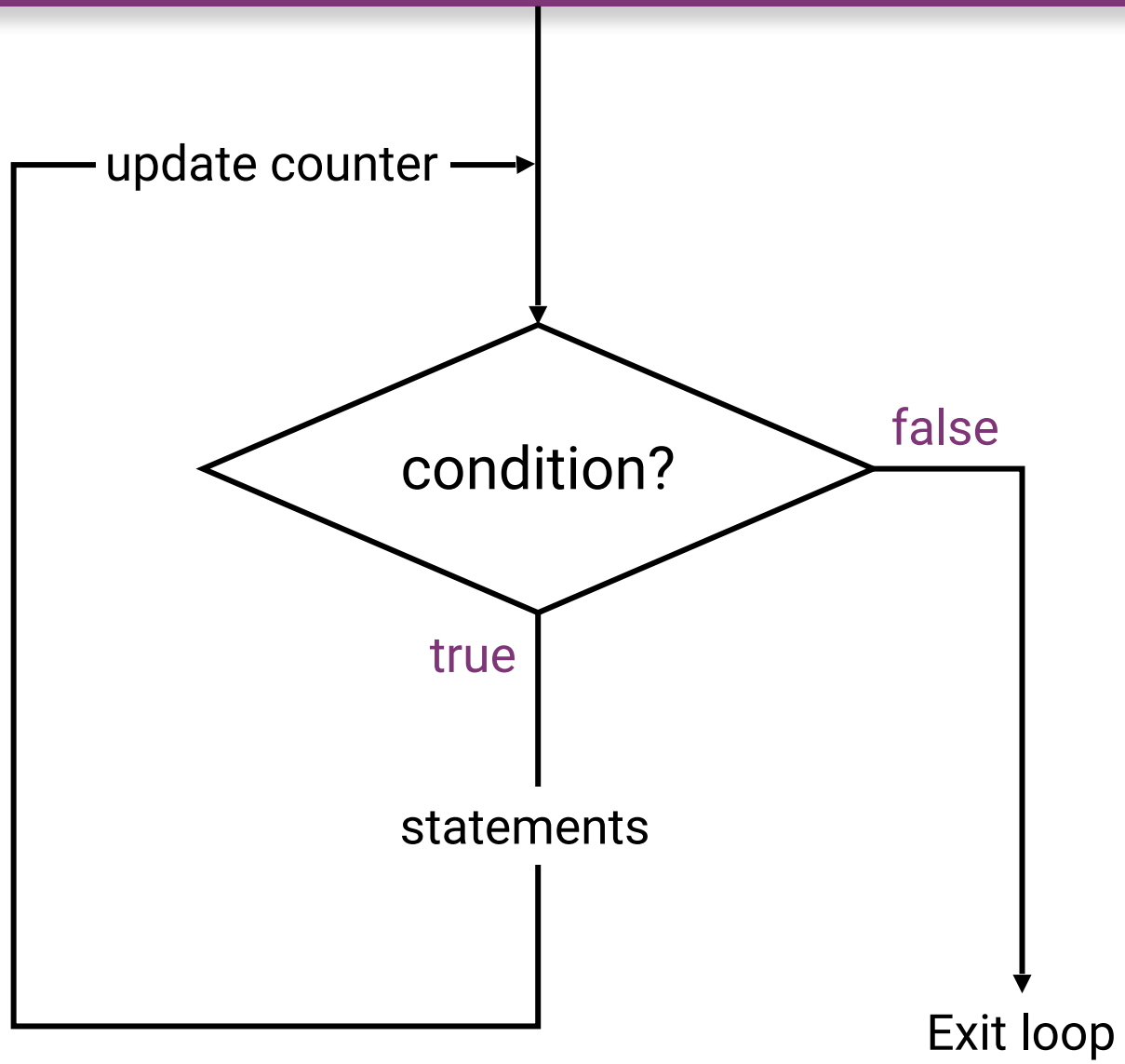
Output ?

```
Phils-iMac:C programs phil$
```

for Loop



`for` Loop



The `for` loop is the most general looping construct.

The loop header contains three parts: an ***initialisation***, a ***continuation condition***, and an ***action***.

```
for (<initialisation>;<continuation>;<action>)
{
    <statements>
}
```

The ***initialisation*** is executed once before the body of the loop is entered. The loop continues to run as long as the ***continuation condition*** remains true (like a `while`).

After every execution of the loop, the ***action*** is executed.

Example 3.1:

```
#include <stdio.h>
int main(void)
{
    int i;
    for ( i=1; i<=3; i++ )
    {
        printf( "%d\n", i );
    }
    return 0;
}
```

Output ?

```
Phils-iMac:C programs phil$
```

Various forms of `for` loop:

1.

```
for ( num=10; num<20; num=num+1 )
```

2.

```
int num=10;
for( ; num<20; num++ )
```

3.

```
for( num=10 ; num<20; )
{
    // statements
    num++;
}
```

4.

```
int num=10;
for( ; num<20; )
{
    //Statements
    num++;
}
```

5.

```
for( num=20; num>10; num--)
```

Example 3.2 – Nested `for` Loop:

```
#include<stdio.h>
int main(void)
{
    for( int i=0; i<2; i++)
    {
        for( int j=0; j<4; j++)
        {
            printf( "%d,%d\n", i, j );
        }
    }
    return 0;
}
```

Output

??

Example 3.2 – Nested `for` Loop:

```
#include<stdio.h>
int main(void)
{
    for( int i=0; i<2; i++)
    {
        for( int j=0; j<4; j++)
        {
            printf( "%d,%d\n", i, j );
        }
    }
    return 0;
}
```

Output

```
0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3
```

Example 3.3 – Multiple initialisation inside `for` Loop:

```
#include<stdio.h>
int main(void)
{
    int i, j;
    for( i=1, j=1; i<3 || j<5; i++, j++ )
    {
        printf("%d, %d\n",i,j);
    }
    return 0;
}
```

Output

??

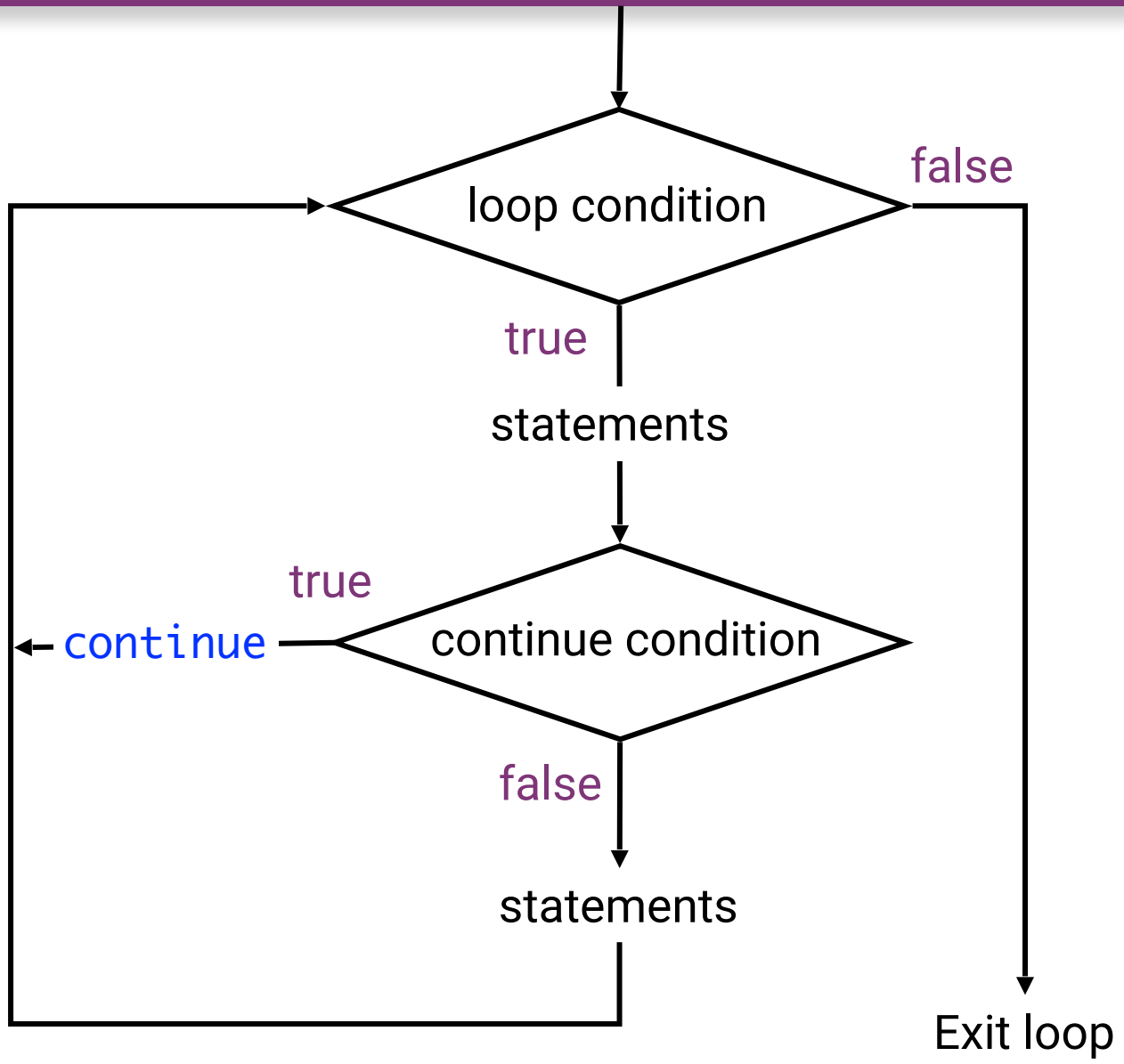
Example 3.3 – Multiple initialisation inside `for` Loop:

```
#include<stdio.h>
int main(void)
{
    int i, j;
    for( i=1, j=1; i<3 || j<5; i++, j++ )
    {
        printf("%d, %d\n",i,j);
    }
    return 0;
}
```

Output

```
1, 1
2, 2
3, 3
4, 4
```

continue



The `continue` **statement** is used inside **loops**. When a continue statement is encountered inside a loop, control jumps to the beginning of the loop for next iteration, skipping the execution of statements inside the body of loop for the current iteration.

Example 4.1 – `continue` statement inside `for` Loop:

```
#include <stdio.h>
int main(void)
{
    for (int j=0; j<=8; j++)
    {
        if (j==4)
        {
            /* The continue statement is encountered when
             * the value of j is equal to 4.*/
            continue;
        }
        /* This print statement would not execute for the
         * loop iteration where j == 4  because in that case
         * this statement would be skipped.
         */
        printf("%d\n", j);
    }
    return 0;
}
```

Output

??

Example 4.1 – `continue` statement inside `for` Loop:

```
#include <stdio.h>
int main(void)
{
    for (int j=0; j<=8; j++)
    {
        if (j==4)
        {
            /* The continue statement is encountered when
             * the value of j is equal to 4.*/
            continue;
        }
        /* This print statement would not execute for the
         * loop iteration where j == 4  because in that case
         * this statement would be skipped.
         */
        printf("%d\n", j);
    }
    return 0;
}
```

Output

```
0
1
2
3
5
6
7
8
```

Example 4.2 – `continue` statement inside `while` Loop:

```
#include <stdio.h>
int main(void)
{
    int counter = 10;
    while (counter >= 0)
    {
        if (counter == 7)
        {
            counter--;
            continue;
        }
        printf("%d\n", counter);
        counter--;
    }
    return 0;
}
```

Output

??

Example 4.2 – `continue` statement inside `while` Loop:

```
#include <stdio.h>
int main(void)
{
    int counter = 10;
    while (counter >= 0)
    {
        if (counter == 7)
        {
            counter--;
            continue;
        }
        printf("%d\n", counter);
        counter--;
    }
    return 0;
}
```

Output

```
10
9
8
6
5
4
3
2
1
0
```

The print statement is skipped when counter value was 7.

Example 4.3 – `continue` statement inside `do…while` Loop:

```
#include <stdio.h>
int main(void)
{
    int j = 0;
    do
    {
        if ( j == 7 )
        {
            j++;
            continue;
        }
        printf("%d\n", j);
        j++;
    }while( j < 10 );
    return 0;
}
```

Output

??

Example 4.3 – `continue` statement inside `do…while` Loop:

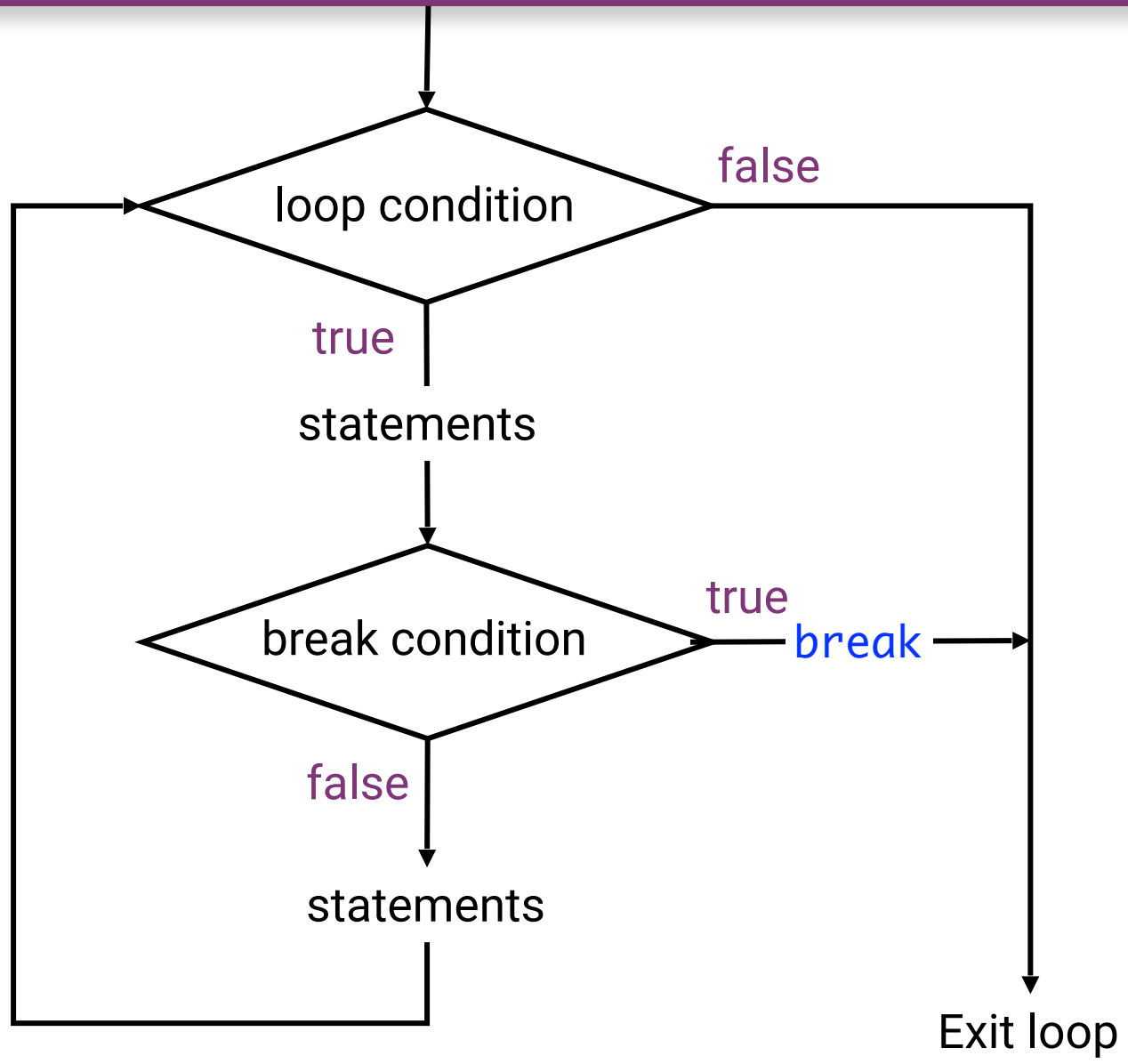
```
#include <stdio.h>
int main(void)
{
    int j = 0;
    do
    {
        if ( j == 7 )
        {
            j++;
            continue;
        }
        printf("%d\n", j);
        j++;
    }while( j < 10 );
    return 0;
}
```

Output

```
0
1
2
3
4
5
6
8
9
```

The print statement is skipped when counter value was 7.

break



- It is used to come out of the **loop** *instantly*. When a **break** statement is encountered inside a loop, the control directly comes out of loop and the loop gets terminated. When used inside a loop it is generally invoked via an **if** statement.
- Break can also be used in the **switch case control structure**. Whenever it is encountered in switch-case block, the switch-case exits immediately.

Example 5.1 – Use of break in a `while` Loop:

```
#include <stdio.h>
int main(void)
{
    int num =0;
    while( num <= 100)
    {
        printf("%d\n", num);
        if (num==2)
        {
            break;
        }
        num++;
    }
    printf("Exit from while-loop");
    return 0;
}
```

Output

??

Example 5.1 – Use of break in a `while` Loop:

```
#include <stdio.h>
int main(void)
{
    int num =0;
    while( num <= 100)
    {
        printf("%d\n", num);
        if (num==2)
        {
            break;
        }
        num++;
    }
    printf("Exit from while-loop");
    return 0;
}
```

Output

```
0
1
2

Exit from while-loop
```

Example 5.2 – Use of break in a `for` Loop:

```
#include <stdio.h>
int main(void)
{
    int var;
    for ( var = 100; var >= 10; var -- )
    {
        printf("%d\n", var);
        if ( var == 99 )
        {
            break;
        }
    }
    printf("Exit from for-loop");
    return 0;
}
```

Output

??

Example 5.2 – Use of break in a `for` Loop:

```
#include <stdio.h>
int main(void)
{
    int var;
    for ( var = 100; var >= 10; var -- )
    {
        printf("%d\n", var);
        if ( var == 99 )
        {
            break;
        }
    }
    printf("Exit from for-loop");
    return 0;
}
```

Output

```
100
99

Exit from for-loop
```

Example 5.3 – Use of break in a `switch-case`:

```
int main(void)
{
    int num;
    printf("Enter value of num: ");
    scanf("%d",&num);
    switch(num)
    {
        case 1:
            printf("You have entered value 1.\n");
            break;
        case 2:
            printf("You have entered value 2.\n");
            break;
        case 3:
            printf("You have entered value 3.\n");
            break;
        default:
            printf("Input value is other than 1, 2 & 3.\n");
    }
    return 0;
}
```

Example 5.3 – Use of break in a `switch-case`:

```
int main(void)
{
    int num;
    printf("Enter value of num: ");
    scanf("%d",&num);
    switch(num)
    {
        case 1:
            printf("You have entered value 1.\n");
            break;
        case 2:
            printf("You have entered value 2.\n");
            break;
        case 3:
            printf("You have entered value 3.\n");
            break;
        default:
            printf("Input value is other than 1, 2 & 3.\n");
    }
    return 0;
}
```

Output

```
Enter value of num: 2
You have entered value 2.
```

Example 5.3 – Use of break in a `switch-case`:

```
int main(void)
{
    int num;
    printf("Enter value of num: ");
    scanf("%d",&num);
    switch(num)
    {
        case 1:
            printf("You have entered value 1.\n");
            break;
        case 2:
            printf("You have entered value 2.\n");
            break;
        case 3:
            printf("You have entered value 3.\n");
            break;
        default:
            printf("Input value is other than 1, 2 & 3.\n");
    }
    return 0;
}
```

Example 5.3 – Use of break in a `switch-case`:

```
int main(void)
{
    int num;
    printf("Enter value of num: ");
    scanf("%d",&num);
    switch(num)
    {
        case 1:
            printf("You have entered value 1.\n");
            break;
        case 2:
            printf("You have entered value 2.\n");
            break;
        case 3:
            printf("You have entered value 3.\n");
            break;
        default:
            printf("Input value is other than 1, 2 & 3.\n");
    }
    return 0;
}
```

Output

```
Enter value of num: 999
Input value is other than 1, 2 & 3
```

Example 5.3b – Use of break in a `switch-case`:

```
int main(void)
{
    int num;
    printf("Enter value of num: ");
    scanf("%d",&num);
    switch(num)
    {
        case 1:
            printf("You have entered value 1.\n");
            break;
        case 2:
            printf("You have entered value 2.\n");
            break;
        case 3:
            printf("You have entered value 3.\n");
            break;
        default:
            printf("Input value is other than 1, 2 & 3.\n");
    }
    return 0;
}
```

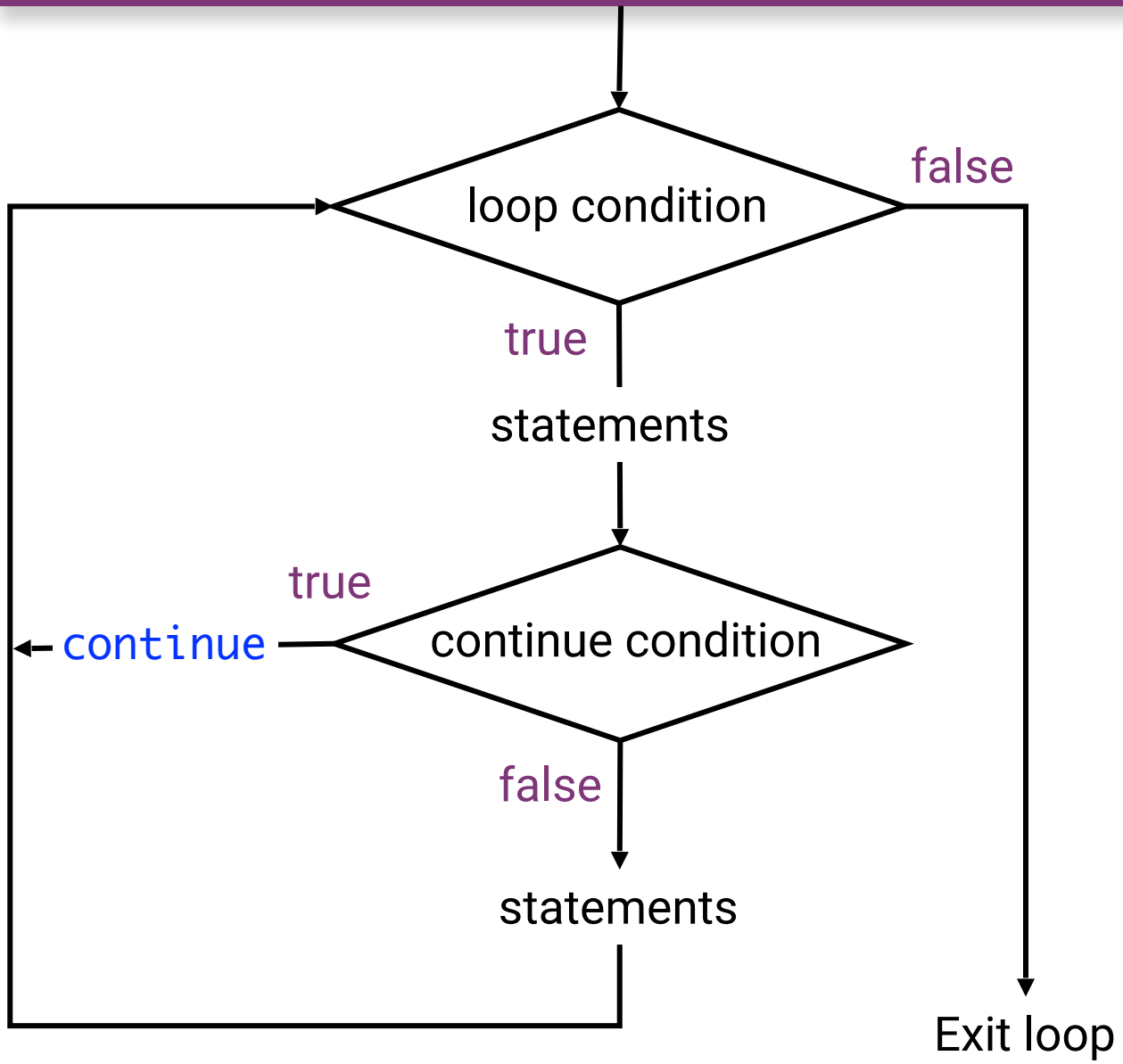
Example 5.3b – Use of break in a `switch-case`:

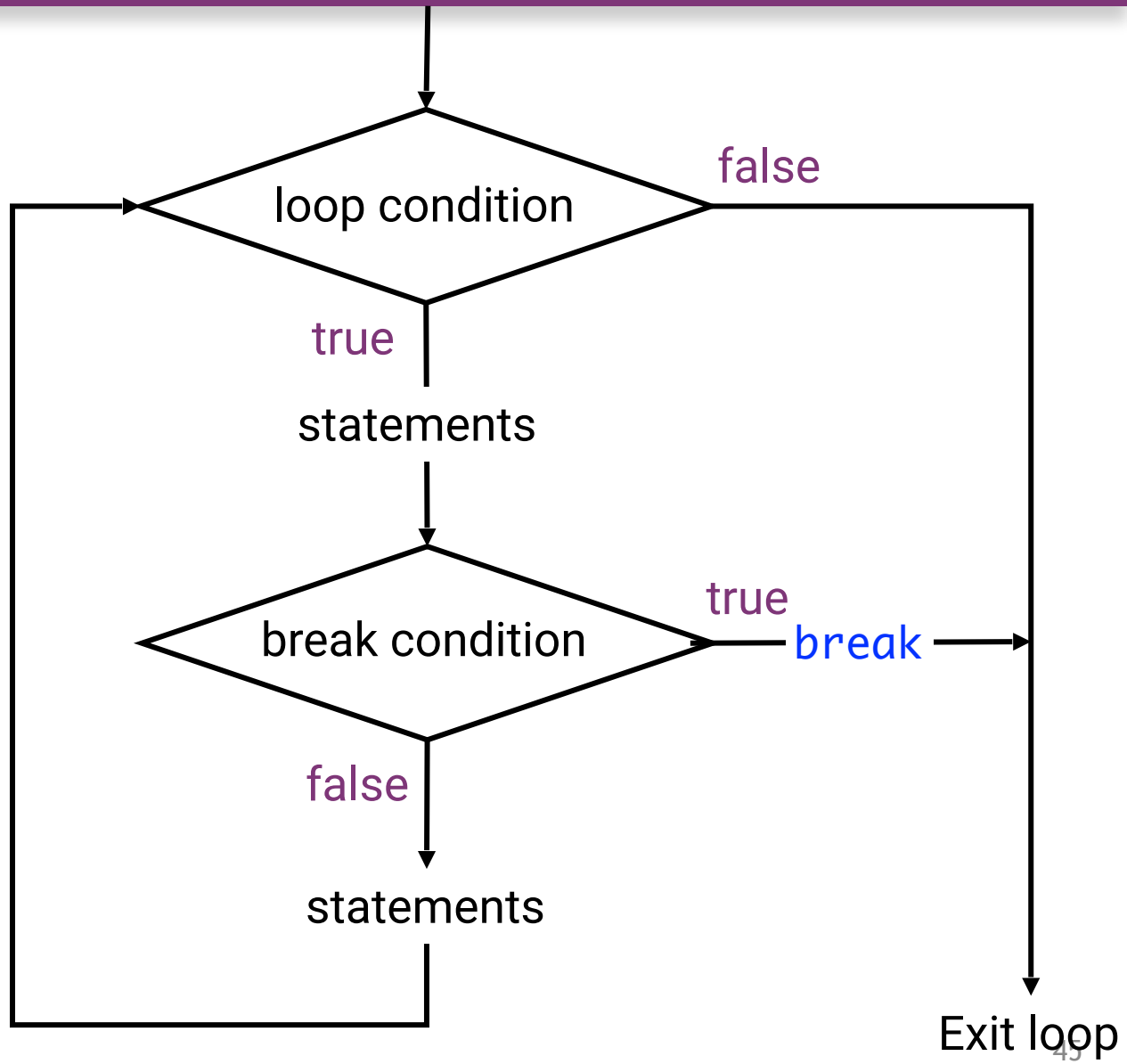
```
int main(void)
{
    int num;
    printf("Enter value of num: ");
    scanf("%d",&num);
    switch(num)
    {
        case 1:
            printf("You have entered value 1.\n");
            break;
        case 2:
            printf("You have entered value 2.\n");
            break;
        case 3:
            printf("You have entered value 3.\n");
            break;
        default:
            printf("Input value is other than 1, 2 & 3.\n");
    }
    return 0;
}
```

Output

```
Enter value of num: 2
You have entered value 2.
You have entered value 3.
```

continue vs break





goto

```
goto <label_name>;
…
…
<label_name>: <statements>
```

- When a `goto` statement is encountered, the control **jumps directly** to the label mentioned in the goto statement.
- The goto statement is rarely used because it makes the program confusing, less readable and complex. Also, when this is used, the control of the program won't be easy to trace, hence it makes testing and debugging difficult.

Example 6 – Use goto:

```
#include <stdio.h>
int main(void)
{
    int sum=0;
    for( int i = 0; i <= 10; i++ )
    {
        sum = sum + i;
        if( i==5 ){
            goto addition;
        }
    }
    addition:
    printf( "%d\n", sum );
    return 0;
}
```

Output

??

Example 6 – Use goto:

```
#include <stdio.h>
int main(void)
{
    int sum=0;
    for( int i = 0; i <= 10; i++ )
    {
        sum = sum + i;
        if( i==5 ){
            goto addition;
        }
    }
    addition:
    printf( "%d\n", sum );
    return 0;
}
```

Output

```
15
```

Example 6 – Use `goto`:

```
#include <stdio.h>
int main(void)
{
    int sum=0;
    for( int i = 0; i <= 10; i++ )
    {
        sum = sum + i;
        if( i==5 ){
            goto addition;
        }
    }
    addition:
    printf( "%d\n", sum );
    return 0;
}
```

Output

15

In this example, we have a label addition and when the value of i (inside loop) is equal to 5 then we are jumping to this label using `goto`. This is the reason the sum is displaying the sum of numbers up to 5 even though the loop is set to run from 0 to 10.

CodeGrade

CodeGrade

- Programming exercises
- Assignment submissions

Lab Work (week 2)

Due at 2023-02-20 23:59

codegrade

Latest submission

Create submission

Assignment description

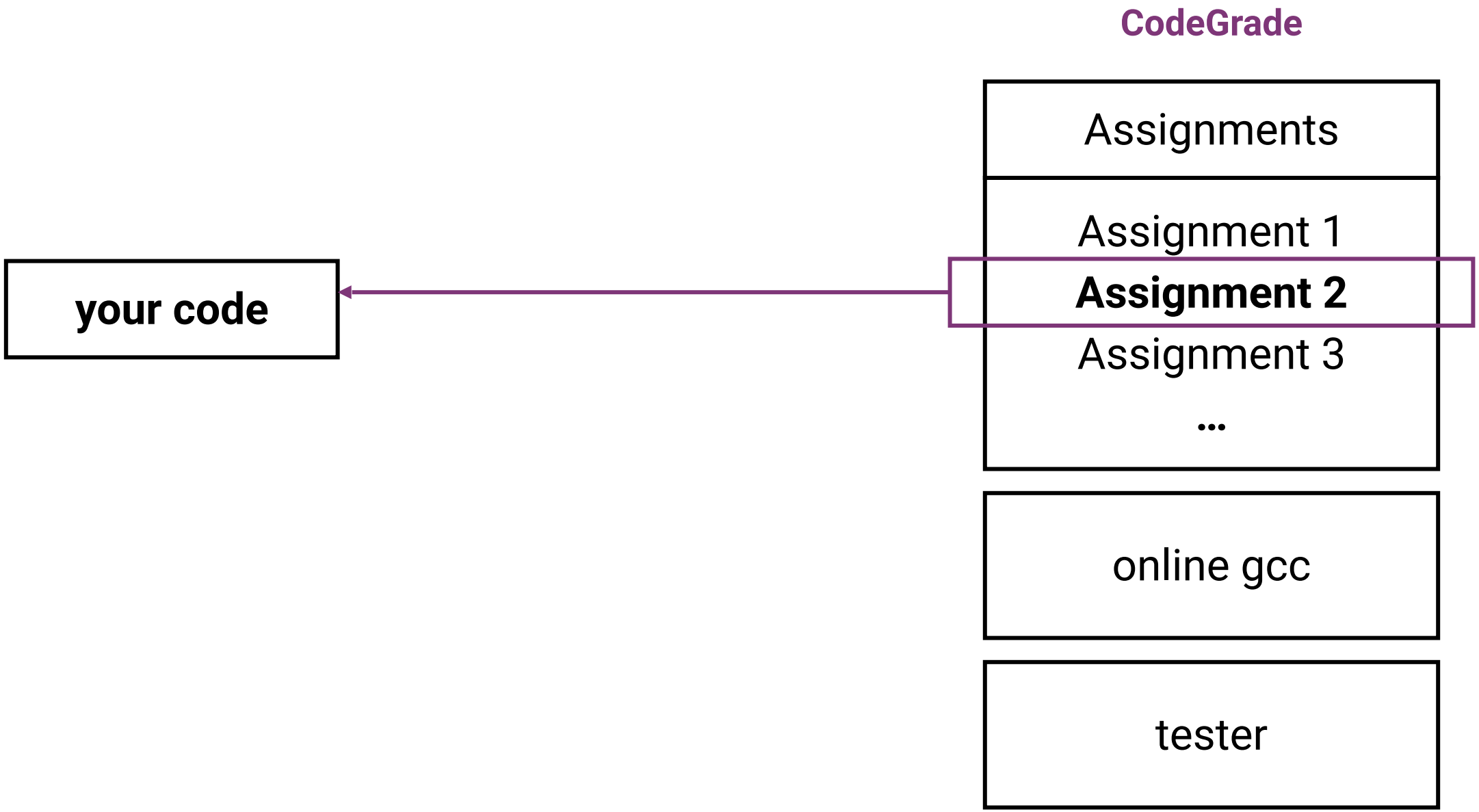
How does CodeGrade work?

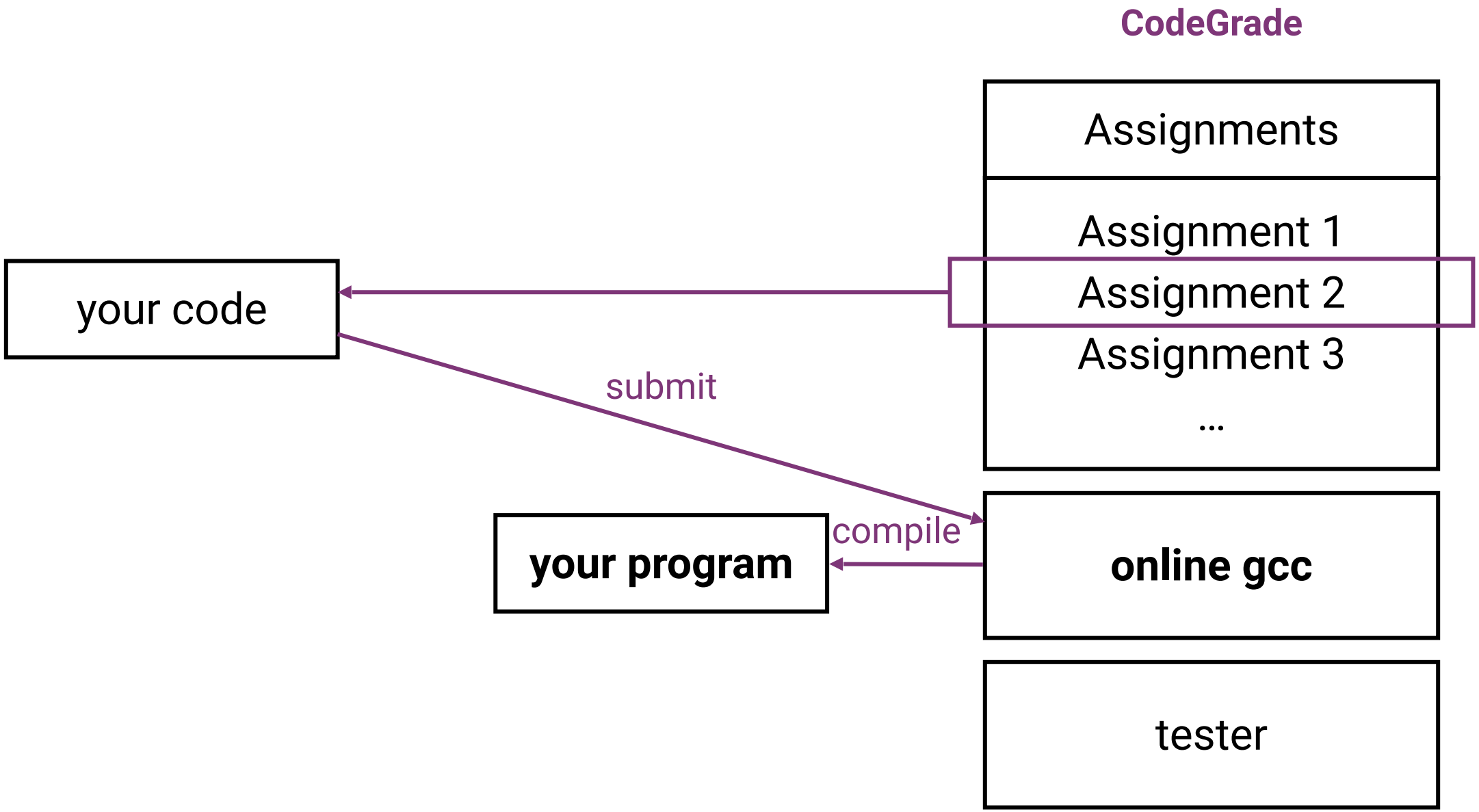
CodeGrade

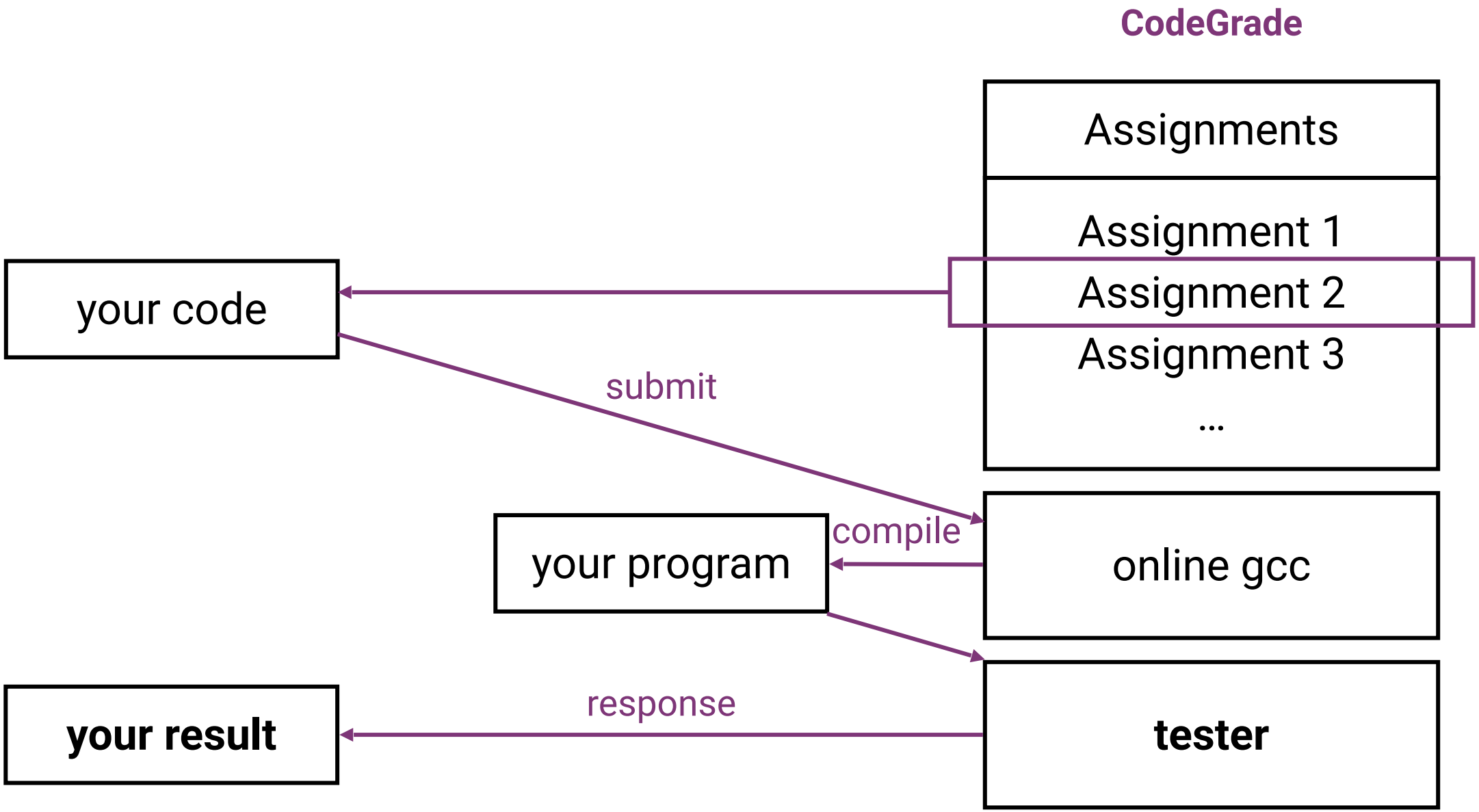
Assignments
Assignment 1 Assignment 2 Assignment 3 ...

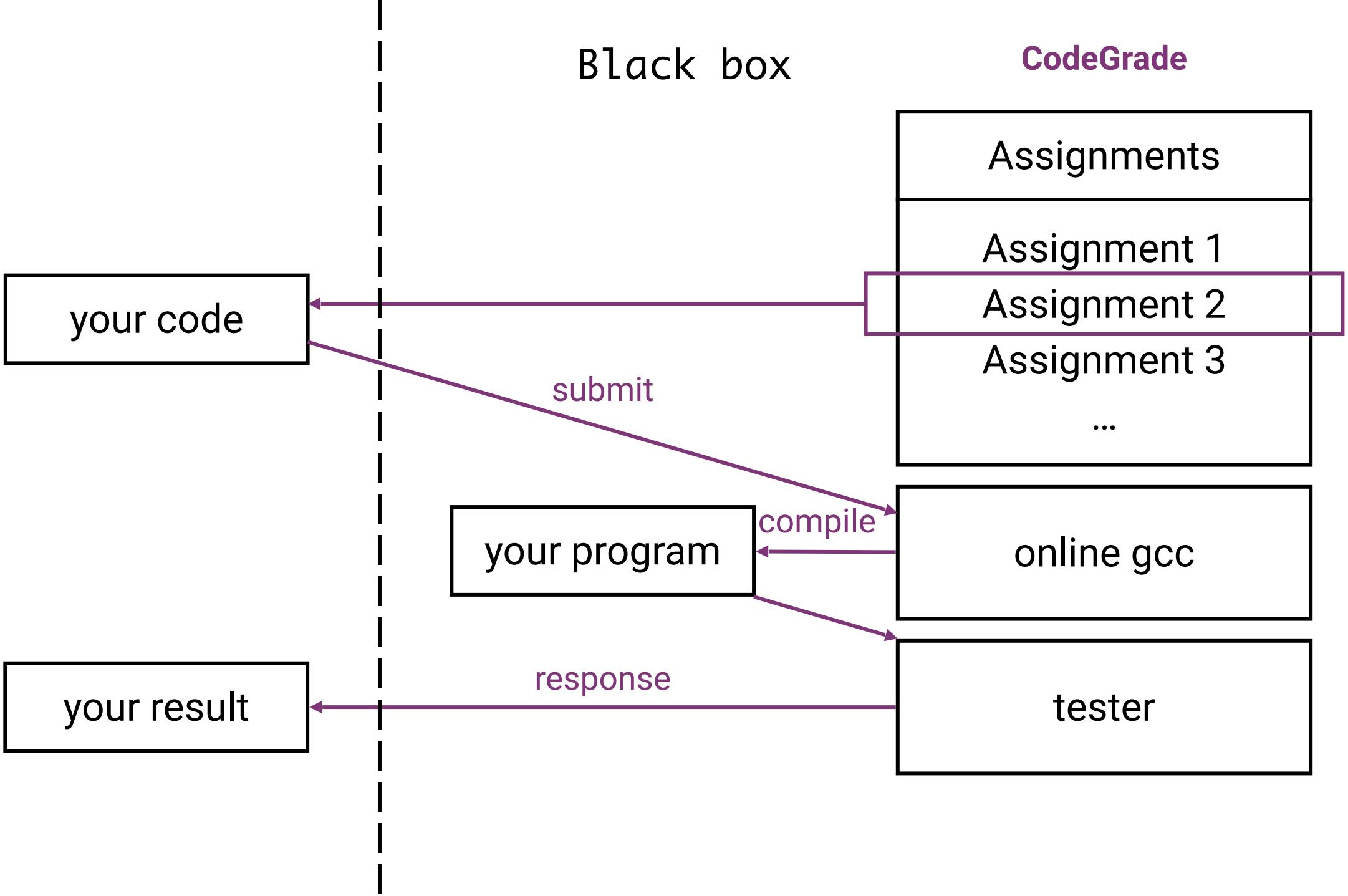
online gcc

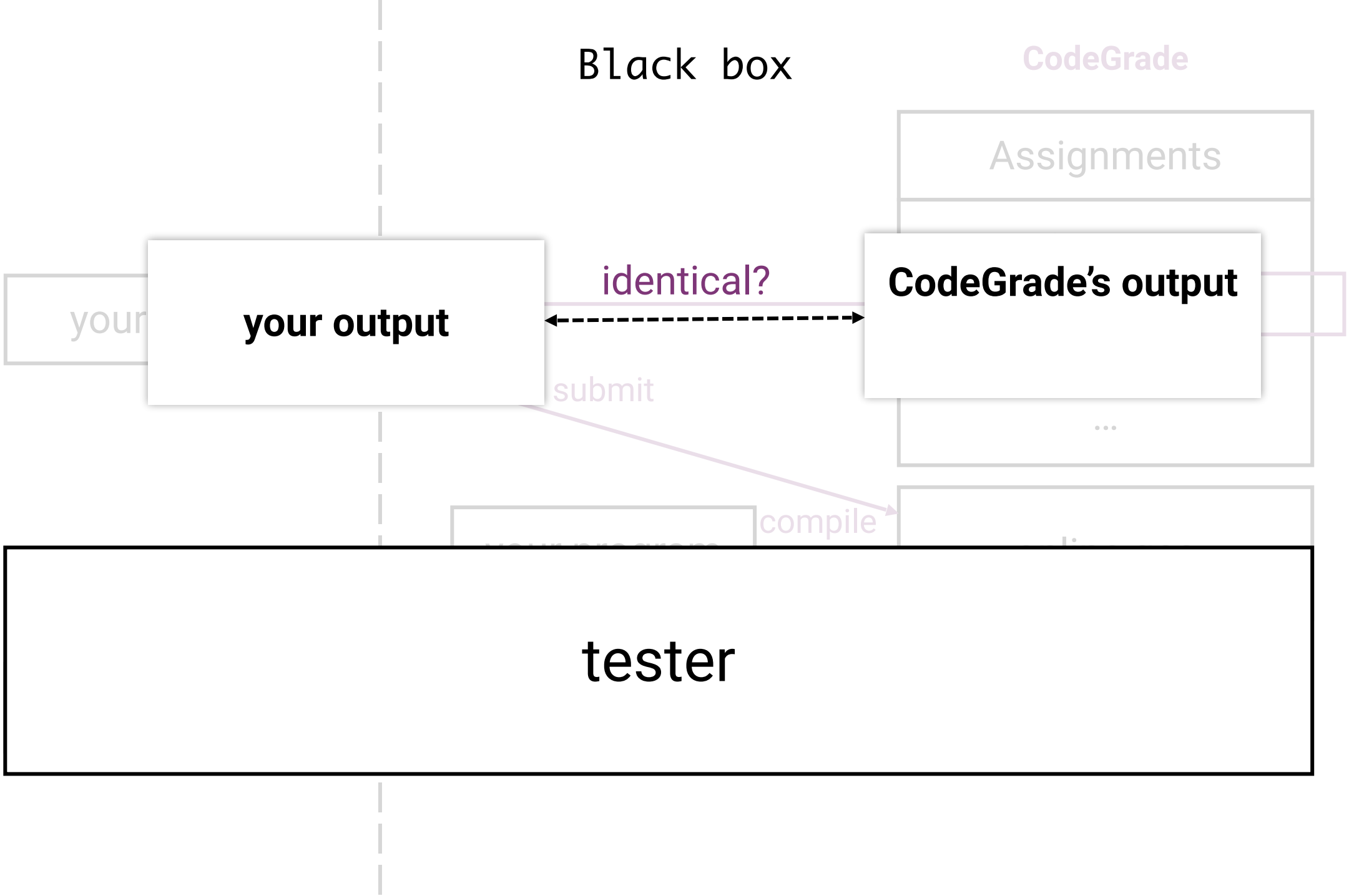
tester


CodeGrade
Assignments
Assignment 1
Assignment 2
Assignment 3
...
your code
online gcc
tester


CodeGrade
Assignments
Assignment 1
Assignment 2
Assignment 3
...
your code
submit
your program
compile
online gcc
tester


CodeGrade
Assignments
Assignment 1
Assignment 2
Assignment 3
...
your code
submit
your program
compile
online gcc
your result
response
tester


Black box
CodeGrade
Assignments
Assignment 1
Assignment 2
Assignment 3
...
your code
submit
your program
compile
online gcc
your result
response
tester


Black box
CodeGrade
Assignments
your
your output
identical?
CodeGrade's output
submit
...
compile
tester

Reproducing CodeGrade

1. Prepare `test.in`

   a text file with input in the format specified in assignment

2. Compile

   ```
   % gcc YOURCODE.c
   ```

3. Run your code 'piping' the input into it

   ```
   % ./a.out < test.in > test.out
   ```

This generates the output in the file `test.out`.

Inspect it to see if there are any unexpected elements in your output

Example 7

```
#include <stdio.h>
int main(void)
{
    int x, y;
    scanf("%i %i", &x, &y);
    printf( "%i %i", x*2, y*2);
    return 0;
}
```

Terminal

```
./a.out < test.in > test.out
```

Reproducing CodeGrade

`-Wall`

enables all compiler warnings...this is **useful**!!

```
gcc -Wall [options][source files][object files][-o output file]
```

Example 8

```
#include <stdio.h>
int main()
{
    printf("Program run!\n");
    int i=10;
}
```

Example 8

```
#include <stdio.h>
int main()
{
    printf("Program run!\n");
    int i=10;
}
```

Terminal

```
gcc -Wall 8.c
8.c:6:9: warning: unused variable 'i' [-Wunused-variable]
    int i=10;
        ^
1 warning generated.
```

Example problem: **A+B+C=?**

- **Description**
  Calculate a+b+c
- **Input**
  Three integers a, b, c (0<=a<=10, 0<=b<=10, 0<=c<=5)
- **Output**
  Output a+b+b
- **Sample Input**
  1 2 3
- **Sample Output**
  6

- **Solution**

```c
#include <stdio.h>
int main(void)
{
        int a, b, c, answer; //declare the input vars and output var

        /* read three integer number as input & store them in a, b & c */
        scanf("%i", &a);
        scanf("%i", &b);
        scanf("%i", &c);

        /* compute the solution and store it in answer */
        answer = a + b + c;

        /* print out the solution as a integer number */
        printf("%i\n", answer);

        return 0;
}
```

Kinds of errors that may occur

- **Runtime Error**: Your program failed during the execution (segmentation fault, floating point exception, etc.). Ensure that your program returns a 0 code to the shell.
- **Time Limit Exceeded**: Your program tried to run and took too much time; this error doesn't allow you to know if your program would eventually reach the correct solution to the problem or not.
- **Memory Limit Exceeded**: Your program tried to use more memory than CodeGrade allows.
- **Output Limit Exceeded**: Your program tried to write out too much information. This usually occurs if it goes into an infinite loop.

Kinds of errors that may occur

- **Runtime Error**: Your program failed during the execution (**segmentation fault**, floating point exception, etc.). Ensure that your program returns a 0 code to the shell.
- **Time Limit Exceeded**: Your program tried to run and took too much time; this error doesn't allow you to know if your program would eventually reach the correct solution to the problem or not.
- **Memory Limit Exceeded**: Your program tried to use more memory than CodeGrade allows.
- **Output Limit Exceeded**: Your program tried to write out too much information. This usually occurs if it goes into an infinite loop.

Kinds of errors that may occur

- **Runtime Error**: Your program failed during the execution (**segmentation fault**, floating point exception, etc.). Ensure that your program returns a 0 code to the

- A "segmentation fault" is a common runtime error.
- When you run your program and the system reports a "segmentation fault", it means your program has attempted to access an area of **memory** that it is not allowed to access.
- In other words, it attempted to stomp on **memory** ground that is **beyond the limits** that the OS has allocated for your program.

usually occurs if it goes into an infinite loop.

Troubleshooting http://web.mit.edu/10.001/Web/Tips/tips_on_segmentation.html

When CodeGrade says “No”

- Before you start debugging... **Read CodeGrade’s response!**
  - There may be useful information there!
  - e.g., is it a 'run-time' error or 'compilation' error?
- Things to do:
  - Read the problem again
    - Are you sure you have interpreted it correctly?

When CodeGrade says “No”

- Things to do:
  - Test it yourself:
    - Did you try different test cases?
    - What **assumptions** did you make about the test cases?
    - Did you assume the test case(s) give you all of the possibilities?
    - Did you try large numbers, small numbers, negative numbers, etc.?
  - Check the output
    - Does the output match exactly?
    - Whitespace may matter if in the middle of the output
    - Don’t print **anything** else to the screen.

When CodeGrade says “Compilation Error”

- “...and it works on my computer”
- That is great, but not sufficient!
  - remember: compilers are different
- But compilation errors are easy to fix: compiler tells you what went wrong, so try using gcc.
- Also, check the output details in CodeGrade
  - it gives you the error message!

When CodeGrade says “Runtime Error”

- Problem: this typically is a segmentation fault.
- Unfortunately, more difficult to resolve
- Approaches:
  - **Testing:** try different inputs until you can reproduce the error.
  - **Print debugging**: use print statements to understand where the problem occurs.
  - **Use a debugger tool**: use the debugger gdb (in later lecture) to find out what causes the error.
- If that does not work, resolve at the lab!

Key Points

- You are free to submit solutions
  - as often as you like.
  - I may make some additional exercises available for you to practice, so go for it.
  - keep your programs as efficient as reasonably possible!

Summary

Today

- C Language Basics
  - **Loop**
- CodeGrade

Next

- Functions