南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

C/C++ Program Design

Lab 5, pointers and dynamic memory

王大兴，廖琪梅

Pointers and Dynamic Memory

- Pointers
- Dynamic Memory
- Debugging with gdb
- Debugging with vscode

Pointers

A pointer is a special type who holds the address of a value.

A pointer can point to a variable, a structure, an array and so on.

Use **reference operator(&)** to get the address of a variable and **dereference operator(*)** to get the value stored in the memory address.

Caution:

When you create a pointer, the computer allocates memory to hold an address, but it does not allocate memory to hold the data to which the address points.

Initialize a pointer to a definite and appropriate address before you apply the dereferencing operator (*) to it.

Pointers to array

When a pointer points to an array, it needs not use **&** to get the address of array, because the array name is a constant address. Dereference operator(*) combines with shifting the pointer to access the elements’ value rather than using subscript (index) .

Caution:

When shifting a pointer to an array, it can be out of bound without any warning, do not use such pointer to access(or modify) the content of that address.

Dynamic Memory

With C style, you can use **calloc** or **malloc** function to allocate memory for a pointer.

When you are not in need of memory any more, you should release that memory by calling the function **free().**

With C++ style, you can use **new** operator to allocate memory for a pointer and **delete** operator to release that memory.

Caution:

When you request memory by function or operator, check the pointer whether it is NULL.

When allocating a contiguous memory by **new []**, de-allocate by **delete []**. Be careful, memory leak.

Debugging with gdb

- Debugging is a common process in the software development process.
- Let us assume we don't know why the following code went wrong:

```
#include<iostream>
using std::cout;
using std::endl;

int plus( int a, int b ) {
        return a + b;
}

int main() {
        int a = 1234567890;
        int b = 1234567890;
        int c = plus(a, b);
        cout<<"a + b = "<<c<<endl;
return 0;
}
```

```
a + b = -1825831516
```

Debugging with gdb

- One common way of debugging: inserting “cout” everywhere.
- Simple to understand.
- After you find the bug, you need to remove all the “cout”, could take a lot of time.
- Using gdb could be a better choice.

```
int plus( int a, int b ) {
        cout<<"in plus "<<a<<endl;
        cout<<"in plus "<<b<<endl;
        return a + b;
}

int main() {
        int a = 1234567890;
        cout<<"in main "<<a<<endl;
        int b = 1234567890;
        cout<<"in main "<<b<<endl;
        int c = plus(a, b);
        cout<<"a + b = "<<c<<endl;
return 0;
}
```

Debugging with gdb

- First step of using gdb:
- Compile the program with **-g**, tell the compiler to include the symbol table. You can not debug the program without **-g**.

- For example, run the following command to compile main.cpp:

```
g++ -g -o main main.cpp
```

Debugging with gdb

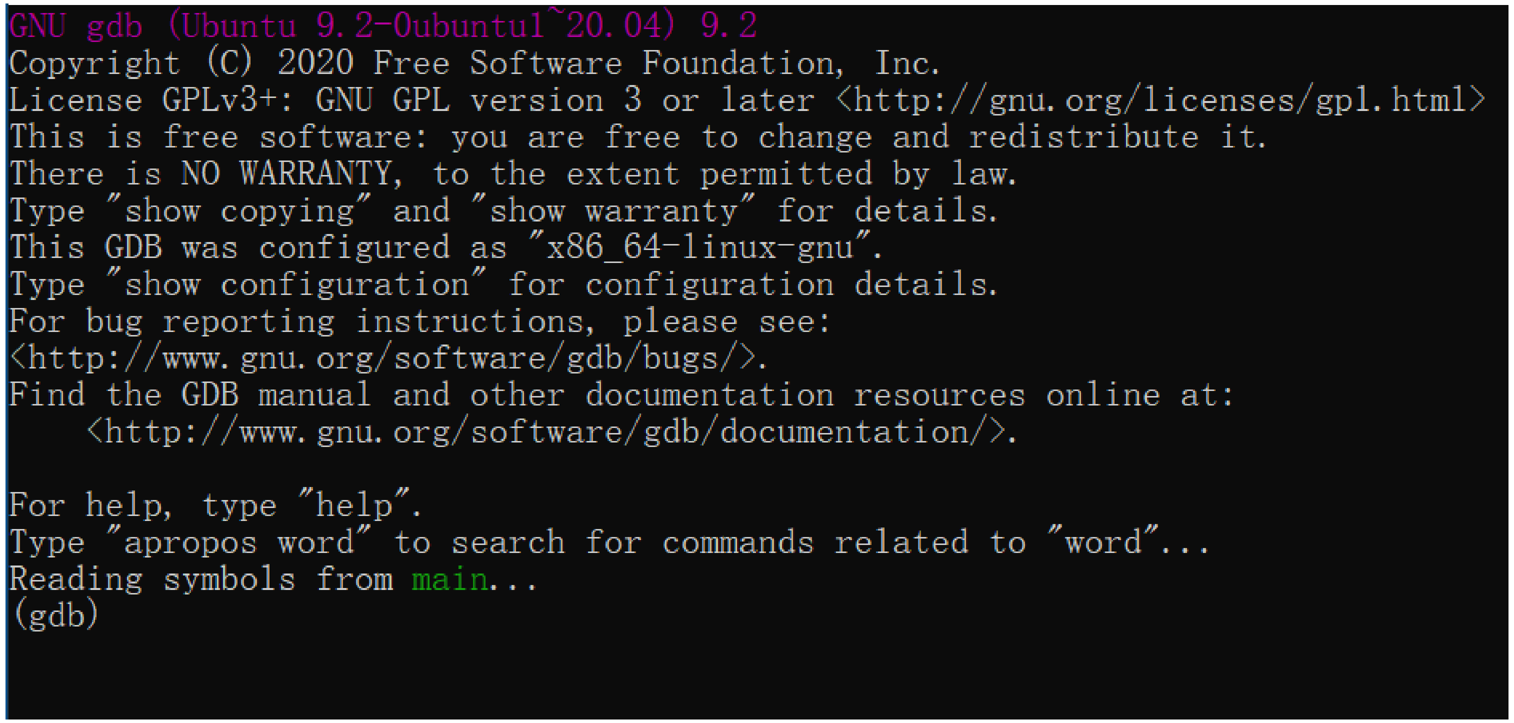
- Next, running the compiled program with gdb:

```
gdb program_name
```

- For example, if your executable name is “main”, then run:

```
gdb main
```

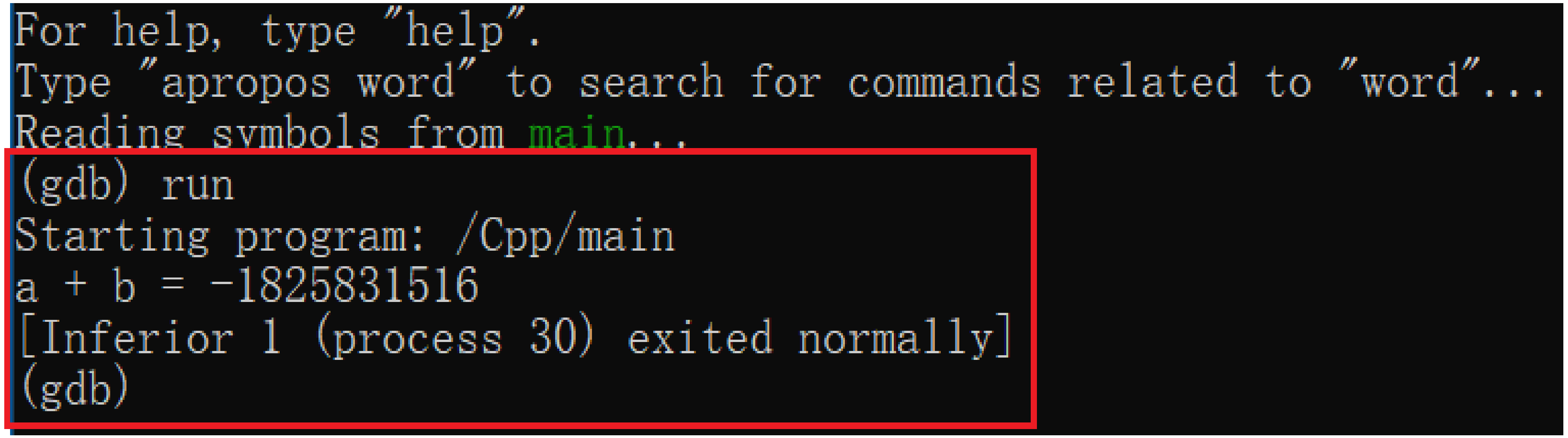
- You see something like this:



```
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main...
(gdb)
```

Debugging with gdb

- Now gdb is running, you can execute some command on the gdb.
- Using the “run” command to make gdb execute your program



```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main...
(gdb) run
Starting program: /Cpp/main
a + b = -1825831516
[Inferior 1 (process 30) exited normally]
(gdb)
```

Debugging with gdb

- One important feature of gdb is that it can execute to some point of the program, and see the current state of the program.

- Suppose we want to insert a “break point” before

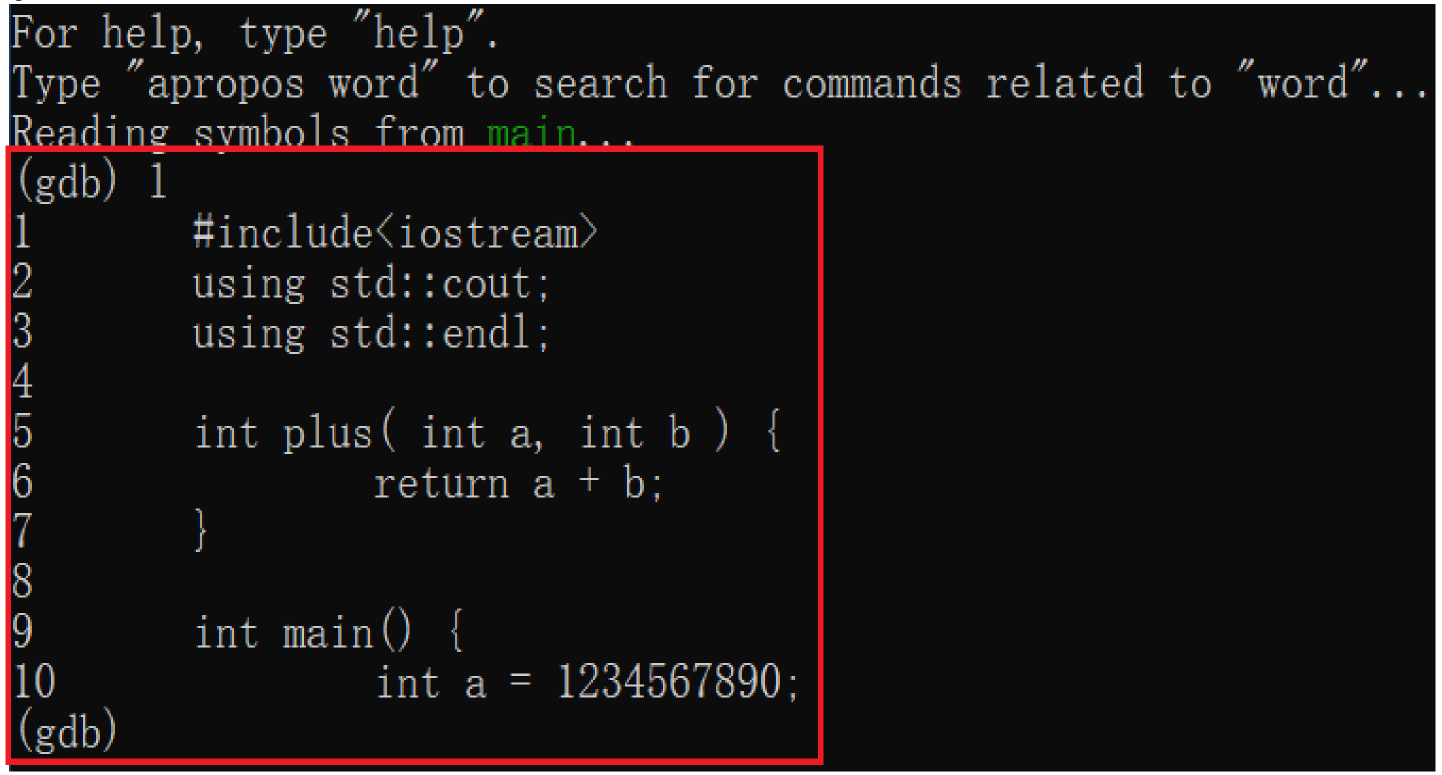
  int c = plus(a,b);

```
#include<iostream>
using std::cout;
using std::endl;

int plus( int a, int b ) {
        return a + b;
}

int main() {
        int a = 1234567890;
        int b = 1234567890;
        int c = plus(a, b);
        cout<<"a + b = "<<c<<endl;
return 0;
}
```

Insert break point before:
int c = plus(a,b);

Debugging with gdb

- Press “q” and enter to quit the execution, and rerun “gdb main” to execute gdb on the “main” executable.
- Press “l” and enter to list the line numbers.
- We need to insert a break point at the 12th line.



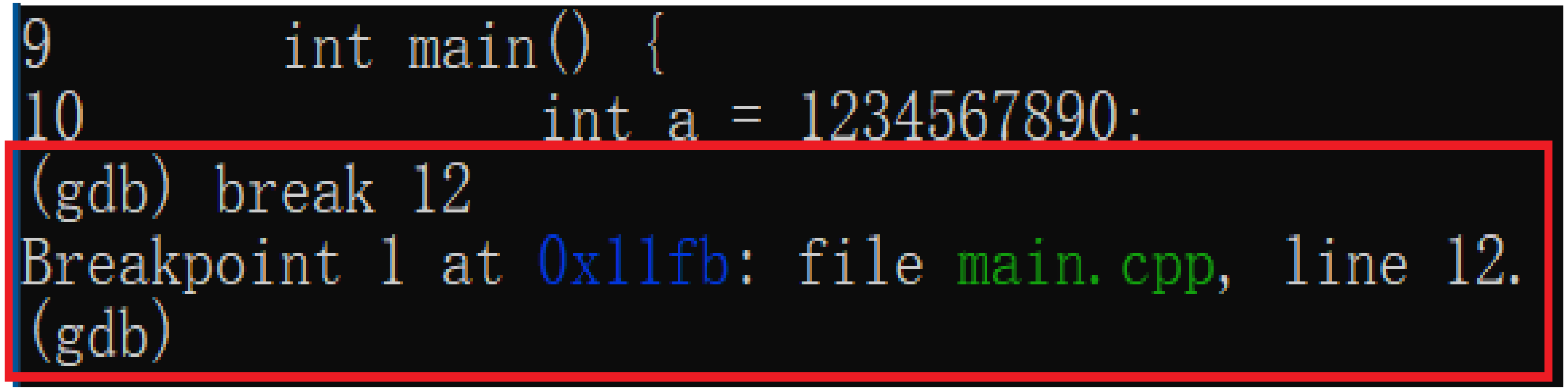
```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main...
(gdb) l
1       #include<iostream>
2       using std::cout;
3       using std::endl;
4
5       int plus( int a, int b ) {
6               return a + b;
7       }
8
9       int main() {
10              int a = 1234567890;
(gdb)
```

Debugging with gdb

- Set a break point at the 12th line with:

```
break 12
```

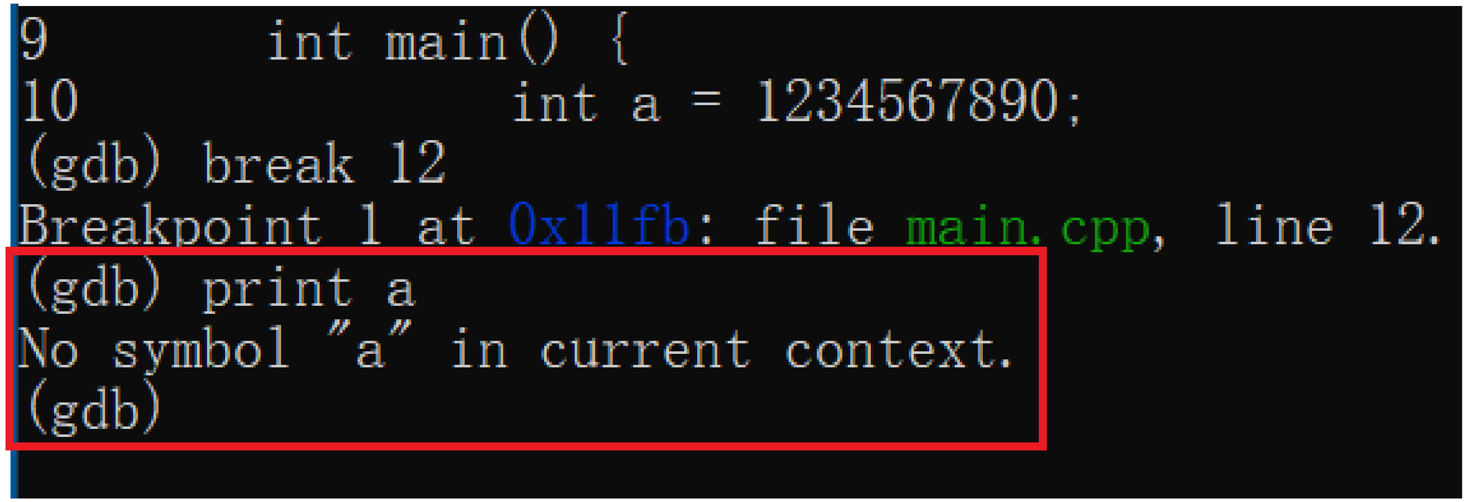
- See the output:



```
9          int main() {
10                 int a = 1234567890;
(gdb) break 12
Breakpoint 1 at 0x11fb: file main.cpp, line 12.
(gdb)
```

Debugging with gdb

- Print current state of variables, for example, print variable a:
- print a
- See the output:

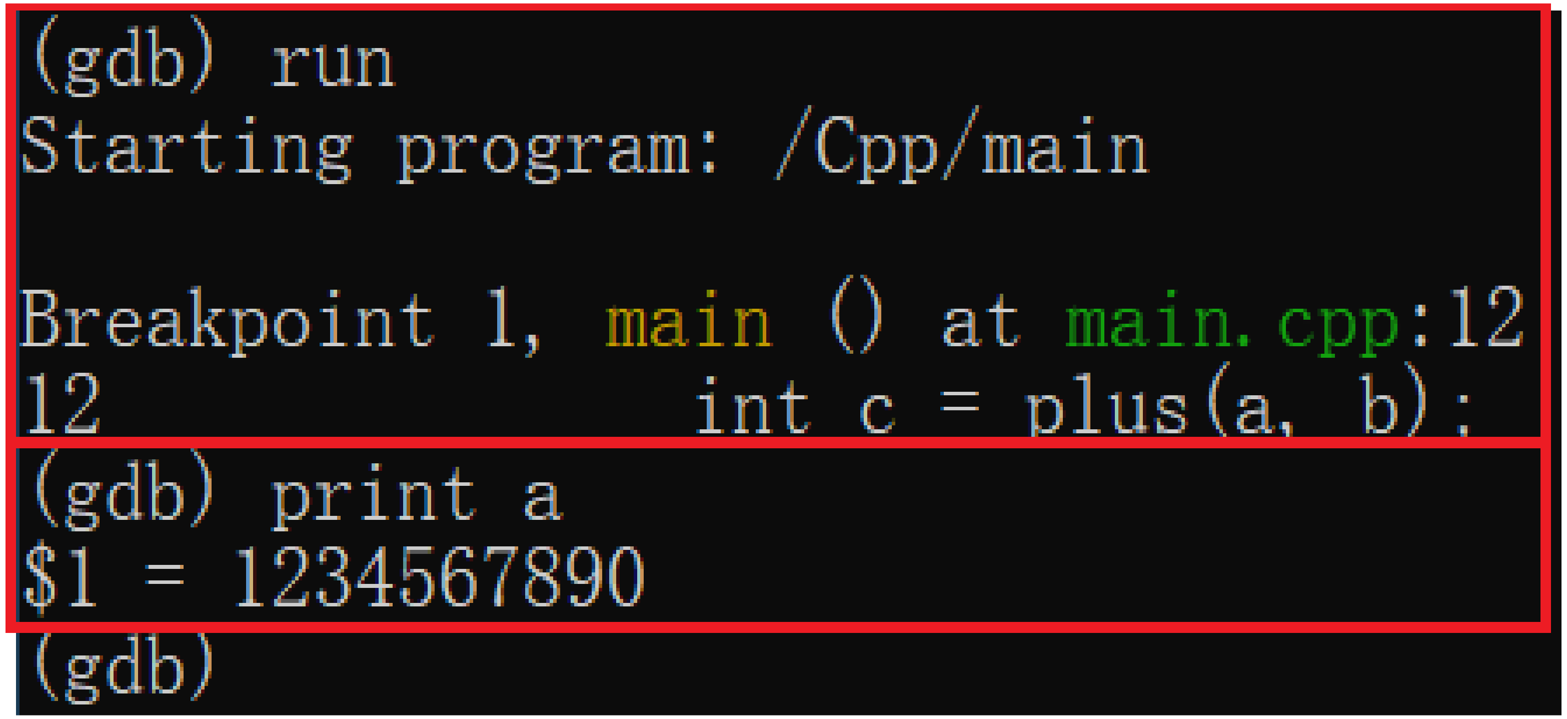


```
9           int main() {
10                   int a = 1234567890;
(gdb) break 12
Breakpoint 1 at 0x11fb: file main.cpp, line 12.
(gdb) print a
No symbol "a" in current context.
(gdb)
```

- Because we haven’t run anything yet.

Debugging with gdb

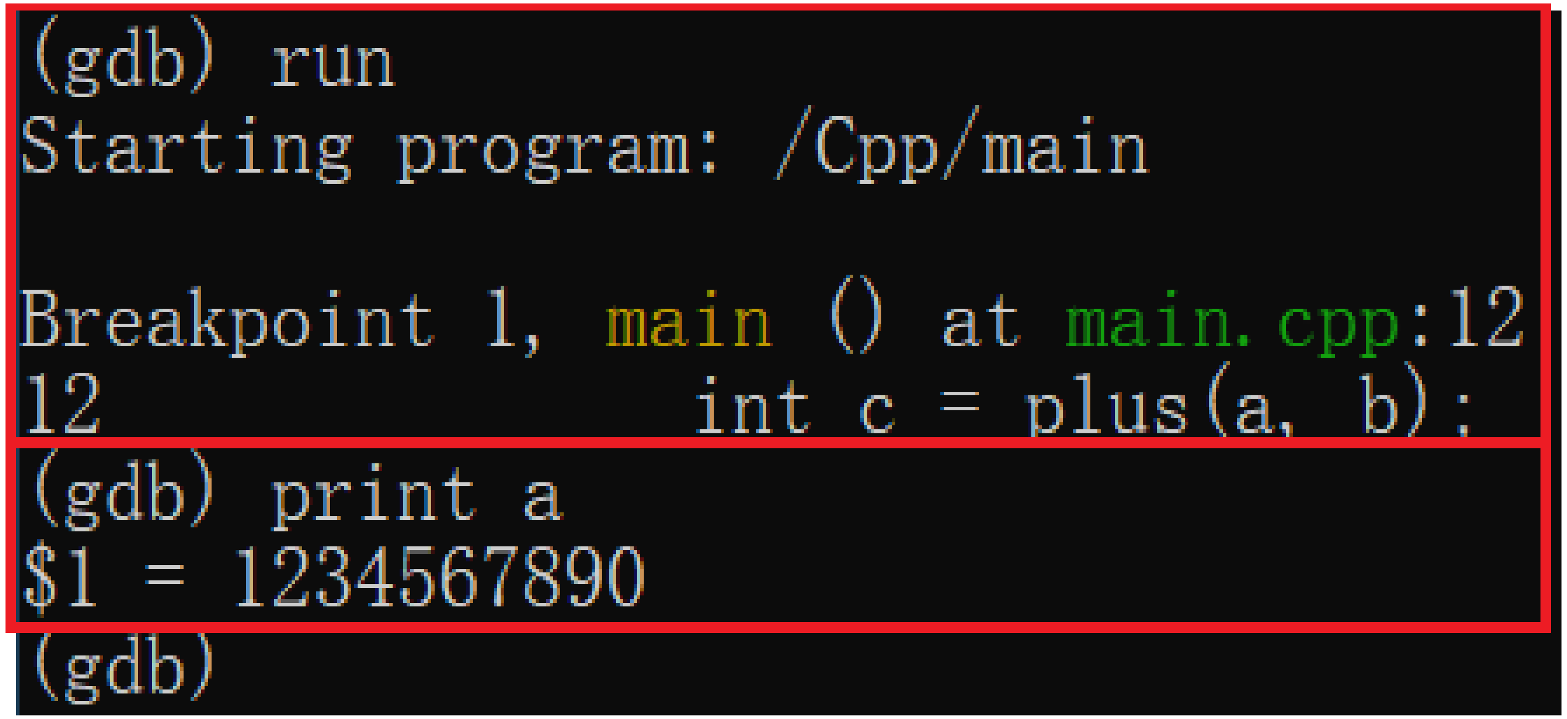
- Now let’s “execute” run and “print a” again.



```
(gdb) run
Starting program: /Cpp/main

Breakpoint 1, main () at main.cpp:12
12              int c = plus(a, b);
(gdb) print a
$1 = 1234567890
(gdb)
```

Debugging with gdb

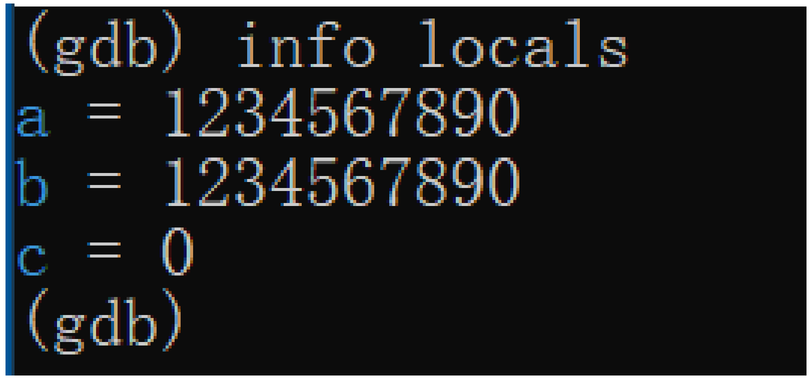
- Two information we can get from here:
- The run command will stop at the position of the break point as line 12.
- The print command prints the variable “a” successfully.



```
(gdb) run
Starting program: /Cpp/main

Breakpoint 1, main () at main.cpp:12
12              int c = plus(a, b);
(gdb) print a
$1 = 1234567890
(gdb)
```

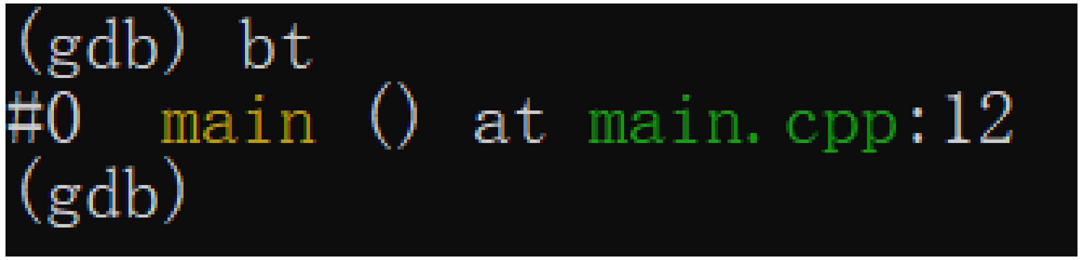
Debugging with gdb

- Using “info locals” to show all the variables in the current stack:



```
(gdb) info locals
a = 1234567890
b = 1234567890
c = 0
(gdb)
```

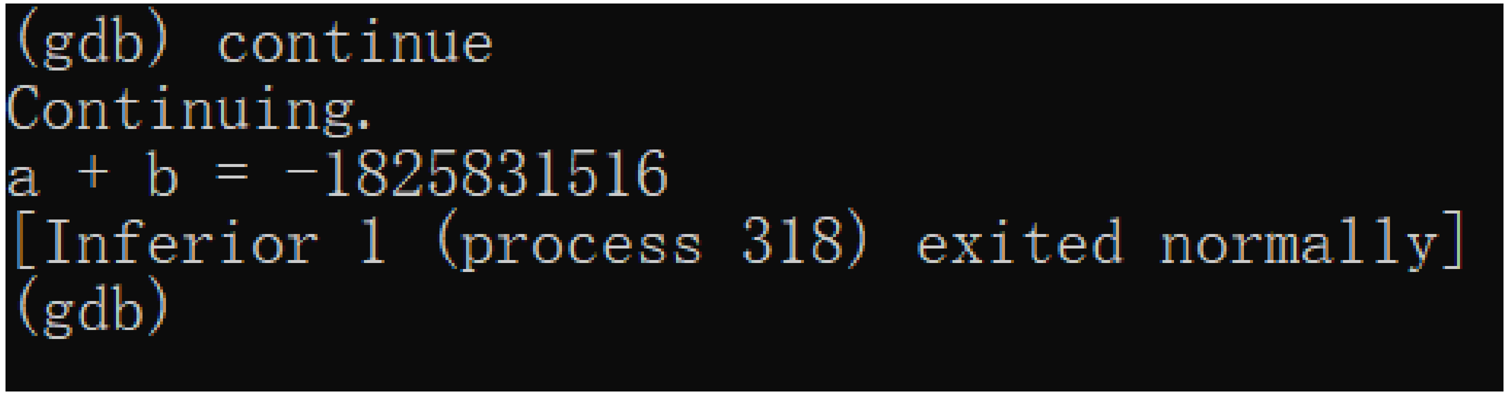
- And use “bt” to print the function stack:



```
(gdb) bt
#0  main () at main.cpp:12
(gdb)
```

Debugging with gdb

- Now that our program is waiting here.
- We can use “continue” command to make the program continue to run.



```
(gdb) continue
Continuing.
a + b = -1825831516
[Inferior 1 (process 318) exited normally]
(gdb)
```

Debugging with gdb

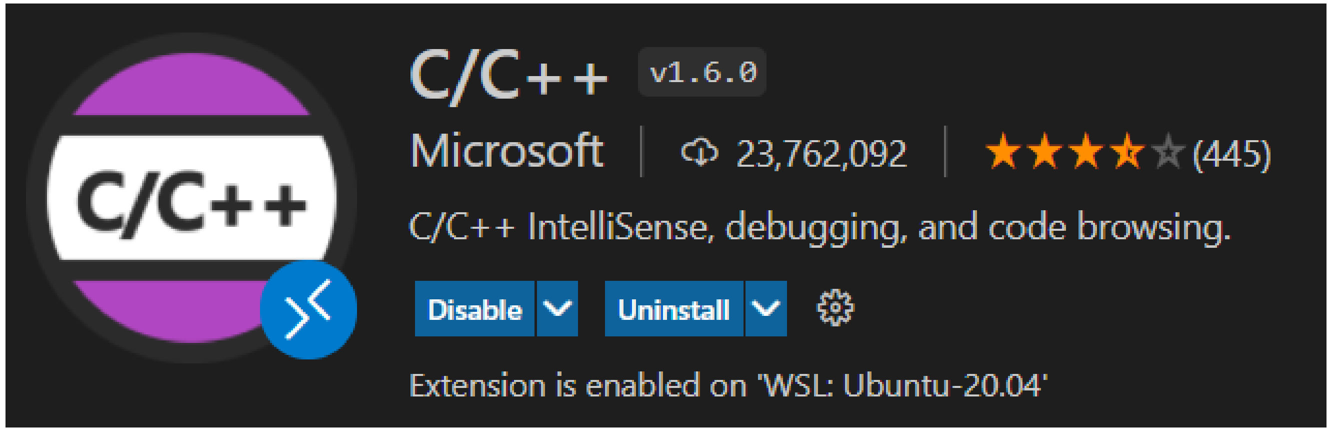
- You may need to refer to the GNU document for more detail:
- https://www.gnu.org/software/gdb/documentation/

- Or if you could search for a beginner tutorial.

Debugging with vscode

- After you know how to use gdb in a command line, you may want to use vscode to do the debugging.

- From the directory of your C/C++ codes, run “code .” to start vscode.
- You probably have already installed the C/C++ extension, if not, install it:



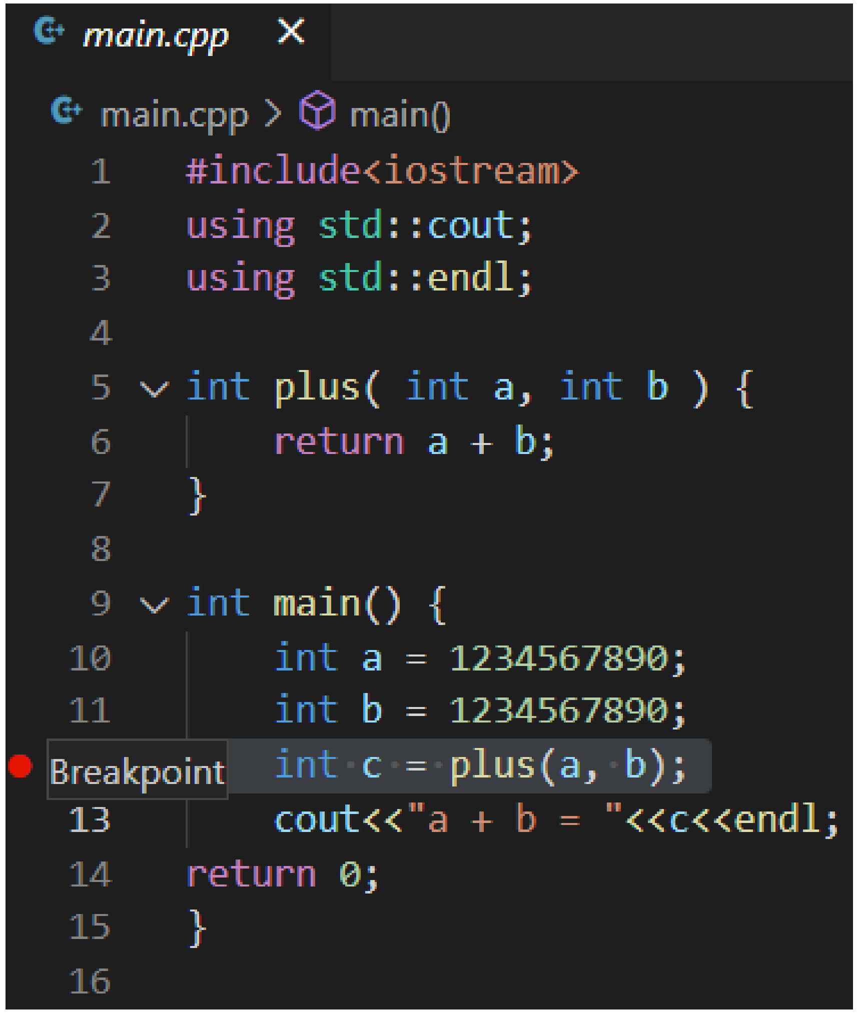
C/C++ IntelliSense, debugging, and code browsing.

Extension is enabled on 'WSL: Ubuntu-20.04'

Debugging with vscode

- Not when you click on the left, you can easily set break points.

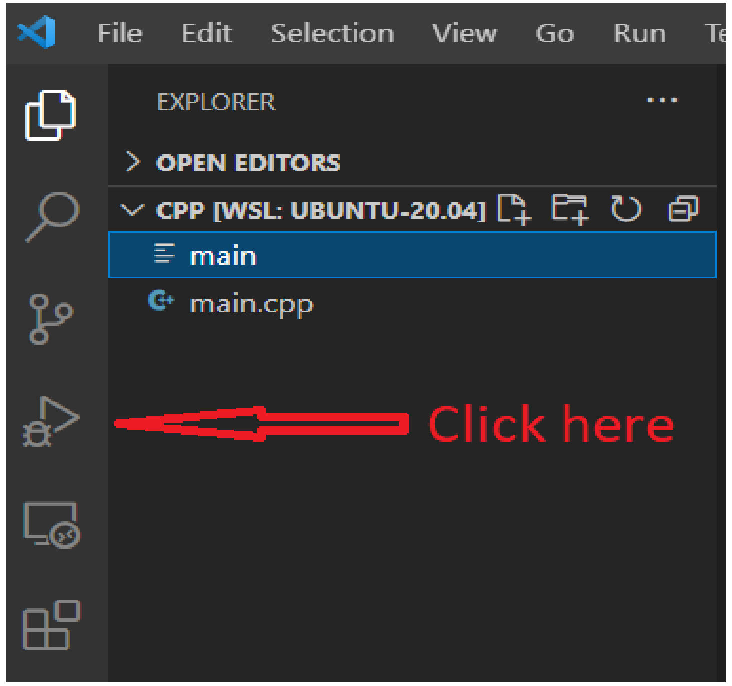
- You can remove a break point by clicking again.

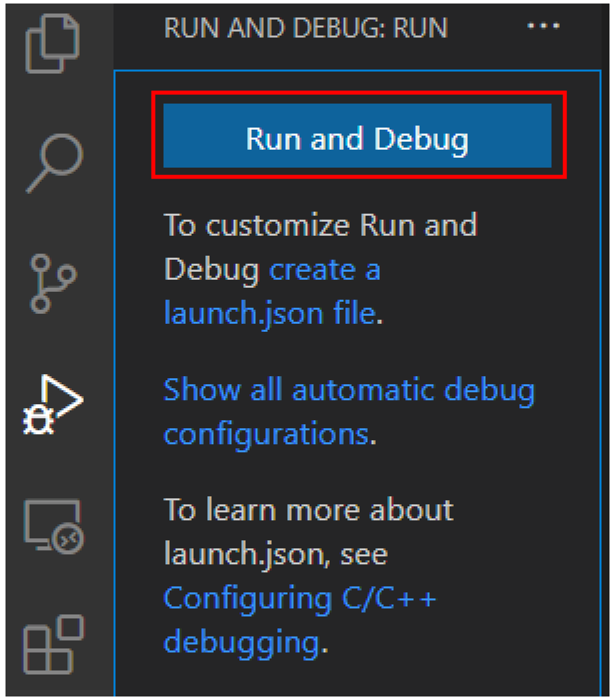


```cpp
#include<iostream>
using std::cout;
using std::endl;

int plus( int a, int b ) {
    return a + b;
}

int main() {
    int a = 1234567890;
    int b = 1234567890;
    int c = plus(a, b);
    cout<<"a + b = "<<c<<endl;
return 0;
}

```

Debugging with vscode

- Now click the “run and debug” to debug the program.
- Everything will run automatically.

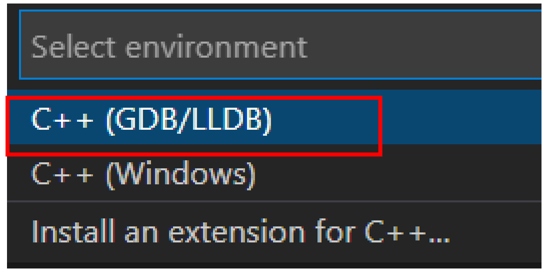


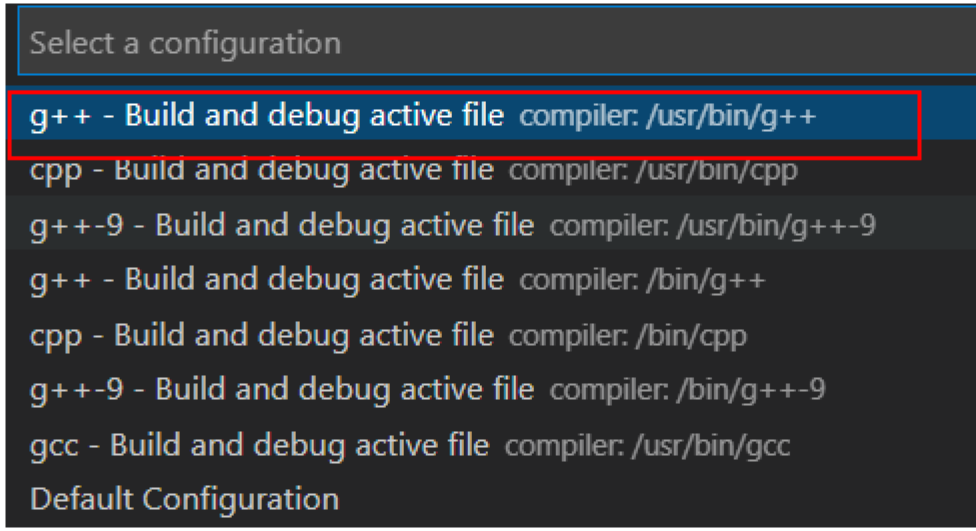


To customize Run and Debug create a launch.json file.

Show all automatic debug configurations.

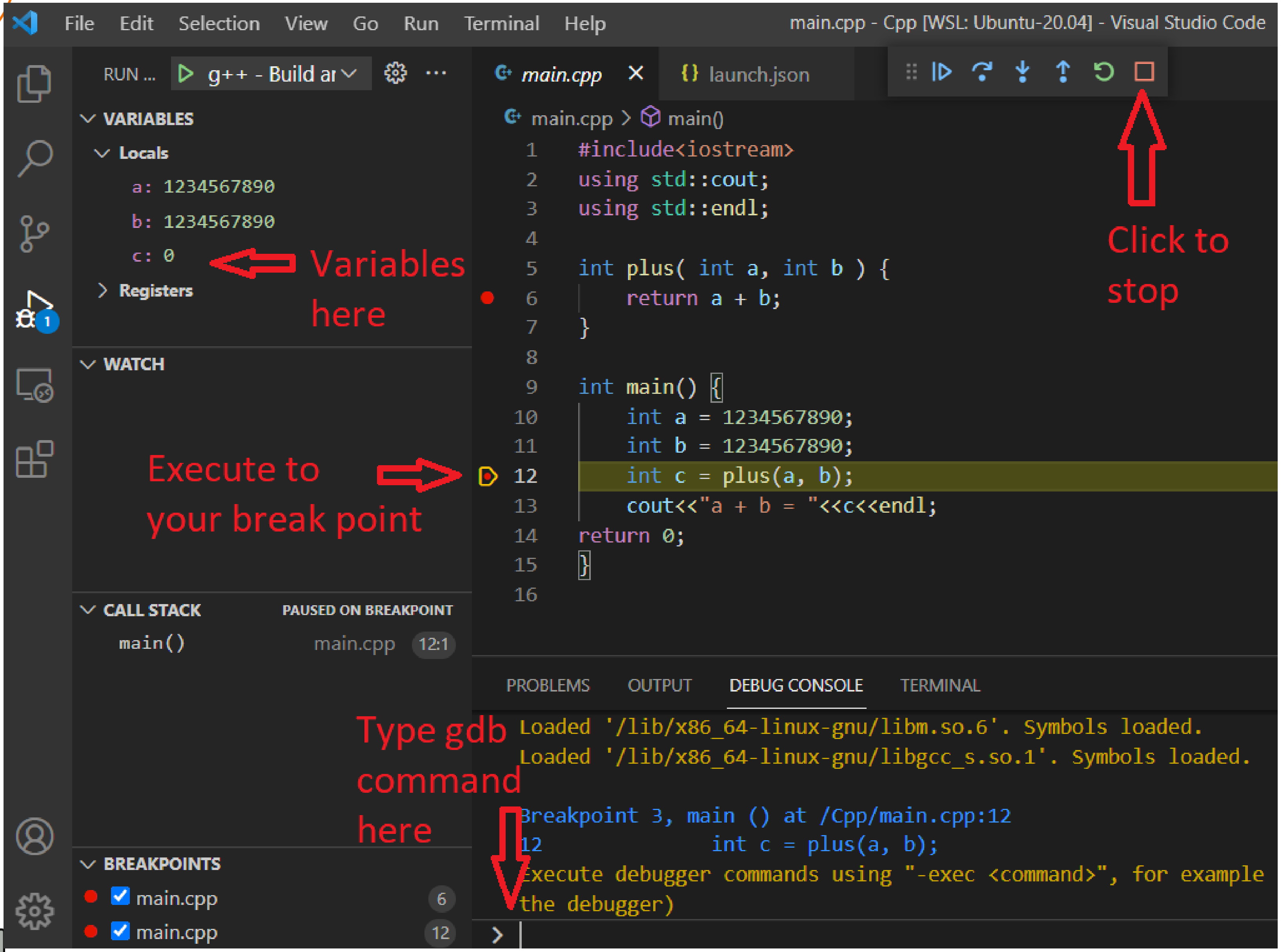
To learn more about launch.json, see Configuring C/C++ debugging.





g++ - Build and debug active file compiler: /usr/bin/g++
cpp - Build and debug active file compiler: /usr/bin/cpp
g++-9 - Build and debug active file compiler: /usr/bin/g++-9
g++ - Build and debug active file compiler: /bin/g++
cpp - Build and debug active file compiler: /bin/cpp
g++-9 - Build and debug active file compiler: /bin/g++-9
gcc - Build and debug active file compiler: /usr/bin/gcc
Default Configuration

Debugging with vscode



File Edit Selection View Go Run Terminal Help
main.cpp - Cpp [WSL: Ubuntu-20.04] - Visual Studio Code
RUN ... g++ - Build ar
main.cpp
launch.json
main.cpp > main()
Variables here
Click to stop
Execute to your break point
Type gdb command here
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

VARIABLES
Locals

```
a: 1234567890
b: 1234567890
c: 0
```

Registers

WATCH

CALL STACK — PAUSED ON BREAKPOINT

```
main()    main.cpp  12:1
```

BREAKPOINTS

✓	main.cpp	6
✓	main.cpp	12

```
1   #include<iostream>
2   using std::cout;
3   using std::endl;
4
5   int plus( int a, int b ) {
6       return a + b;
7   }
8
9   int main() {
10      int a = 1234567890;
11      int b = 1234567890;
12      int c = plus(a, b);
13      cout<<"a + b = "<<c<<endl;
14  return 0;
15  }
16
```

```
Loaded '/lib/x86_64-linux-gnu/libm.so.6'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libgcc_s.so.1'. Symbols loaded.

Breakpoint 3, main () at /Cpp/main.cpp:12
12              int c = plus(a, b);
Execute debugger commands using "-exec <command>", for example
the debugger)
>
```

https://code.visualstudio.com/Docs/editor/debugging

Exercise 1

Write a program that use ***new*** to allocate the array dynamically of five integers.

- The five values will be stored in an array using a pointer.
- Print the elements of the array in reverse order using a pointer.

Exercise 2

Allocate memory for an array of characters, modify elements by integer values one by one, then print out the result as a string. Please try to modify the element out of range and see what will happen.

Exercise 3

- Using gdb or vscode to debug the following program:

```
#include<iostream>
using std::cout;
using std::endl;

// naive approach to see if num is a prime number
bool isPrime( int num ) {
        for( int i = 1; i <= num; ++ i )
                if( num % i == 0 )
                        return false;
        return false;
}

int main() {
        int a = 23;
        cout<<isPrime(a)<<endl;
return 0;
}
```