Debugging

Author: Peter Tse (mcreng)

Introduction

This tutorial mainly goes through the skills required for debugging in C++11. During software development, bugs are inevitable, and debugging skills can help trace the locations of errors. Main types of bugs include

- Compilation Errors,
- Linking Errors,
- Runtime Errors, and
- Logical Errors

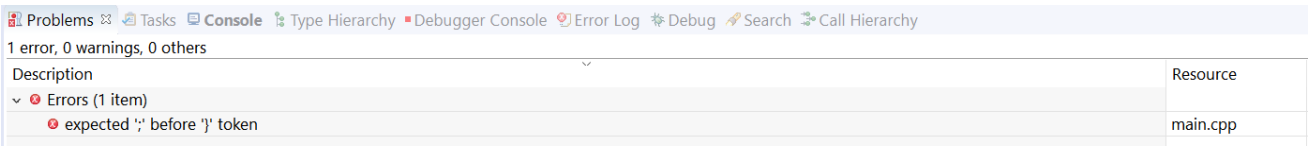
Compilation Errors

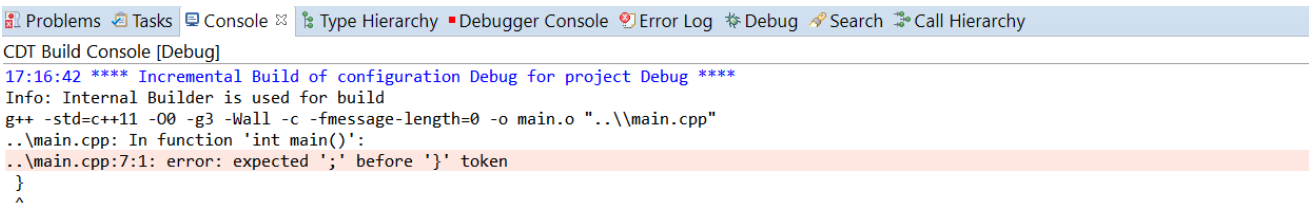
Compilation errors can be discovered when you compile your program. It is shown under both 'Problems' and 'Console' section in Eclipse.

A Simple Example

```cpp
#include <iostream>

using namespace std;

int main() {
  cout << "Hello World" << endl
}
```

This program cannot be compiled because of a missing colon.





Error messages are helpful but sometimes could be ambiguous. In that case, Google search the error would help.

To locate the error, you can either

- Go to 'Problems' tab, and double click on the error, or
- Go to 'Console' tab, and double click on the error (red line).

Both of the approaches allow you to go to the line and fix the error.

Exercises

In the following questions, locate the errors and the fix.

Question 1.

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

int main() {
  std::string name = 'Smart Car';
  std::cout << name << std::endl;
}
```

Question 2.

```
#include <iostream>

int main() {
  int a[] = {1, 2, 3, 4, 5};
  cout << a[4] << endl;
}

```

Question 3.

```
/* foo.h */
#ifndef FOO_H_
#define FOO_H_

class Foo {
public:
  int x;
  Foo(int x) : x(x){}
  Bar toBar() { return Bar(x); }
};

class Bar {
public:
  int x;
  Bar(int x) : x(x){}
  Foo toFoo() { return Foo(x); }
};

#endif /* FOO_H_ */
```

```cpp
/* main.cpp */
#include <iostream>
#include "foo.h"
using namespace std;

int main() {
  Foo foo(2);
  Bar bar = foo.toBar();
  cout << bar.x; // expected 2
}
```

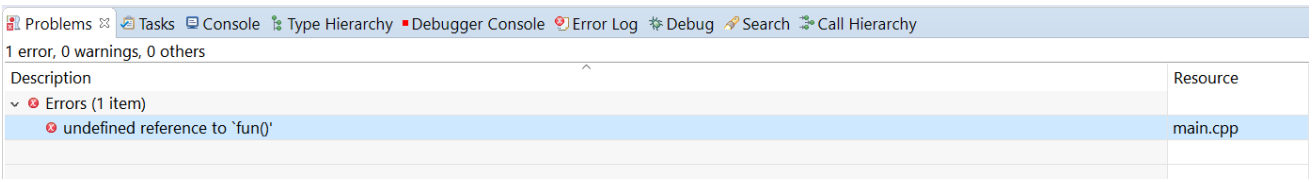
Linking Errors

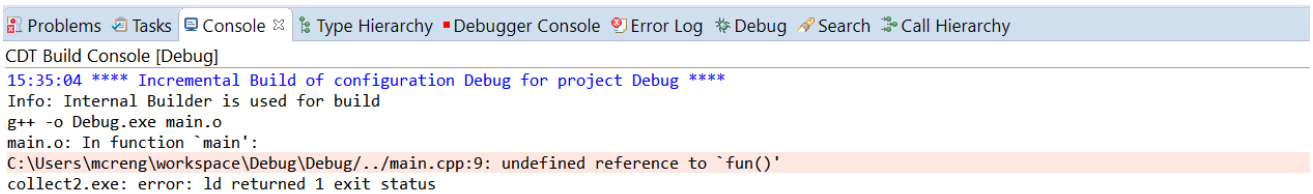
Linking errors are reflected by linker, which usually occur when working with multiple source files. Linking errors sometimes are not reflected under 'Problems' tab, so it's best to check the 'Console' tab if any error occurs.

Example

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

int fun();

int main() {
  cout << fun() << endl;
}
```

The linker could not find a definition to `fun()` so it returns an error.





The common error returned by linker is 'undefined reference', but it is not useful in telling you what is wrong.

Exercises

Question 1.

There are two linking errors in this question.

```cpp
/* foo.h */
```

```
#ifndef FOO_H_
#define FOO_H_
#include <iostream>
using namespace std;

class Foo {
public:
  Foo();
  void print() { cout << foo_number << endl; }
  static int foo_number;

};

#endif /* FOO_H_ */
```

```
/* main.cpp */
#include <iostream>
#include "foo.h"
using namespace std;

int main() {
  Foo foo;
  foo.print();
}
```

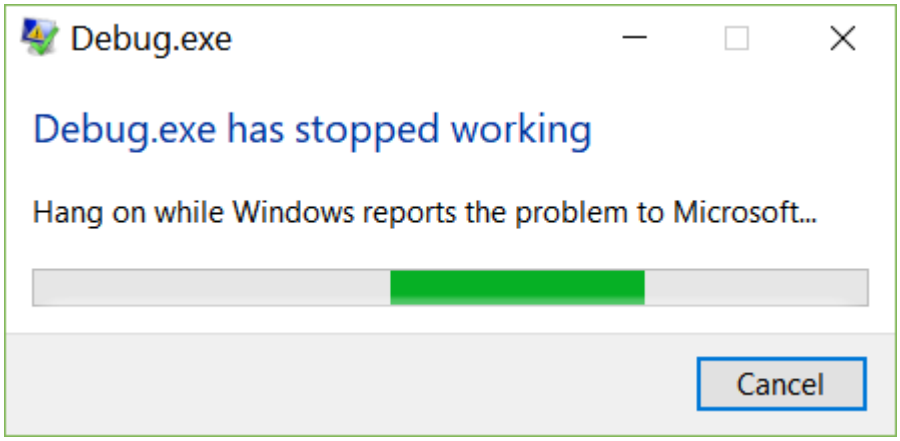
Runtime Errors

Runtime errors cannot be discovered upon compilation/linking. It can only be found when your program runs and crashes unexpectedly. To debug these errors, we will have to run the program in 'Debug' mode.

Example 1

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

int main() {
  vector<int> v;
  cout << v[0] << endl;
}
```

The program crashes unexpectedly.



Note that when programming K60/KEA, if the program crashes, the prompt is different. It could be

```
WDOG_EWM_IRQHandler:
00000524:   ldr     r0, [pc, #0]
184           bx r0
00000526:   bx      r0
183           ldr r0, =DefaultISR
```

breaking into watchdog(WDOG) interrupt(IRQ) handler, it could be

```
main.cpp   main.cpp   assert.c
 * assert.c

#include <assert.h>
#include <stdint.h>
#include <stdio.h>

#include "libbase/misc_utils_c.h"

void __assert_func(const char *file, int line, const char *fn,
        const char *expression)
{
    while (1)
    {
        iprintf("Assertion(%s) failed in %s:%s at line %d\n", expression, file,
                fn, line);
        // Arbitrary delay
        for (uint32_t i = 0; i < 50000000; ++i)
        {
            asm("nop");
        }
        __BREAKPOINT();
        __HARDFAULT();
    }
}
```

breaking into `assert.c`, or even

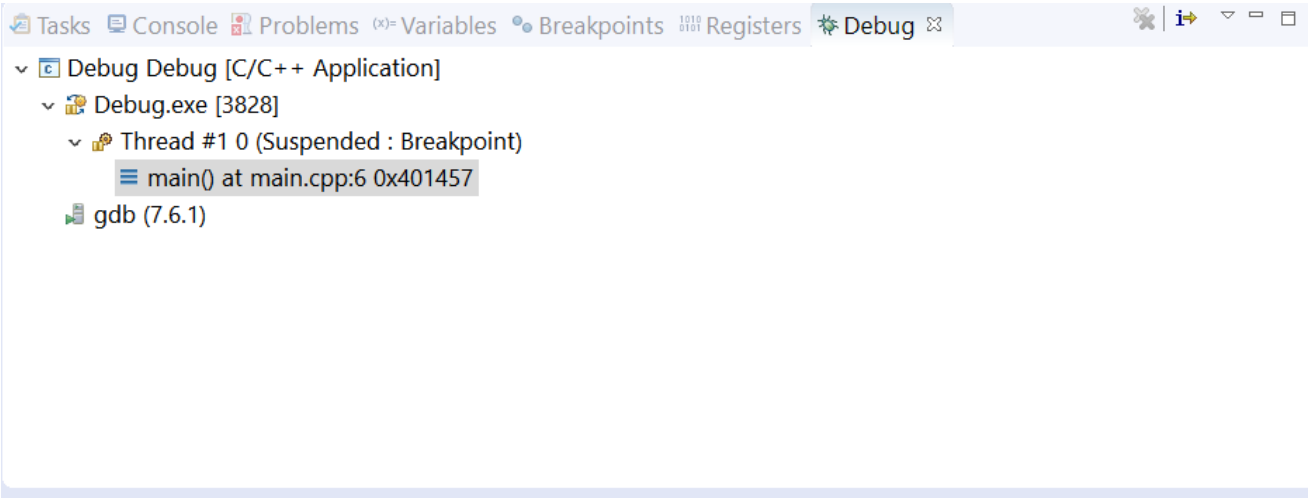
main.cpp main.cpp vectors.c

```c
    stacked_psr = ((unsigned long)hardfault_args[7]);

    // Configurable Fault Status Register
    // Consists of MMSR, BFSR and UFSR
    _CFSR = (*((volatile unsigned long*)(0xE000ED28)));

    // Hard Fault Status Register
    _HFSR = (*((volatile unsigned long*)(0xE000ED2C)));

    // Debug Fault Status Register
    _DFSR = (*((volatile unsigned long*)(0xE000ED30)));

    // Auxiliary Fault Status Register
    _AFSR = (*((volatile unsigned long*)(0xE000ED3C)));

    // Read the Fault Address Registers. These may not contain valid values.
    // Check BFARVALID/MMARVALID to see if they are valid values
    // MemManage Fault Address Register
    _MMAR = (*((volatile unsigned long*)(0xE000ED34)));
    // Bus Fault Address Register
    _BFAR = (*((volatile unsigned long*)(0xE000ED38)));

    if (g_hard_fault_handler)
    {
        g_hard_fault_handler();
    }
    __BREAKPOINT(); // Break into the debugger
}
```

breaking into `vectors.c`.

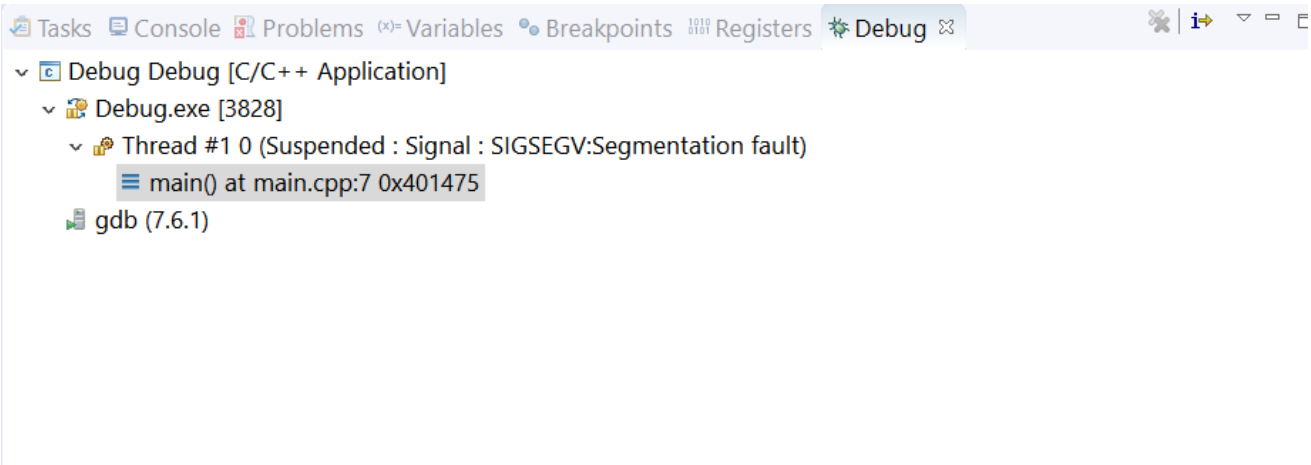
Anyway, if you encounter one of the above situations (or any other, such that you suspect it is runtime error), you should run your program in 'Debug' mode.

Inside 'Debug' mode, you should find a 'Debug' tab.



If you did not change any settings in Eclipse, when you run the program in 'Debug' mode, it sets a breakpoint at the first line of `main()`. You can see the current status of the program in `Thread #1 0 (Suspended : Breakpoint)`. The program is now being suspended by the initial breakpoint.

After clicking 'Continue', the program reaches the point where it crashes.



The debugger suspends the program before it crashes so you can see where the fault is. Now the status read `SIGSEGV:Segmentation fault` at `main.cpp:7`, which means at line 7 in `main.cpp`, it triggers a segmentation fault. Segmentation faults happen during illegal memory access or manipulation, and in our example we are accessing an illegal entry of the vector, which results in the crash.

In K60/KEA, the trace back in 'Debug' tab could be long.

An example of `HardFaultHandlerC()` at `vectors.c`:

workspace - Debug - /Users/Mr.King/git/libsccc/src/libbas

Debug

- Chasing17 Default [GDB SEGGER J-Link Debugging]
  - inno14-d-2017-chasing.elf
    - Thread #1 57005 (Suspended : Signal : SIGTRAP:Trace/breakpoint trap)
      - HardFaultHandlerC() at vectors.c:458 0x1e4aa
      - <signal handler called>() at 0xffffffe9
      - __vect_table() at 0x0
      - DebugConsole::DebugConsole() at 0x94b2
  - JLinkGDBServer
  - arm-none-eabi-gdb

main.cpp | Moving.h | main.cpp | main.h | main() at 0x661a | vectors.c

```c
    // Check BFARVALID/MMARVALID to see if they are valid values
    // MemManage Fault Address Register
    _MMAR = (*((volatile unsigned long*)(0xE000ED34)));
    // Bus Fault Address Register
    _BFAR = (*((volatile unsigned long*)(0xE000ED38)));

    if (g_hard_fault_handler)
    {
        g_hard_fault_handler();
    }
    __BREAKPOINT(); // Break into the debugger
}

#pragma GCC diagnostic pop
```

Console | Tasks | Properties | Problems | Executables | Memory

Chasing17 Default [GDB SEGGER J-Link Debugging] JLinkGDBServer

```
Read 2 bytes @ address 0x00009428 (Data = 0x60F8)
Read 2 bytes @ address 0x0000942A (Data = 0x60B9)
Read 2 bytes @ address 0x0000942C (Data = 0x607A)
Read 2 bytes @ address 0x0000942E (Data = 0x603B)
Read 2 bytes @ address 0x00009430 (Data = 0x68FB)
Read 2 bytes @ address 0x00009432 (Data = 0x2200)
Read 2 bytes @ address 0x00009434 (Data = 0x601A)
```

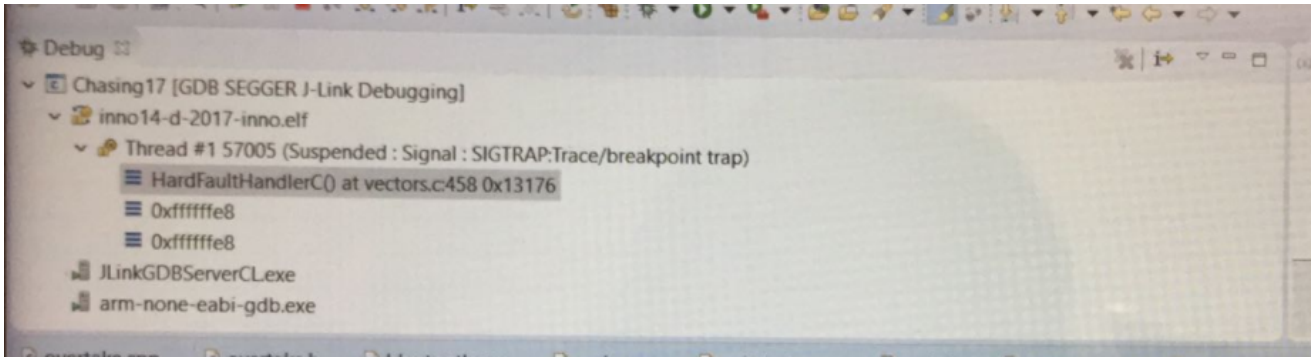
An example of `__assert_func()` at `assert.c`:

Debug

- Chasing17 Default [GDB SEGGER J-Link Debugging]
  - inno14-d-2017-chasing.elf
    - Thread #1 57005 (Suspended : Signal : SIGTRAP:Trace/breakpoint trap)
      - __assert_func() at assert.c:27 0x389c
      - _sbrk() at syscall.c:85 0xa7f8
      - _sbrk_r() at 0x1b61c
      - _malloc_r() at 0x1abc4
      - operator new() at 0x19ec0
      - void std::vector<std::pair<unsigned short, unsigned short>, std::allocator<std::pair<unsigned short, unsigned short> > >::emplace_back<std::pair<unsigned short, unsigned short> >() at 0x74c
      - algorithm::optimal::car1::(anonymous namespace)::FindOneLeftEdge() [clone .lto_priv.41] at 0xcaa
      - main() at 0x32ce
  - JLinkGDBServer
  - arm-none-eabi-gdb

The trace back usually includes details of STL and libsccc, and since you can safely assume they do not contain bugs, you should go from top to bottom to locate the functions that are written by you.

If you get the following trace back, good luck...



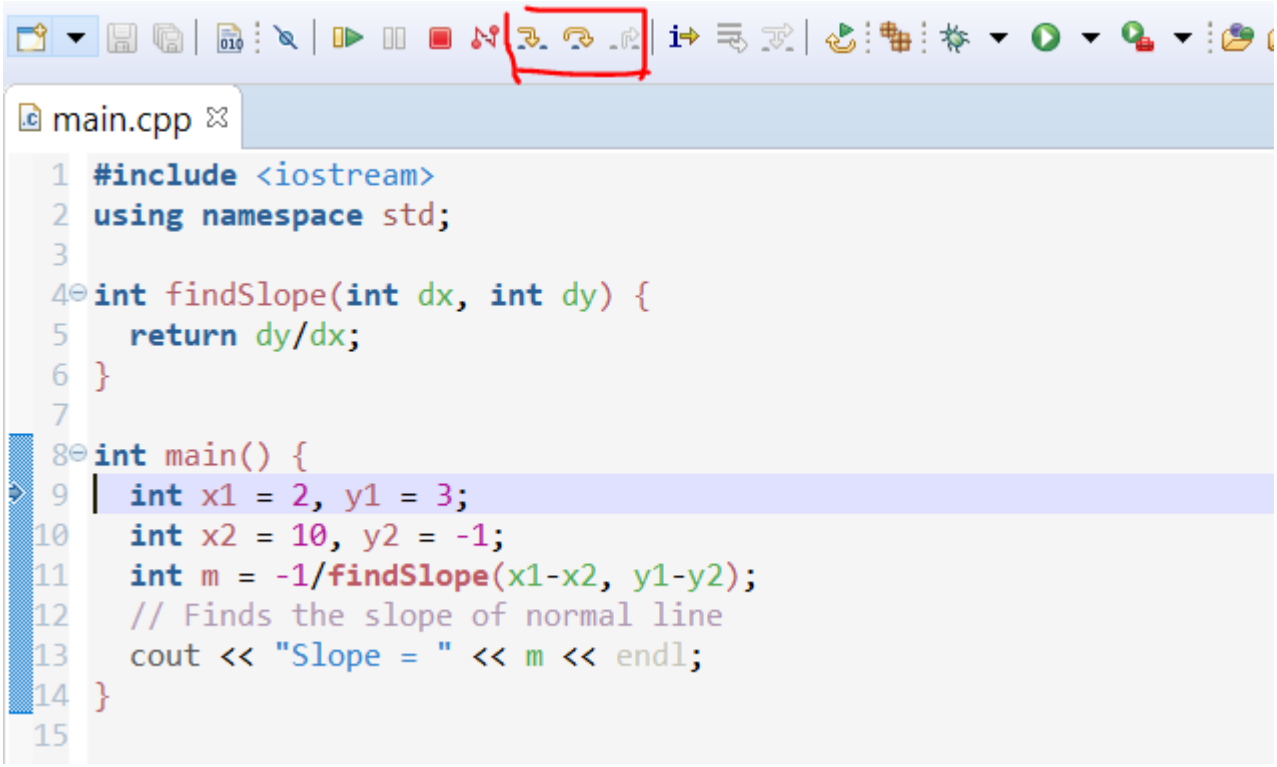
Without any details you need to run the code line by line and locate the source of error.

Example 2

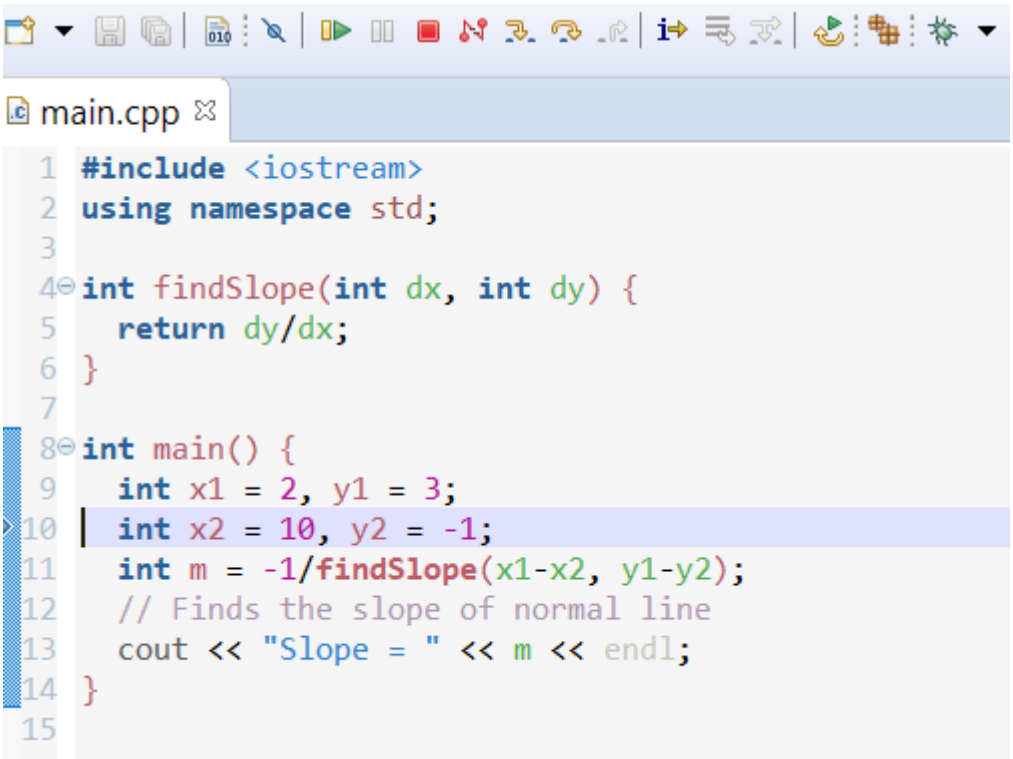
Consider the following code which calculates the slope of the line normal to the line passing through `(x1, y1)` and `(x2, y2)`.

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

int findSlope(int dx, int dy) {
  return dy/dx;
}

int main() {
  int x1 = 2, y1 = 3;
  int x2 = 10, y2 = -1;
  int m = -1/findSlope(x1-x2, y1-y2);
  // Finds the slope of normal line
  cout << "Slope = " << m << endl;
}
```

The program crashes and **assume** that you cannot look at the trace back at 'Debug', we will find out the problem by going through the code line by line.



Instead of clicking 'continue', we use the three buttons (inside the red box) instead. They are (from left to right), 'Step into', 'Step over' and 'Step return'. If you wish to go to the next line of execution, press 'Step over'.



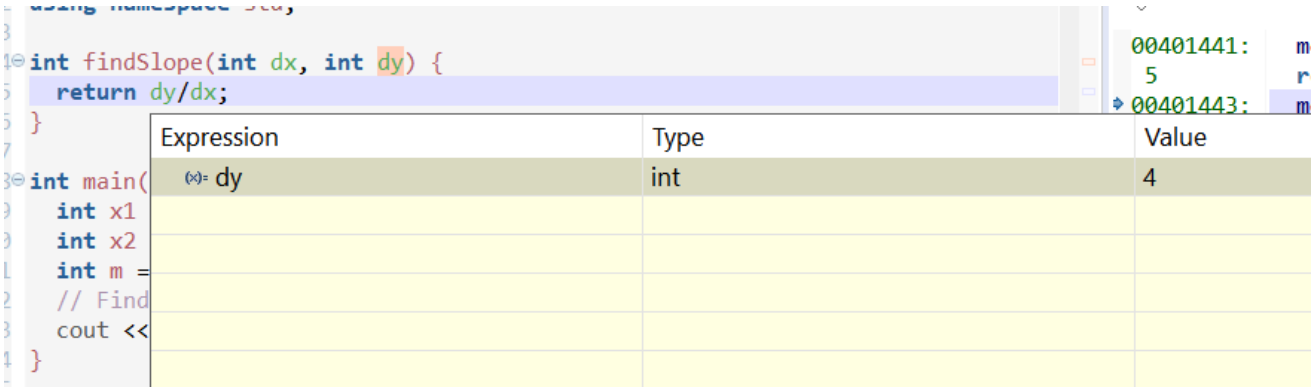
Nothing abnormal up until now, let us 'Step over' once more.

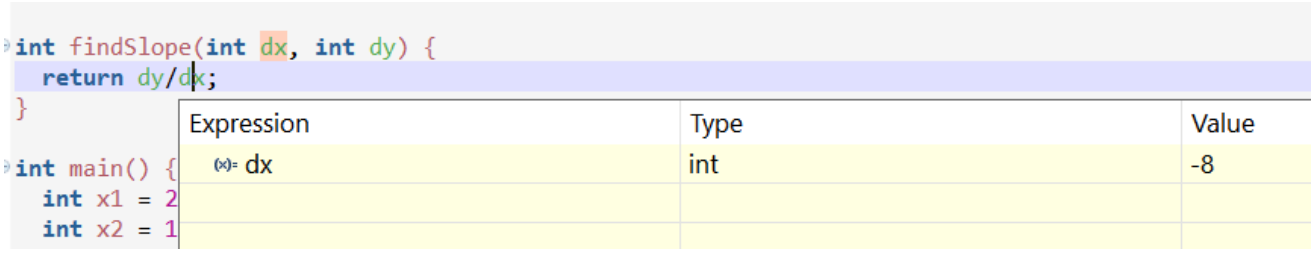
```
#include <iostream>
using namespace std;

int findSlope(int dx, int dy) {
  return dy/dx;
}

int main() {
  int x1 = 2, y1 = 3;
  int x2 = 10, y2 = -1;
  int m = -1/findSlope(x1-x2, y1-y2);
  // Finds the slope of normal line
  cout << "Slope = " << m << endl;
}
```

Here we reach a function `findSlope()`, we wish to see if there is anything wrong so we 'step into' the function.

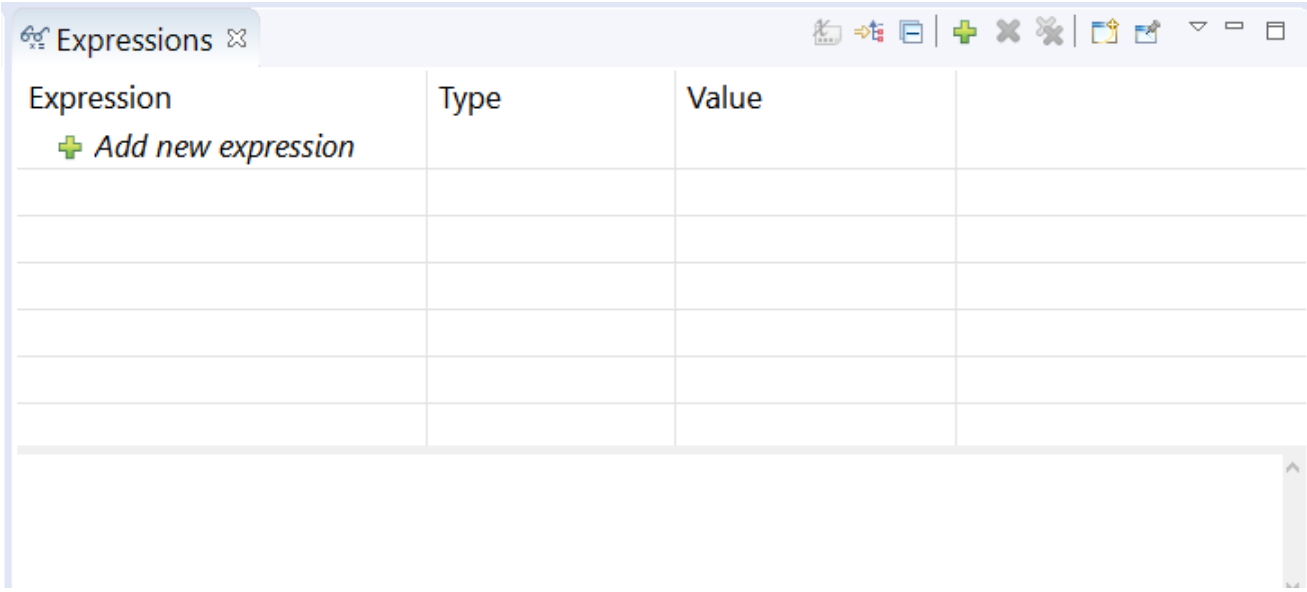
```
#include <iostream>
using namespace std;

int findSlope(int dx, int dy) {
  return dy/dx;
}

int main() {
  int x1 = 2, y1 = 3;
  int x2 = 10, y2 = -1;
  int m = -1/findSlope(x1-x2, y1-y2);
  // Finds the slope of normal line
  cout << "Slope = " << m << endl;
}
```

We then reach the line inside of the function. Now I wish to check the values of `dx` and `dy` and see if there would be a division by zero. You can hover on `dx` and `dy` and you can see the values.





Still nothing abnormal. Now let us use 'Expressions' to see if `dy/dx` gives our expected value of `-0.5`.



Here you may keep track of some variables/expressions during debugging. Note that only the variables visible in current scope is readable.

Expression	Type	Value
dx	int	-8
dy	int	4
findSlope(dx, dy)	int	0
dy/dx	int	0
x1		Error: Multiple err...
Add new expression		

Here we can see that `dy/dx` is not giving the same value as intended (because of int/int division). Now we know one logical bug but since the program has not crashed, we will go on. Also note the the variable `x1` is not visible here since the scope is not correct.

Now by clicking 'step return', we return to `main()`.

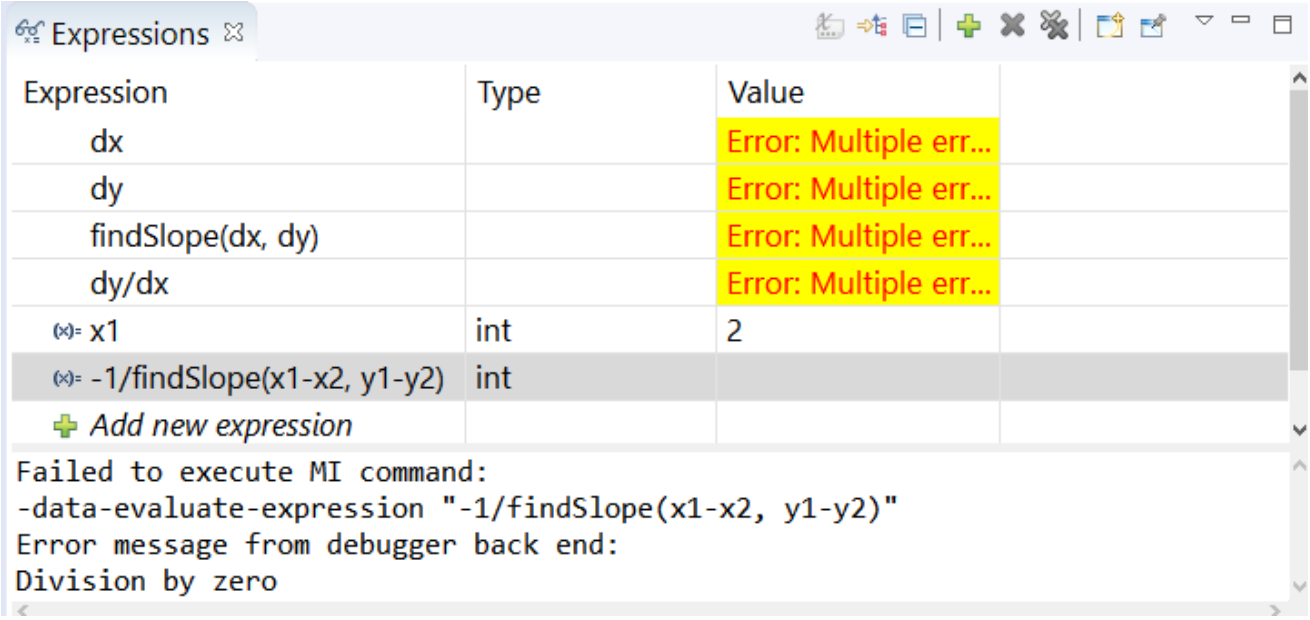
main.cpp

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

int findSlope(int dx, int dy) {
  return dy/dx;
}

int main() {
  int x1 = 2, y1 = 3;
  int x2 = 10, y2 = -1;
  int m = -1/findSlope(x1-x2, y1-y2);
  // Finds the slope of normal line
  cout << "Slope = " << m << endl;
}
```

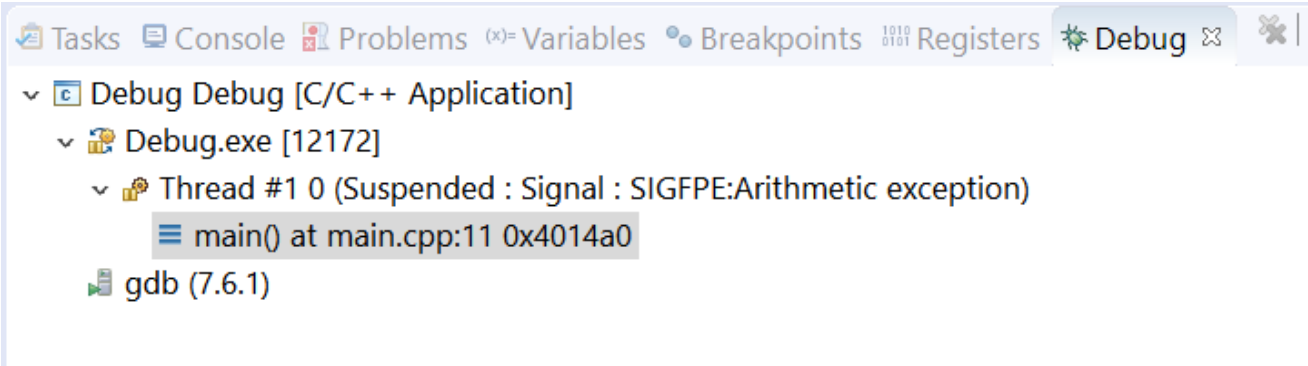
The expressions except `x1` are now not readable.

Expression	Type	Value
dx		Error: Multiple err...
dy		Error: Multiple err...
findSlope(dx, dy)		Error: Multiple err...
dy/dx		Error: Multiple err...
x1	int	2
Add new expression		

Now we know the program crashes because of division by zero. We can verify this by adding `-1/findSlope(x1-x2, y1-y2)` into 'Expressions' and see.



And expectedly, by clicking 'step over' once more, the program crashes.



Logical Errors

Logical errors are the hardest to detect since the problem lies within your algorithm. You might want to use the debugging tools mentioned above and see if the algorithm runs like what you expected. We could not tell you much because the algorithms are written by you!

Wrap-up Exercise

Consider the following algorithm of `Quick Select`.

```
// Returns the k-th smallest element of list within left..right inclusive
// (i.e. left <= k <= right).
// The search space within the array is changing for each round - but the list
// is still the same size. Thus, k does not need to be updated with each round.
function select(list, left, right, k)
    pivotIndex  := ...     // select a pivotIndex between left and right
    pivotIndex  := partition(list, left, right, pivotIndex)
    // The pivot is in its final sorted position
    if left = right        // If the list contains only one element,
        return list[left]  // return that element
    else if k = pivotIndex

        return list[k]
```

```
    else if k < pivotIndex
        return select(list, left, pivotIndex - 1, k)
    else
        return select(list, pivotIndex + 1, right, k)

function partition(list, left, right, pivotIndex)
    pivotValue := list[pivotIndex]
    swap list[pivotIndex] and list[right]  // Move pivot to end
    storeIndex := left
    for i from left to right-1
        if list[i] < pivotValue
            swap list[storeIndex] and list[i]
            increment storeIndex
    swap list[right] and list[storeIndex]  // Move pivot to its final place
    return storeIndex
```

The following is an implementation of `Quick Select` in C++.

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

template <class T>
void swap(T a, T b) {
  T temp = a;
  a = b;
  b = temp;
}

int select(int list, int left, int right, int k) {
  int pivotIndex = left + rand() % (right - left);
  pivotIndex = partition(list, left, right, pivotIndex);
  if (left == right) return list[left];
  else if (k == pivotIndex) return list[k];
  else if (k < pivotIndex) return select(list, left, pivotIndex-1, k);
  else return select(list, pivotIndex+1, right, k);
}

int partition(int list, int left, int right, int pivotIndex) {
  int pivotValue = list[pivotIndex];
  swap(list[pivotIndex], list[right]);
  int storeIndex = left;
  for (int i = left; i < right-1; i++)
    if (list[i] < pivotValue) swap(list[storeIndex++], list[i]);
  swap(list[right], list[storeIndex]);
  return storeIndex;
}

int main() {
  int a[] = {1, 5, 3, 2, 4, 10, 3, 2};
  //Expected output: 1, 2, 2, 3, 3, 4, 5, 10
  for (int i = 0; i < 8; i++) std::cout << select(a, 0, 7, i) << " ";
}
```

```

```

The function `select(a, 0, 7, i)` is supposed to get the `i`-th sorted element in the array `a`. However, in this code there are several bugs. Try to debug it with the skills you learned above, with the psuedocode as reference to the algorithm design. One of the bugs is more difficult and the hint for that specific bug is to consider the use of the library of `<ctime>`. There is also one optimization you can make that is **not** reflected on both psuedocode and C++ code. If all the bugs are fixed, running the code should give you the expected output as stated above.