

Lecture 13

# Debugging & Exercises

Gyuhyeon Seo, Doyoun Kim

# Overview

- Debugging
  - Debugging techniques
  - Why use debugger
- Exercise
  - 1. list\_pop.cpp
  - 2. operator\_overloading.cpp
  - 3. simple\_vector\_bug.cpp

# Debugging in Visual Studio Code

Gyuhyeon Seo

Slides from Jongwook Han, Sp24 Computing 1

# Debugging

9/2  
9/9

0800 arctan started  
1000 " stopped - arctan ✓ { 1.2700 9.037847025  
13'00 (032) MP-MC 1.982142000 9.037846995 correct  
(033) PRO 2 2.130476415  
correct 2.130676415  
Relays 6-2 on 033 failed special speed test  
on Relay 10.000 test.  
Relays changed  
Started Cosine Tape (Sine check)  
Started Multi Adder Test.

1100  
1525

1545  Relay #70 Panel F  
(moth) in relay.

First actual case of bug being found.  
1630 arctangent started.  
1700 closed down.

- Debugging is the process of finding and resolving bugs (defects or problems that prevent correct operation) within computer programs, software, or systems.

A computer log entry from the Mark2, with a moth taped to the page

# Debugging techniques



Image from:

[https://www.reddit.com/r/ProgrammerHumor/  
comments/ntr076/all\\_the\\_print\\_statements/](https://www.reddit.com/r/ProgrammerHumor/comments/ntr076/all_the_print_statements/)

- Using std::cout
- Using a debugger
- Testing, logging
- Rubber duck debugging

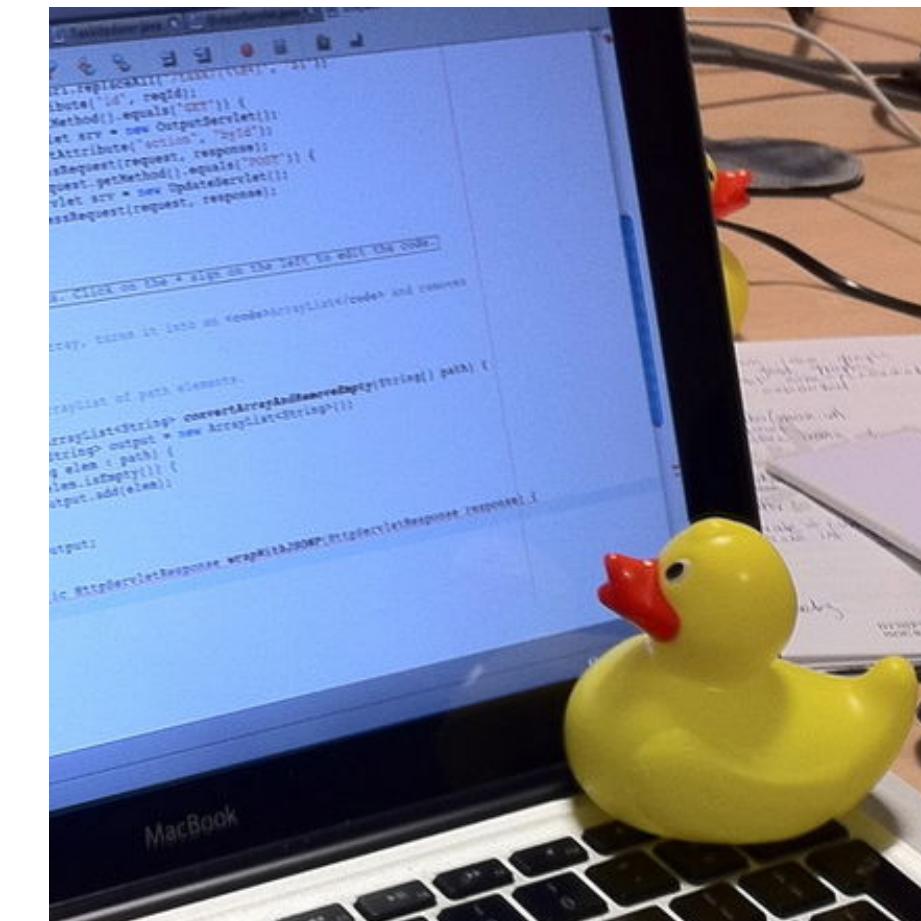


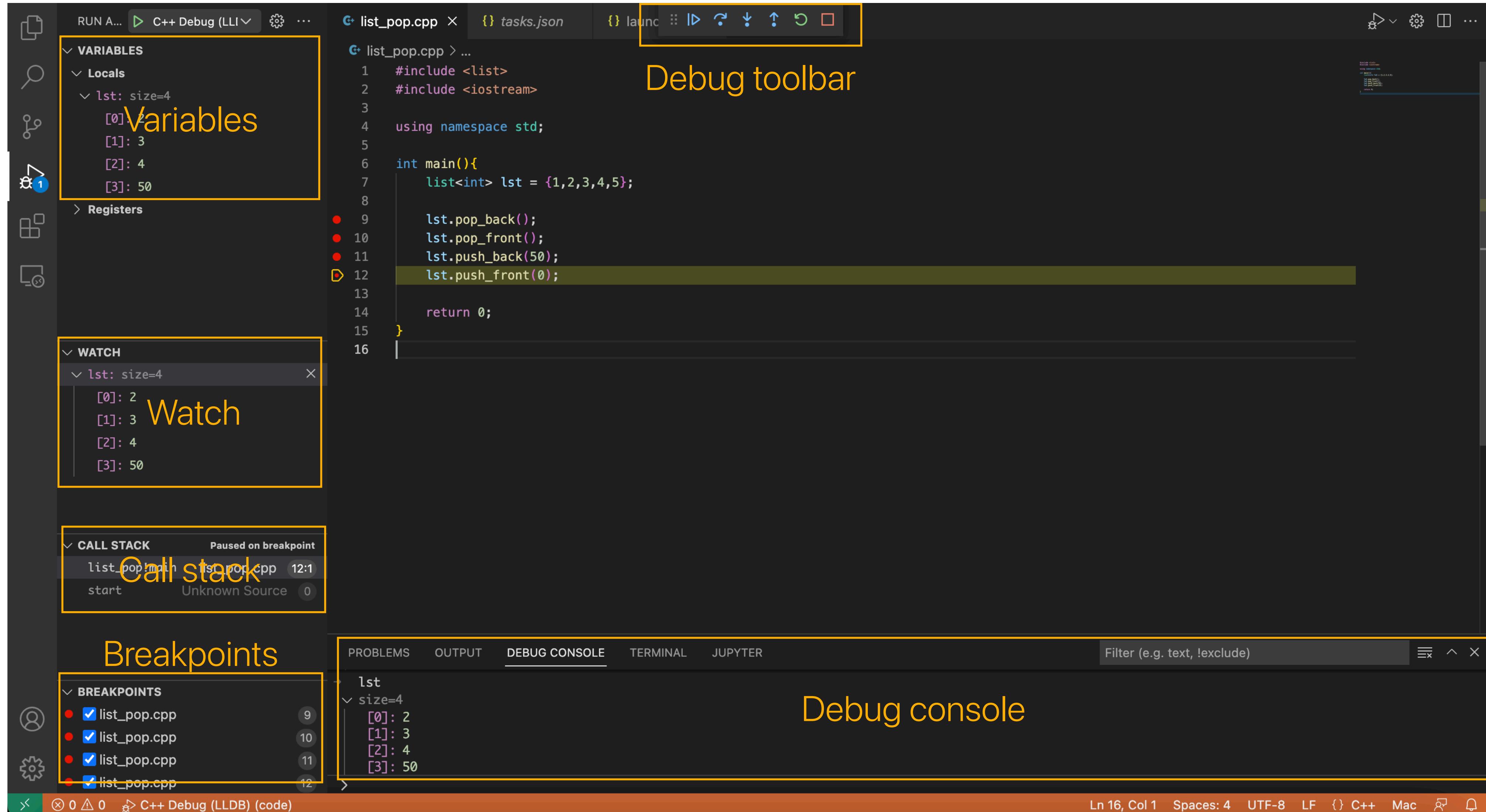
Image from:

[https://en.wikipedia.org/wiki/  
Rubber\\_duck\\_debugging](https://en.wikipedia.org/wiki/Rubber_duck_debugging)

# Why should I use a debugger?

- Understanding unfamiliar code
- Memory exploration
- Efficiency and productivity
- <https://news.ycombinator.com/item?id=29923007>

# Overview of using a debugger in VSCode



# Using multiple breakpoints

The screenshot shows a VS Code interface with a dark theme, illustrating the use of multiple breakpoints in a C++ application named `list_pop.cpp`.

**Breakpoints:** A yellow box highlights the `Breakpoints` section in the bottom-left corner of the interface.

**Locals:** The `Locals` section in the Variables sidebar shows the state of the variable `lst`:

[0]	[1]	[2]	[3]
2	3	4	50

**Registers:** The Registers sidebar shows the current register values.

**WATCH:** The WATCH sidebar shows the same `lst` variable state as the Locals view.

**CALL STACK:** The CALL STACK sidebar indicates the application is "Paused on breakpoint" at the `list_pop!main` entry point.

**Breakpoints:** A yellow box highlights the `Breakpoints` section in the bottom-left corner of the interface.

**Breakpoints:** The Breakpoints sidebar lists four breakpoints set on line 12 of `list_pop.cpp`, all of which are currently active (indicated by checked checkboxes).

```
#include <list>
#include <iostream>
using namespace std;

int main(){
    list<int> lst = {1,2,3,4,5};

    lst.pop_back();
    lst.pop_front();
    lst.push_back(50);
    lst.push_front(0);

    return 0;
}
```

Bottom status bar: `Ln 16, Col 1 Spaces: 4 UTF-8 LF {} C++ Mac`

# Turning off breakpoints

Click breakpoint

```
int main() {  
    list<int> lst = {1, 2, 3, 4, 5};  
    lst.pop_back(); // Breakpoint  
    lst.pop_front();  
    lst.push_back(50);  
    lst.push_front(0);  
    return 0;  
}
```

WATCH

lst: size=5

- [0]: 1
- [1]: 2
- [2]: 3
- [3]: 4
- [4]: 5

CALL STACK

Paused on breakpoint

list\_pop!main list\_pop.cpp 9:1

start Unknown Source 0

BREAKPOINTS

- list\_pop.cpp 9 (checked)
- list\_pop.cpp 10 (checked)
- list\_pop.cpp 11 (unchecked)
- list\_pop.cpp 12 (checked)

PROBLEMS OUTPUT DEBUG C

```
Loaded '/usr/lib/system/libc.dylib'  
Loaded '/usr/lib/system/libSystem.dylib'  
Loaded '/usr/lib/libobjc.A.dylib'  
Loaded '/usr/lib/liboah.dylib'  
=thread-selected,id="1"  
Execute debugger commands u
```

# Jumping lines

The screenshot shows a debugger interface with the following details:

- Code Editor:** Displays the file `debugging_test.cpp` with the following code:

```
#include <list>
using namespace std;

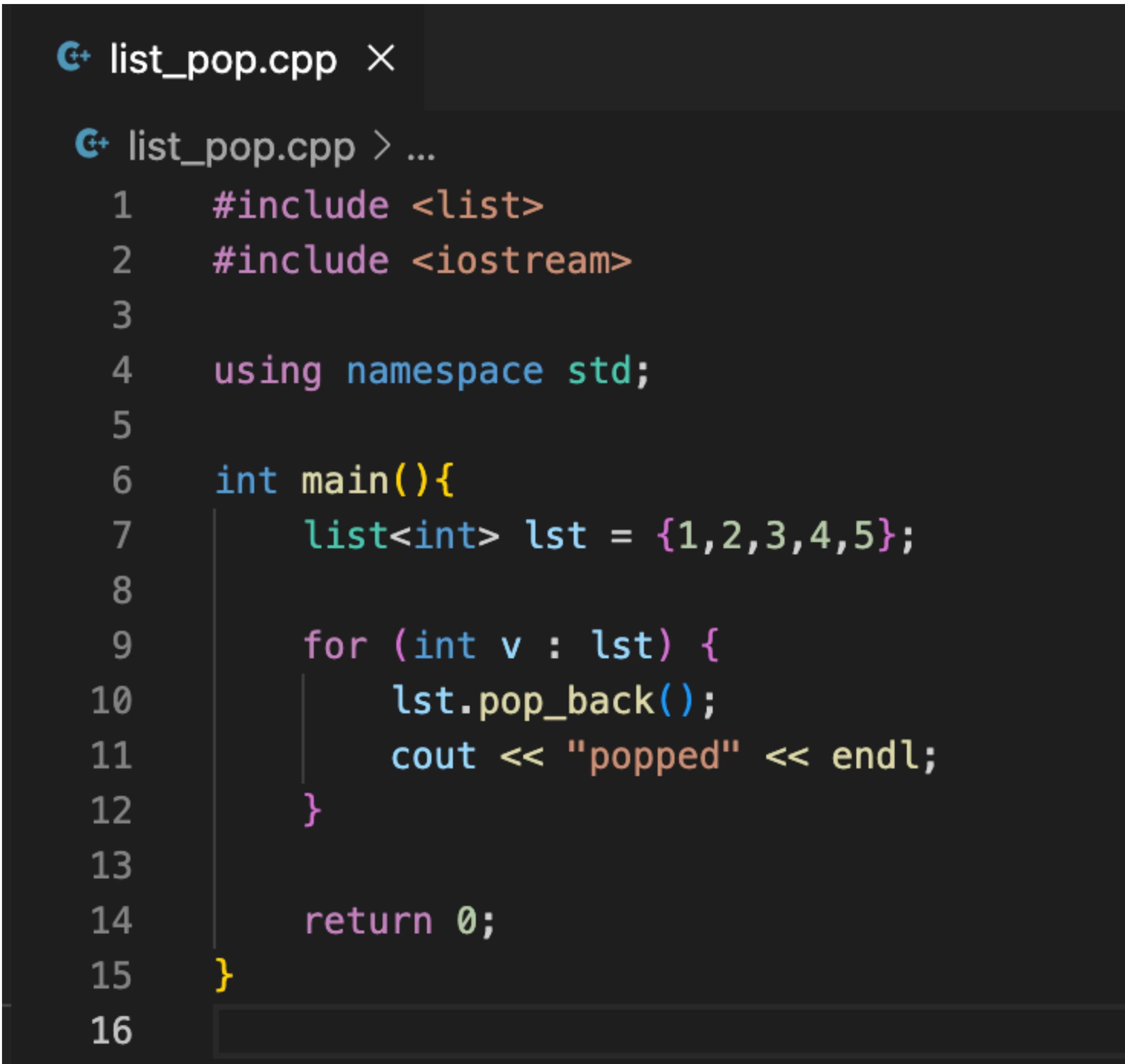
int main(){
    list<int> lst = {1,2,3,4,5};
    lst.pop_back();
    lst.pop_front();
    lst.push_back(50);
    lst.push_front(0);
    return 0;
}
```
- Variables View:** Shows the `lst` variable in the `Locals` section with elements [0]: 2, [1]: 3, [2]: 4, [3]: 50.
- Registers View:** Shows registers 9, 10, 11, and 12.
- WATCH View:** Shows a watch point on line 11.
- Call Stack:** Shows a single entry: `[1] PAUSED ON BREAKPOINT main() debugging_test...`.
- Breakpoints View:** Shows multiple breakpoints set across the file, with the one at line 11 being active (`-exec jump 5`).

The screenshot shows a debugger interface with the following details:

- Code Editor:** Displays the same code as the first screenshot.
- Variables View:** Shows the `lst` variable in the `Locals` section with elements [0]: 1, [1]: 2, [2]: 3, [3]: 4.
- Registers View:** Shows registers 9, 10, 11, and 12.
- WATCH View:** Shows a watch point on line 11.
- Call Stack:** Shows a single entry: `[1] PAUSED ON BREAKPOINT main() debugging_test...`.
- Breakpoints View:** Shows multiple breakpoints set across the file.
- Debug Console:** Shows the command `-exec jump 5` being typed.
- Output:** Shows the message "Continuing at 0x40155f."

- Type `-exec jump {line_number}` in the debug console
- If you use M1/M2(LLDB), type `thread jump --line {line_number}`

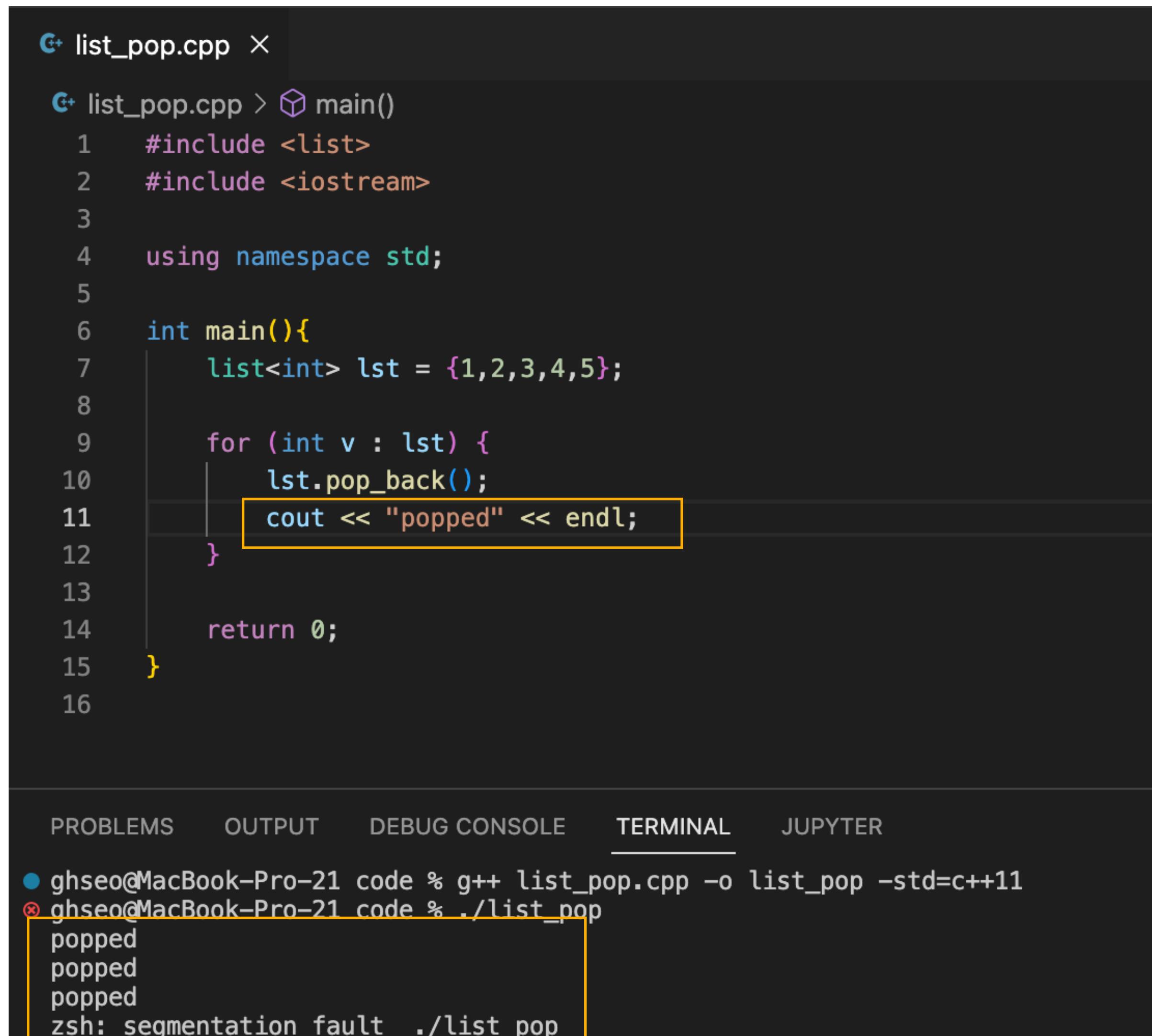
# Exercise 1: list\_pop.cpp



The image shows a code editor window with a dark theme. The file is named "list\_pop.cpp". The code itself is as follows:

```
1 #include <list>
2 #include <iostream>
3
4 using namespace std;
5
6 int main(){
7     list<int> lst = {1,2,3,4,5};
8
9     for (int v : lst) {
10         lst.pop_back();
11         cout << "popped" << endl;
12     }
13
14     return 0;
15 }
16
```

# Exercise 1: Using cout



```
C++ list_pop.cpp X

C++ list_pop.cpp > ⚙ main()
1 #include <list>
2 #include <iostream>
3
4 using namespace std;
5
6 int main(){
7     list<int> lst = {1,2,3,4,5};
8
9     for (int v : lst) {
10         lst.pop_back();
11         cout << "popped" << endl;
12     }
13
14     return 0;
15 }
16
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

- ghseo@MacBook-Pro-21 code % g++ list\_pop.cpp -o list\_pop -std=c++11
- ghseo@MacBook-Pro-21 code % ./list\_pop

```
popped
popped
popped
zsh: segmentation fault ./list_pop
```

# Exercise 1: Using debugger

The screenshot shows a debugger interface with the following details:

- Toolbar:** Includes icons for play, step, pause, and stop.
- File:** C++ list\_pop.cpp
- VARIABLES View:** Shows a local variable `lst` of size 2, containing values 1 and 2. A yellow box highlights the `Locals` section.
- Registers View:** Not explicitly shown but implied by the context.
- WATCH View:** Shows the same variable `lst` with size 2, and its elements [0]: 1 and [1]: 2.
- CALL STACK View:** Paused on exception, showing the stack trace: list\_pop!main
- Code Editor:** Displays the `main()` function with the following code:

```
#include <list>
#include <iostream>
using namespace std;

int main(){
    list<int> lst = {1,2,3,4,5};
    for (int v : lst) {
        lst.pop_back();
        // cout << "popped" << endl;
    }
    return 0;
}
```

A red arrow points to the `lst.pop_back();` line, which is highlighted in green.
- Exception Message:** An orange bar at the bottom indicates an exception has occurred: **Exception has occurred. ×** EXC\_BAD\_ACCESS (code=1, address=0x10)

# Exercise 1: Using debugger

The screenshot shows a code editor window with a dark theme. The file is named `list_pop.cpp`. The code defines a `main` function that creates a `list<int>` with elements {1,2,3,4,5}, then iterates over it using a `for` loop to call `pop_back` on each element. The output is printed to `cout`. The code editor has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and JUPYTER. The TERMINAL tab is active, showing the command `g++ list_pop.cpp -o list_pop -std=c++11` and its output: "popped" repeated five times.

```
❷ list_pop.cpp ×

❷ list_pop.cpp > ⚭ main()
1 #include <list>
2 #include <iostream>
3
4 using namespace std;
5
6 int main(){
7     list<int> lst = {1,2,3,4,5};
8
9     int lst_size = lst.size();
10    for (int i=0; i<lst_size; i++) {
11        lst.pop_back();
12        cout << "popped" << endl;
13    }
14
15    return 0;
16 }
17
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    **TERMINAL**    JUPYTER

- ❸ ghseo@MacBook-Pro-21 code % g++ list\_pop.cpp -o list\_pop -std=c++11
- ❸ ghseo@MacBook-Pro-21 code % ./list\_pop

popped  
popped  
popped  
popped  
popped

# Exercise 2: operator\_overloading

```
#include <string>
#include <iostream>

using namespace std;

int operator+(const string &lhs, const int rhs){
    return 1;
}

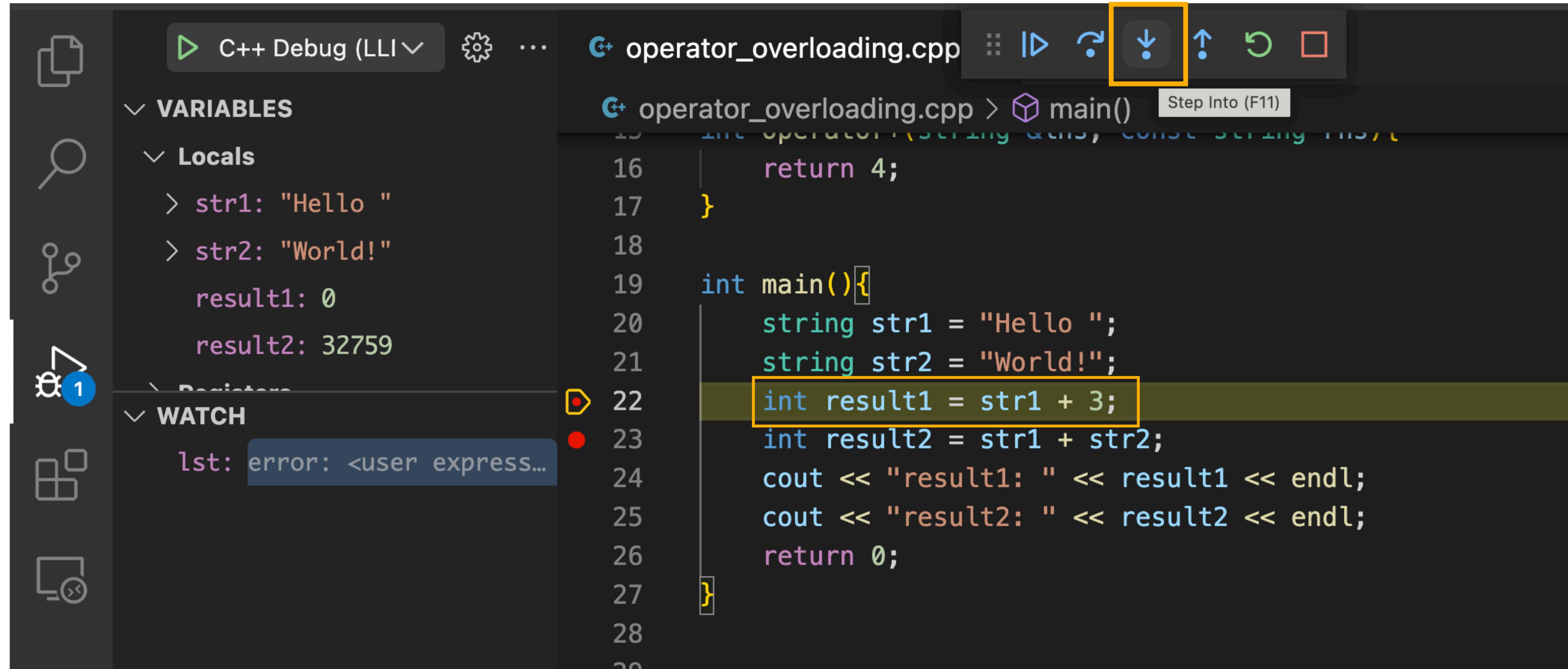
int operator+(string &lhs, const int rhs){
    return 2;
}

int operator+(const string &lhs, const string rhs){
    return 3;
}

int operator+(string &lhs, const string rhs){
    return 4;
}

int main(){
    string str1 = "Hello ";
    string str2 = "World!";
    int result1 = str1 + 3;
    int result2 = str1 + str2;
    cout << "result1: " << result1 << endl;
    cout << "result2: " << result2 << endl;
    return 0;
}
```

# Exercise 2: operator\_overloading



The screenshot shows a debugger interface with the following details:

- Toolbar:** Includes icons for file, C++ Debug (LLI), settings, and more.
- Code Editor:** Displays the file `operator_overloading.cpp` with the main function. A yellow box highlights the step-down button (`↓`) in the toolbar.
- Variables Panel:** Shows the `VARIABLES` section with `Locals` expanded, displaying:
  - `str1: "Hello "`
  - `str2: "World!"`
  - `result1: 0`
  - `result2: 32759`
- Registers Panel:** Shows a list of registers with a blue circle containing the number 1.
- Watch Panel:** Shows a list of watched expressions:
  - `lst: error: <user express...` (highlighted with a red circle)
  - `int result1 = str1 + 3;` (highlighted with a yellow box)
  - `int result2 = str1 + str2;`
  - `cout << "result1: " << result1 << endl;`
  - `cout << "result2: " << result2 << endl;`
  - `return 0;`

# Exercise 2: operator\_overloading

The screenshot shows a debugger interface with the following elements:

- VARIABLES:** A section showing local variables. It contains:
  - Locals:** lhs: "Hello " (highlighted with a yellow box, labeled 3)
  - rhs: 3
- WATCH:** A section showing a user expression 'lst' with an error message: 'error: <user express...>'.
- CALL STACK:** Paused on step 'operator\_overloading!operator'.
- Code Editor:** The code for 'operator\_overloading.cpp' is displayed, showing four operator+ overloads and the main function. The second overload (line 10) is highlighted with a yellow box and labeled 2. The line containing 'int result1 = str1 + 3;' is also highlighted with a yellow box and labeled 1.

```
C++ operator_overloading.cpp x
C++ operator_overloading.cpp > % operator+(string &, const int)
6   int operator+(const string &lhs, const int rhs){
7     return 1;
8   }
9   int operator+(string &lhs, const int rhs){ 2
10    return 2;
11  }
12  int operator+(const string &lhs, const string rhs){
13    return 3;
14  }
15  int operator+(string &lhs, const string rhs){
16    return 4;
17  }
18
19  int main(){
20    string str1 = "Hello ";
21    string str2 = "World!";
22    int result1 = str1 + 3; 1
23    int result2 = str1 + str2;
24    cout << "result1: " << result1 << endl;
25    cout << "result2: " << result2 << endl;
```

# Exercise 2: operator\_overloading

The screenshot shows a debugger interface with the following details:

- Toolbar:** C++ Debug (LLI), gear icon, three dots, file icon labeled "operator\_overloading.cpp", and various control icons.
- VARIABLES View:**
  - Locals:** str1: "Hello ", str2: "World!", result1: 2 (highlighted with a yellow box), result2: 32759.
  - Registers:** Not visible.
- WATCH View:** lst: error: <user express...>
- CALL STACK View:** Paused on breakpoint at operator\_overloading!main.
- Code Editor:** Shows the source code for "operator\_overloading.cpp".

```
7     return 1;
8 }
9 int operator+(string &lhs, const int rhs){
10    return 2;
11 }
12 int operator+(const string &lhs, const string rhs){
13    return 3;
14 }
15 int operator+(string &lhs, const string rhs){
16    return 4;
17 }

18
19 int main(){
20     string str1 = "Hello ";
21     string str2 = "World!";
22     int result1 = str1 + 3; // Breakpoint
23     int result2 = str1 + str2; // Step over
24     cout << "result1: " << result1 << endl;
25     cout << "result2: " << result2 << endl;
26     return 0;
}
```

A red dot marks the breakpoint at line 22, and a yellow diamond marks the step-over point at line 23.

# Exercise 2: operator\_overloading

The screenshot shows a debugger interface with the following details:

- Variables View:** Shows Locals with str1: "Hello ", str2: "World!", result1: 2, and result2: 4. The result2 entry is highlighted with a yellow border and the number 3.
- Watch View:** Shows lst: error: <user express...>.
- Call Stack View:** Shows Paused on step and C/C++ PAUSED ON STEP.
- Code Editor:** The file operator\_overloading.cpp contains the following code:

```
operator_overloading.cpp:main()
8     }
9     int operator+(string &lhs, const int rhs){
10    |     return 2;
11    }
12    int operator+(const string &lhs, const string rhs){
13    |     return 3;
14    }
15    int operator+(string &lhs, const string rhs){
16    |     return 4;
17    }

19    int main(){
20    |     string str1 = "Hello ";
21    |     string str2 = "World!";
22    |     int result1 = str1 + 3;
23    |     int result2 = str1 + str2; 1
24    |     cout << "result1: " << result1 << endl;
25    |     cout << "result2: " << result2 << endl;
26    |
27    }
```
- Breakpoints:** Breakpoints are set at lines 22 and 23. The line 23 breakpoint is highlighted with a yellow border and the number 1.
- Execution:** The program is paused on step 24, indicated by a yellow bar under the line of code.

# Exercise 3: simple\_vector\_bug.cpp

The screenshot shows a dark-themed instance of Visual Studio Code. The main editor window displays the code for `simple_vector_bug.cpp`. The terminal at the bottom shows the output of running the program, which includes a double free error message.

```
11
12     void resize();
13
14 public:
15     SimpleVector(int initialCapacity);
16     SimpleVector(std::initializer_list<T> elements);
17     ~SimpleVector();
18
19     void addElement(T element);
20     int getSize() const;
21
22     T& operator[](int index) const;
23 };
24
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    JUPYTER    zsh ! + ^

```
✖ ghseo@MacBook-Pro-21 code % ./simple_vector_bug
simple_vector_bug(3942,0x7ff8455737c0) malloc Double free of object 0x7fb15cf05e80
simple_vector_bug(3942,0x7ff8455737c0) malloc *** set a breakpoint in malloc_error_break to debug
zsh: abort      ./simple_vector_bug
```

# Exercise 3: simple\_vector\_bug.cpp

The screenshot shows a debugger interface with the following elements:

- Run and Debug**: C++ Debug (LLI)
- VARIABLES**:
  - vec1**: {...}
    - array: 0x00007f96a1c04080
    - size: 3
    - capacity: 3
  - vec2**: {...}
    - array: 0x00007f96a1c04080
    - size: 3
- WATCH**
- CALL STACK**: Paused on step
- simple\_vector\_bug!main** simple\_vector...
- start** Unknown Source 0

**Source Code (simple\_vector\_bug.cpp):**

```
67
68     template <typename T>
69     T& SimpleVector<T>::operator[](int index) const {
70         return array[index];
71     }
72
73     int main() {
74         SimpleVector<int> vec1 = {1, 2, 3};
75         SimpleVector<int> vec2 = vec1;
76
77         vec1.addElement(4);
78         vec2.addElement(5);
79
80         std::cout << "vec1: ";
81         for (int i = 0; i < vec1.getSize(); i++) {
```

# Exercise 3: simple\_vector\_bug.cpp

The screenshot shows a debugger interface with the following details:

- RUN AND DEBUG**: C++ Debug (LLDB)
- VARIABLES**:
  - Locals**: newArray: 0x00007fe097304100, this: 0x00007ff7bfeff3e0
  - Registers**
- WATCH**
- CALL STACK**: Paused on exception (SIGABRT)
  - libsystem\_kernel.dylib!\_\_pthread\_kill
  - libsystem\_pthread.dylib!pthread\_kill
  - libsystem\_c.dylib!abort
  - libsystem\_malloc.dylib!malloc\_vreport
  - libsystem\_malloc.dylib!malloc\_zone\_error
  - simple\_vector\_bug!SimpleVector<int>::resize() simple\_vector\_bug.cpp 64:1
  - simple\_vector\_bug!SimpleVector<int>::addElement(int) simple\_vector\_bug.cpp
  - simple\_vector\_bug!main simple\_vector\_bug.cpp 78:1
  - start Unknown Source 0
- Source View**: simple\_vector\_bug.cpp : resize()

```
template <typename T>
void SimpleVector<T>::resize() {
    capacity = capacity * 2;
    T* newArray = new T[capacity];
    for (int i = 0; i < size; i++)
        newArray[i] = array[i];
    delete[] array;
```

# Exercise 3: simple\_vector\_bug.cpp

The screenshot shows the Xcode debugger interface with the following details:

- RUN AND DEBUG**: Shows the current configuration is "C++ Debug (LLDB)".
- VARIABLES**:
  - Locals**: Shows local variables `vec1` and `vec2`.
  - Registers**: Shows CPU registers.
- WATCH**: Shows no active watch points.
- CALL STACK**: Shows the call stack with the following frames:
  - Unknown Source (Paused on exception)
  - Unknown Source
  - simple\_vector\_bug!SimpleVector<int>::resize() at simple\_vector\_bug.cpp:64:1
  - simple\_vector\_bug!SimpleVector<int>::addElement(int) at simple\_vector\_bug.cpp
  - simple\_vector\_bug!main at simple\_vector\_bug.cpp:78:1
  - start at Unknown Source 0
- Code View**: The code for `main()` is shown, with line 78 highlighted by a yellow box:

```
70     return array[index];
71 }
72
73 int main() {
74     SimpleVector<int> vec1 = {1, 2, 3};
75     SimpleVector<int> vec2 = vec1;
76
77     vec1.addElement(4);
78     vec2.addElement(5); // Line 78 highlighted
79
80     std::cout << "vec1: ";
81     for (int i = 0; i < vec1.getSize(); i++) {
82         std::cout << vec1[i] << " ";
83     }
84     std::cout << std::endl;
85
86     std::cout << "vec2: ";
87     for (int i = 0; i < vec2.getSize(); i++) {
88         std::cout << vec2[i] << " ";
89     }
```

# Exercise 3: simple\_vector\_bug.cpp

The screenshot shows a debugger interface with the following details:

- RUN AND DEBUG**: C++ Debug (LLDB)
- VARIABLES**:
  - Locals**:
    - this: 0x00007ff7bfeff3e0
    - element: 5
  - Registers**: CPU, Other Registers
- WATCH**: No items listed.
- CALL STACK**:
  - Paused on exception
  - Unknown Source (0)
  - simple\_vector\_bug!SimpleVector<int>::resize() simple\_vector\_bug.cpp 64:1
  - simple\_vector\_bug!SimpleVector<int>::addElement(int)** simple\_vector\_bug.c... (selected)
  - simple\_vector\_bug!main simple\_vector\_bug.cpp 78:1
  - start Unknown Source (0)

**Source Code (simple\_vector\_bug.cpp):**

```
C++ simple_vector_bug.cpp > addElement(T)
38
39
40 template <typename T>
41 SimpleVector<T>::~SimpleVector() {
42     delete[] array;
43 }
44
45 template <typename T>
46 void SimpleVector<T>::addElement(T element) {
47     if (size == capacity)
48         resize();
49     array[size] = element;
50     size++;
51 }
52
53 template <typename T>
54 int SimpleVector<T>::getSize() const {
55     return size;
56 }
57
58 template <typename T>
```

# Exercise 3: simple\_vector\_bug.cpp

The screenshot shows a debugger interface with the following details:

- Run and Debug:** C++ Debug (LLDB)
- VARIABLES:**
  - Locals:** newArray: 0x00007fe097304100, this: 0x00007ff7bfeff3e0
  - Registers:** CPU, Other Registers
- WATCH:** None
- CALL STACK:** Paused on exception. The stack trace shows frames from libsystem\_kernel.dylib, libsystem\_pthread.dylib, libsystem\_c.dylib, libsystem\_malloc.dylib, and simple\_vector\_bug!SimpleVector<int>::resize(). The current frame is simple\_vector\_bug!SimpleVector<int>::resize() at simple\_vector\_bug.cpp:64:1.
- Source Code:** simple\_vector\_bug.cpp

```
C++ simple_vector_bug.cpp > ...
54     int SimpleVector<T>::getSize() const {
55         return size;
56     }
57
58     template <typename T>
59     void SimpleVector<T>::resize() {
60         capacity = capacity * 2;
61         T* newArray = new T[capacity];
62         for (int i = 0; i < size; i++)
63             newArray[i] = array[i];
64         delete[] array;
65         array = newArray;
66     }
67
68     template <typename T>
69     T& SimpleVector<T>::operator[](int index) const {
70         return array[index];
71     }
72
73     int main() {
74         SimpleVector<int> vec1 = {1, 2, 3};
```

# Exercise 3: simple\_vector\_bug.cpp

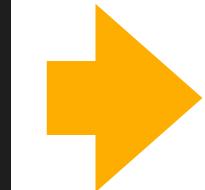
The screenshot shows a debugger interface with the following details:

- VARIABLES:** A list of variables in the current scope:
  - vec1:** {...}
    - array:** `0x00007f96a1c04080`
    - size:** 3
    - capacity:** 3
  - vec2:** {...}
    - array:** `0x00007f96a1c04080`
    - size:** 3
- WATCH:** A list of watched expressions.
- CALL STACK:** Shows the current stack frame: `simple_vector_bug!main`.
- Paused on step:** The program is paused at the second instruction of the main function.
- Code Editor:** The source code of `simple_vector_bug.cpp` is shown, with the main function highlighted. The bug is located in the `operator[]` method of the `SimpleVector` class.
- Call Stack:** The call stack shows the current state of the program, with the main function being executed.

```
simple_vector_bug.cpp:67: template <typename T>
simple_vector_bug.cpp:68:     T& SimpleVector<T>::operator[](int index) const {
simple_vector_bug.cpp:69:         return array[index];
simple_vector_bug.cpp:70:     }
simple_vector_bug.cpp:71: }
simple_vector_bug.cpp:72: int main() {
simple_vector_bug.cpp:73:     SimpleVector<int> vec1 = {1, 2, 3};
simple_vector_bug.cpp:74:     SimpleVector<int> vec2 = vec1;
simple_vector_bug.cpp:75:     vec1.addElement(4);
simple_vector_bug.cpp:76:     vec2.addElement(5);
simple_vector_bug.cpp:77:     std::cout << "vec1: ";
simple_vector_bug.cpp:78:     for (int i = 0; i < vec1.getSize(); i++) {
```

# Exercise 3: simple\_vector\_bug.cpp

```
73 int main() {  
74     SimpleVector<int> vec1 = {1, 2, 3};  
75     SimpleVector<int> vec2 = vec1;  
76  
77     vec1.addElement(4);  
78     vec2.addElement(5);  
79 }
```



```
73 int main() {  
74     SimpleVector<int> vec1 = {1, 2, 3};  
75     SimpleVector<int> vec2 = {1, 2, 3};  
76  
77     vec1.addElement(4);  
78     vec2.addElement(5);  
79 }
```

# Thank you

- <https://code.visualstudio.com/docs/cpp/cpp-debug>

# Operator overloading exercise

Doyoun Kim

# Overview

- A goal of this exercise is to implement class templates and apply operator overloading.
- In this exercise, you will...
  - Learn how to design a generic class that handles multiple data types.
  - Implement custom operators such as [], !=, \*, and ^.

# Overview

## Members of the Point Class:

- T x, y: Represent the two coordinates of the point.
- Supports multiple data types (int, float, double) using class templates.

## Key Methods:

- Constructor: Initializes x and y.
- Copy/Move constructors: Manages copying and moving of objects.
- Operator Overloads:
  - []: Access x and y using indices.
  - !=: Compares two points for inequality.
  - \*: Calculates the dot product of two points.
  - ^: Computes the L1 or L2 norm of the point.

# TODO 1&2: Implement copy/move constructor

- The copy constructor should initialize the new object's x and y with the corresponding values from another Point object.
- The move constructor should take the values from another Point object and reset the original object's values.
- Add debug output to the copy constructor, move constructor to verify whether the intended methods are being called during the tests.

Copy constructor called.  
Move constructor called.

# TODO 3&4: Implement copy/move assignment operator

- Copy assignment operator: The existing object should take on the same values as the other object.
- Move assignment operator: After the assignment, the original object should be reset to its default state.
- Add debug output to the copy and move assignment operator to verify whether the intended methods are being called during the tests.

Copy assingment operator called.  
Move assingment operator called.

# TODO 5: Implement the initializer\_list assignment operator

- Implement the assignment operator for the Point class using an initializer\_list.

```
Point<int> p1;  
p1 = {3, 4};
```

p1 should have x = 3 and y = 4.

# TODO 6: Implement the [] operator

- This operator should allow access to the x and y coordinates using an index 0 and 1. (0 for x, 1 for y).

`p[0]` should return the x coordinate,  
`p[1]` should return the y coordinate.

# TODO 7: Implement the \* operator

- Implement the \* operator to calculate the dot product of two Point objects.

```
Point<int> p1(1, 2), p2(3, 4);  
int result = p1 * p2;
```

The result should be 11.

# TODO 8: Implement the `^` operator

- Implement the `^` operator to calculate the L1 norm and L2 norm of a `Point` object.

```
Point<int> p1(3, 4);  
double result = p1^2;
```

The result should be 5.0.

# TODO 9: Implement the != operator

- Implement the != operator to check if two Point objects are not equal.
- The != operator should return true if the x or y coordinates of the two points are different. Otherwise, it should return false.

```
Point<int> p1(3, 4);  
Point<int> p2(3, 5);  
bool result = (p1 != p2);
```

true

# Run test cases

- Find the TODO sections in exercise.h files and implement them correctly based on the instructions.
- There are 9 TODOs.
- `$ g++ test_cases.cpp -o run_tests -std=c++17`
- `$ ./run_tests`