

# 3

## App Debugging with Code::Blocks

Debugging is an essential step in any app development. It is also an essential part of an IDE and Code::Blocks is no exception. It offers a vast set of features to make app debugging easier.

In this chapter, we will learn about app debugging with Code::Blocks. We'll begin with a simple app to show various features of Code::Blocks.

### Introduction to debugging in Code::Blocks

Code::Blocks supports two debuggers:

- **GNU Debugger** or, as it is popularly known as **GDB**
- Microsoft **Console Debugger** or **CDB**

Code::Blocks installer bundles GDB together with GCC compiler. CDB can be downloaded and installed together with installation of Windows **Software Development Kit (SDK)** for Windows.

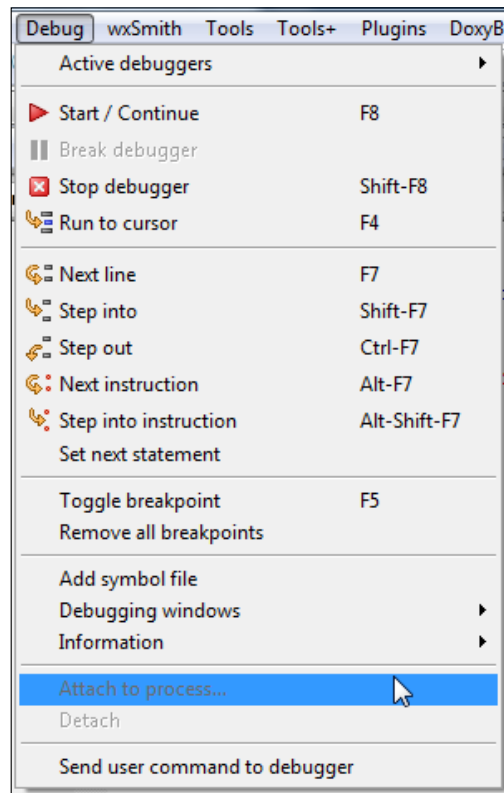


Windows SDK is a collection of tools offered by Microsoft for Microsoft Windows platform. It consists of compiler, headers, libraries, debugger, samples, documentation, and tools required to develop applications for .NET Framework.

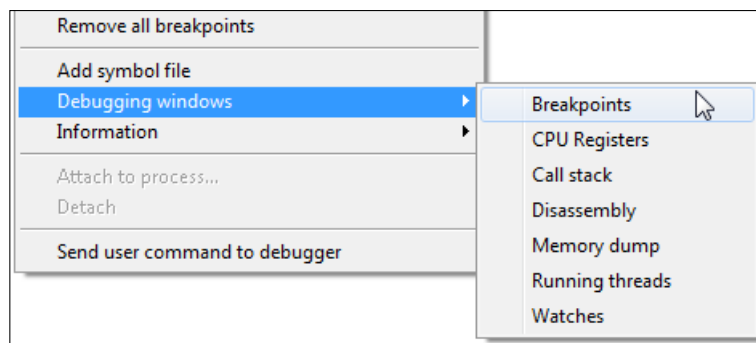
CDB can be downloaded and installed from the following link:

<http://msdn.microsoft.com/en-us/library/windows/hardware/gg463009.aspx>

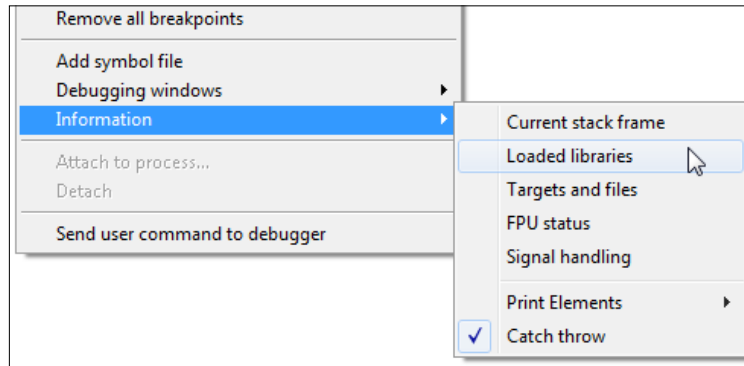
Our focus will be on GDB throughout this chapter. Debugger related functions are available via the **Debug** menu in Code::Blocks as shown in the following screenshot. A debugger toolbar is also provided for quicker access to commonly used functions.



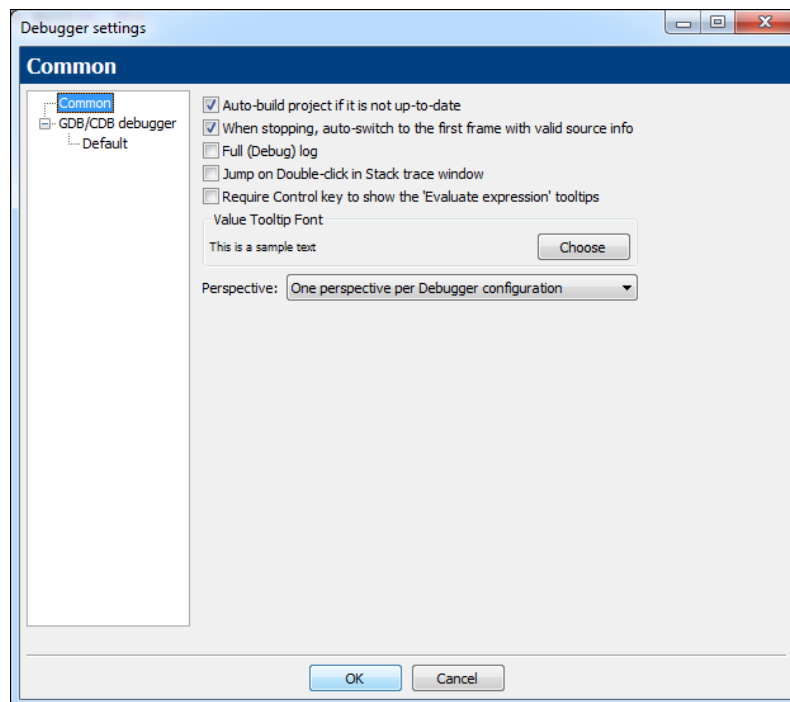
We can access several debugger related windows by navigating to **Debug | Debugging windows** menu options. The following screenshot shows available menu options.



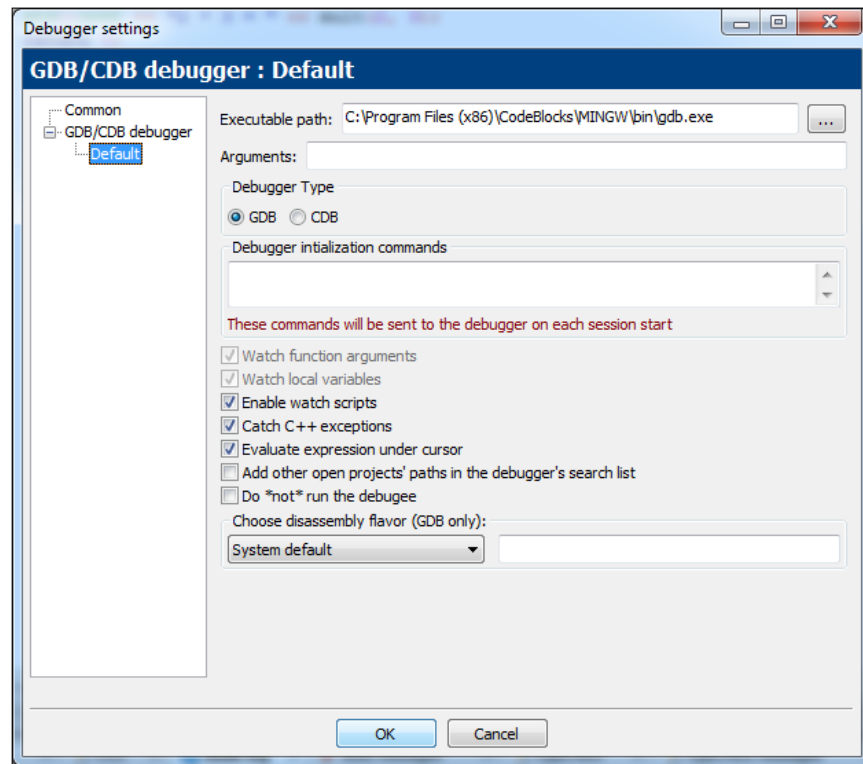
We can get more information about running process from **Debug | Information** and then clicking on appropriate menu option. The following screenshot shows available menu options:



Debugger settings can be accessed by navigating to **Settings | Debugger** menu option. The following screenshot shows the debugger settings dialog:



Select **Default** in the tree on the left-hand side and more debugger related options will be available as shown in the following screenshot:



Select the **Evaluate expressions under cursor** option shown in the previous screenshot. This option will provide a tooltip containing details whenever cursor is moved over a variable.

## First app debugging

Let us create a new console project App7 and replace code inside `main.cpp` file with following code:

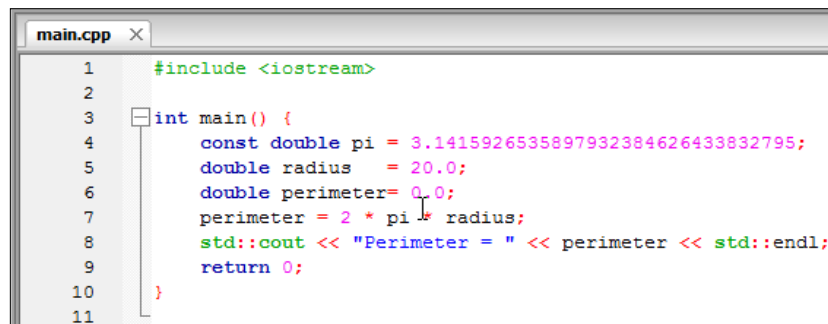
```
#include <iostream>

int main() {
    const double pi = 3.1415926535897932384626433832795;
    double radius   = 20.0;
    double perimeter= 0.0;
```

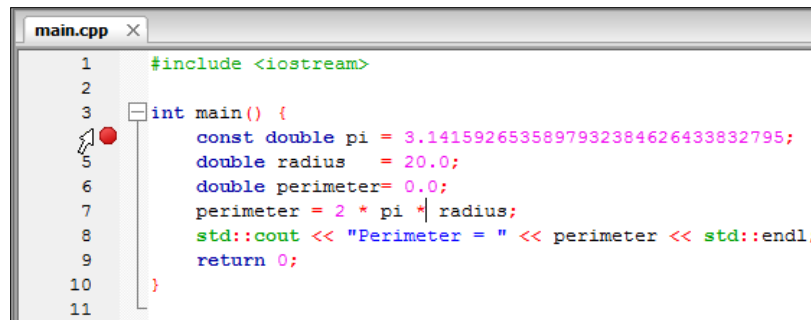
```
    perimeter = 2 * pi * radius;  
    std::cout << "Perimeter = " << perimeter << std::endl;  
    return 0;  
}
```

Ensure that **Debug** target is selected in compiler toolbar and then compile it by clicking compile button. App7 will be compiled for debugging.

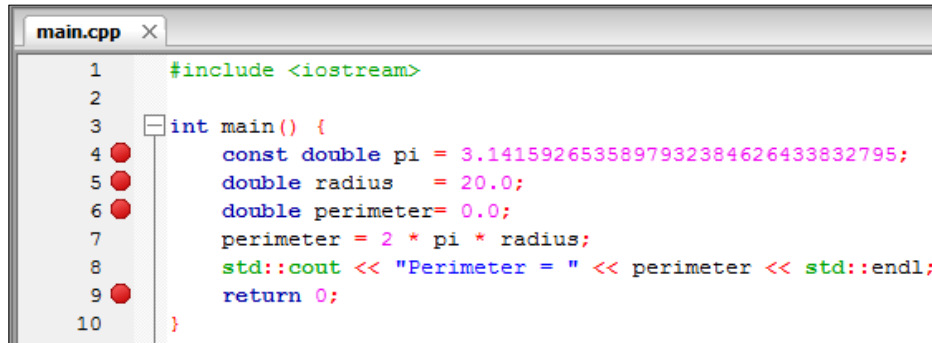
Before we ask GDB to debug we have to create breakpoints for it. After the code is typed in editor window Code::Blocks will look similar to the following screenshot.



To set a breakpoint move cursor to the left side of editor window next to the indicated line numbers. Now the cursor will change to a right-tilted cursor. Pause mouse and left-click. A breakpoint will be set there and will be indicated by a red circle. The following screenshot shows that a breakpoint has been set at line number 4.

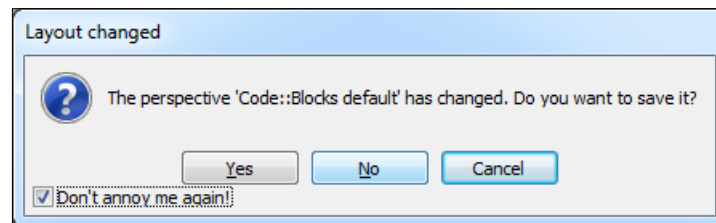


Next follow same method and create breakpoints at line numbers 5, 6 and 9. Editor window will now look similar to the following screenshot:



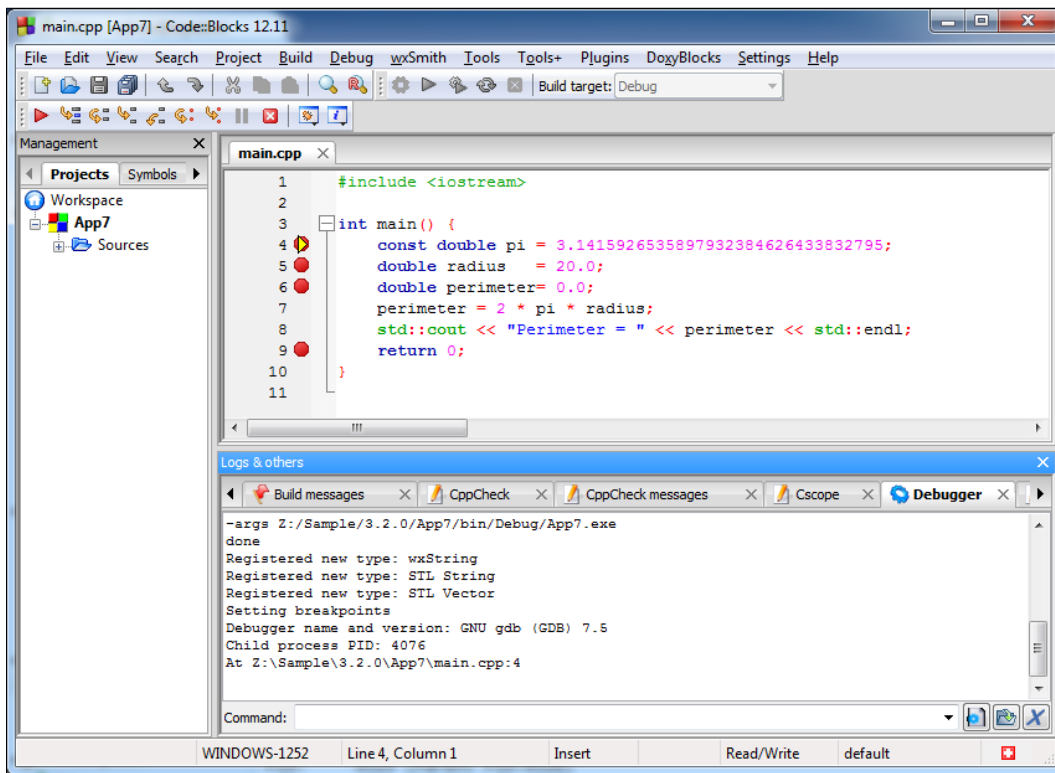
All breakpoints are now visually indicated in the editor window.

We can now start debugging by clicking on the **Debug/Continue** button in debugger toolbar. Alternatively the *F8* key may be used to start debugging. The following window may appear:



This highlights that default layout of Code::Blocks has changed as the **Debugger log** window has received focus (refer to the preceding screenshot). Select the **Don't annoy me again!** checkbox and then click on **No** button to stop it. It won't appear again. Let's look at the entire IDE now.

In the following screenshot execution has stopped at line number 4 and the cursor has changed to a yellow colored triangle. This indicates that debugger has stopped execution at that position. Debugger log window will also be updated when we continue debugging.

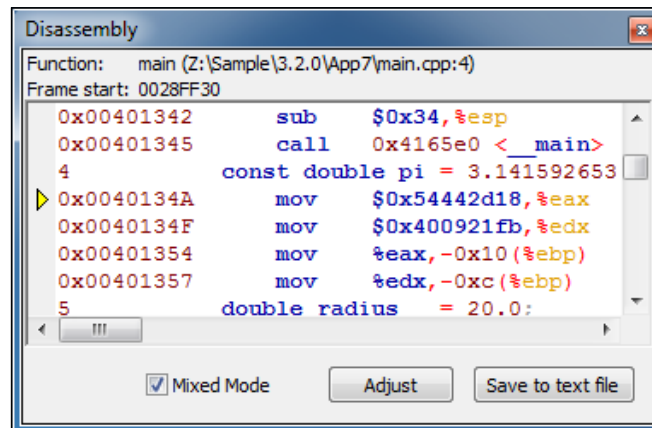


Before continuing with debugging we take a look at debugger related features of Code::Blocks. **CPU Registers** can be examined by navigating to the **Debug | Debugging windows | CPU Registers** menu option. A register is a tiny but a high speed buffer embedded within the processor hardware.

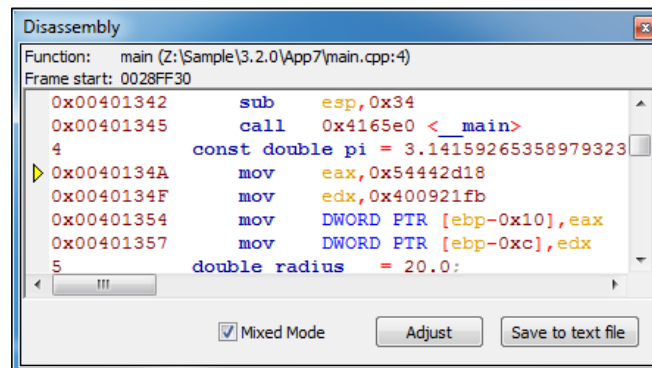
The screenshot shows the **CPU Registers** window, which displays the current state of the CPU registers. The window has a table with three columns: **Register**, **Hex**, and **Integer**.

Register	Hex	Integer
eax	0x1	1
ecx	0x1	1
edx	0x8e3c8	582600
ebx	0x7efde000	2130567168
esp	0x28fee0	2686688
ebp	0x28ff18	2686744
esi	0x0	0
edi	0x0	0
eip	0x40134a	4199242
eflags	0x202	514

Now navigate to the **Debug | Debugging windows | Disassembly** menu option; this can be used to display assembly language representation of current C++ code. The following screenshot shows the **Disassembly** window and also indicates the position where execution has stopped. Clicking on the **Mixed Mode** checkbox will superimpose C++ code and corresponding assembly language code:

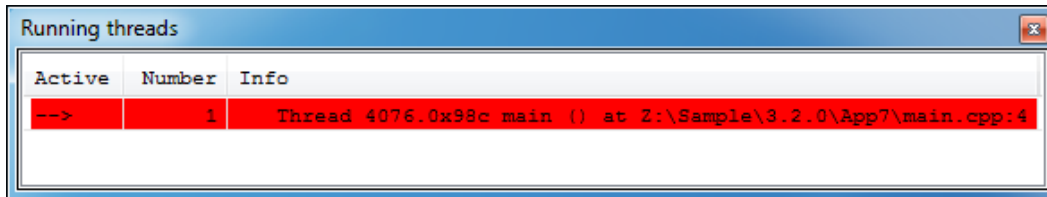


This style of assembly language is known as **AT&T style**. We can switch to **Intel** style assembly language in the disassembly dialog by navigating to the **Settings | Debugger | GDB/Debugger | Default** menu option and selecting the **Intel** option in **Choose disassembly flavor** (GDB only) combo box. Now close the previously opened disassembly dialog and reopen it. It will now show disassembly in Intel flavor as shown in the following screenshot. Please note that the choice of AT&T or Intel style is up to the preference of a developer. It has no effect on the debugging process.

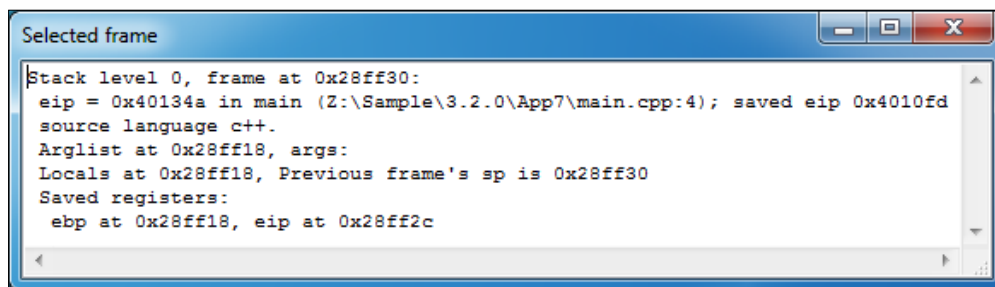




Currently running threads can be examined by navigating to **Debug | Debugging windows | Running threads** menu option. This app is single threaded and thus in the following screenshot we find that only one thread is running:



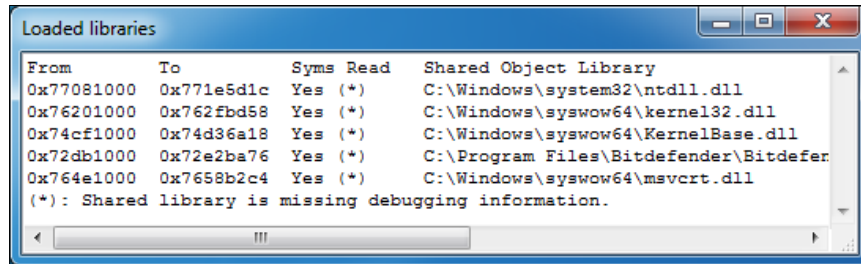
Stack frame can be examined by navigating to **Debug | Information | Current stack frame** menu option. Call stack is a data structure that stores information about current running function. The following screenshot shows the stack frame information of current process:



Call stack is a data structure that works on the principle of **(Last In First Out)** and stores information about active subroutines or program. Stack frame is part of call stack that stores information (local variables, return address and function parameters) of a single subroutine or function.

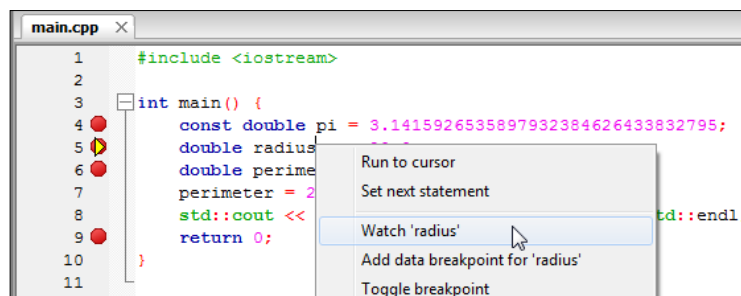
Whenever an app is run on Windows platform several **Dynamic Link Libraries (DLL)** or dynamic libraries are loaded in memory. DLL provide functions that are accessible by other apps without including a copy of function code inside the apps using it. Loaded libraries can be examined by navigating to **Debug | Information | Loaded libraries** menu option.

The following screenshot shows the loaded libraries for our app:

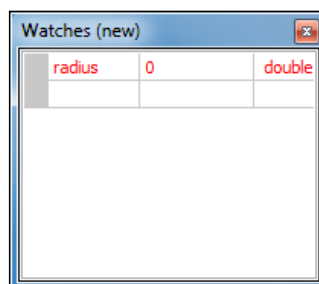


The asterisk next to the DLL name indicates whether their source can be debugged. We find that none of them allows debugging.

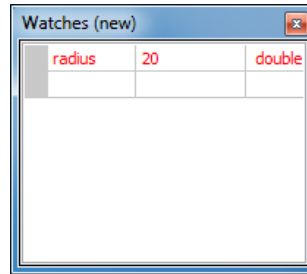
Now we'll continue with debugging after our introduction to several debugger related windows. We'll also learn to set watch on a variable. Click on the **Continue** button and debugger will stop at line number 5. Right-click on **radius** variable in editor window and then choose watch 'radius' menu option.



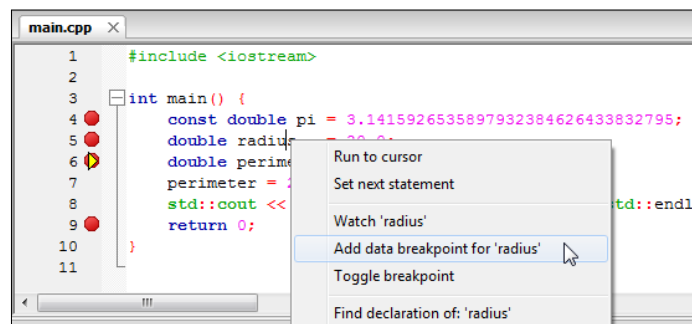
This will create a watch on the variable **radius**. A watch can be defined as an instruction to the debugger to track a variable during execution of an app. A separate window with the variable under watch will now be opened as shown in the following screenshot. Watch window can also be opened via **Debug | Debugging Windows | Watches** menu option:



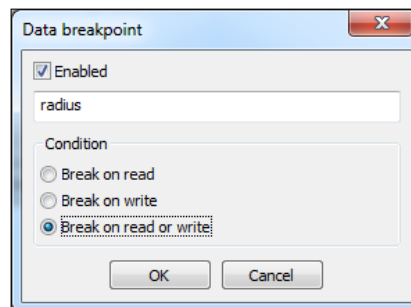
If we click on the **Continue** button again then execution of app will advance to next line. This will update content of the `radius` variable in our app. Watch window will also update its content showing current value of the `radius` variable as shown in the following screenshot:



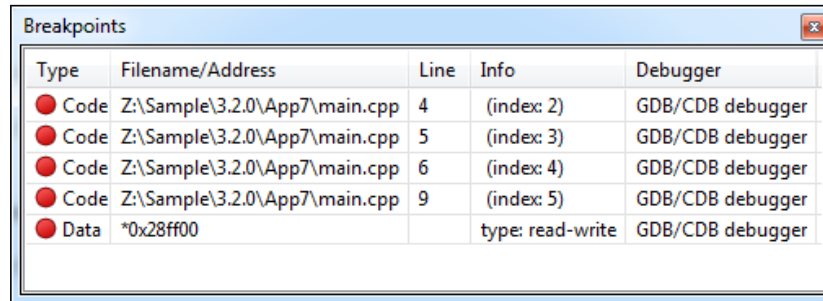
At this step we'll learn about another type of breakpoint known as **data breakpoint**. Right-click on the `radius` variable in line number 5 in editor window shown in the following screenshot and then click on the **Add data breakpoint for 'radius'** menu option:



Select **Break on read or write** option as in the following screenshot and click on the **OK** button. By doing this we are instructing GDB to pause execution whenever the `radius` variable is read or written.

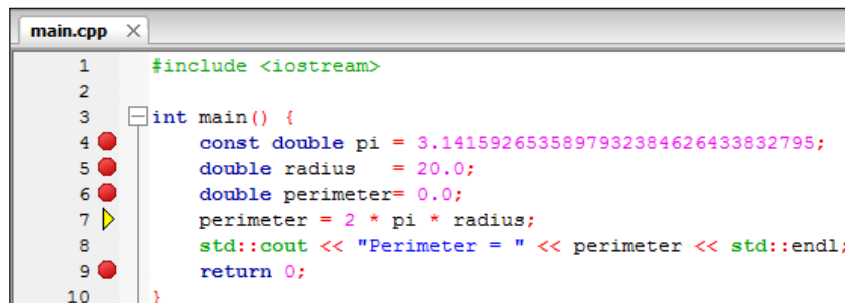


A data breakpoint will now be created. However data breakpoint is not shown visually in editor window. It can be verified from **Breakpoints** window by navigating to the **Debug | Debugging windows | Breakpoints** menu option. Last line in the following screenshot shows that a data breakpoint has been set.



Type	Filename/Address	Line	Info	Debugger
Code	Z:\Sample\3.2.0\App7\main.cpp	4	(index: 2)	GDB/CDB debugger
Code	Z:\Sample\3.2.0\App7\main.cpp	5	(index: 3)	GDB/CDB debugger
Code	Z:\Sample\3.2.0\App7\main.cpp	6	(index: 4)	GDB/CDB debugger
Code	Z:\Sample\3.2.0\App7\main.cpp	9	(index: 5)	GDB/CDB debugger
Data	*0x28ff00		type: read-write	GDB/CDB debugger

Click on the **Continue** button in debugger toolbar or press the *F8* key and execution will continue. It will now stop at line 7 due to the data breakpoint we have set in previous step. Variable `radius` is being read at this line and `gdb` has stopped execution as data breakpoint condition has been met.



```
1  #include <iostream>
2
3  int main() {
4      const double pi = 3.1415926535897932384626433832795;
5      double radius  = 20.0;
6      double perimeter= 0.0;
7      perimeter = 2 * pi * radius;
8      std::cout << "Perimeter = " << perimeter << std::endl;
9      return 0;
10 }
```

Click on the **Continue** button to continue execution of app and subsequently it will stop at line number 9. If we continue clicking on the **Continue** button app, execution will stop several times due to the data breakpoint we have set earlier. This is normal and in order to stop execution immediately click on the **Stop** button in debugger toolbar or press *Shift + F8* key to stop execution.

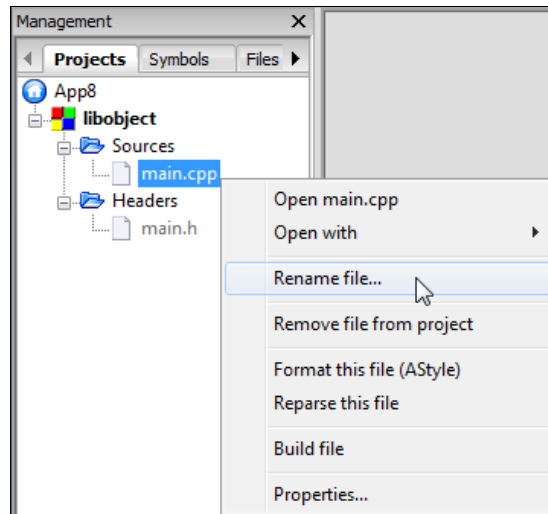
This completes our introduction to app debugging with Code::Blocks.

## Multiple app debugging

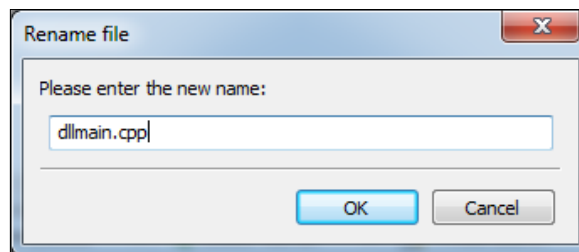
Real life projects are large in size and may consist of several sub-projects. It is essential that an IDE allows debugging of large apps spanning across several projects. With Code::Blocks we can do it easily.

To learn multiple app debugging we'll create two projects – first project a DLL project and second one is a console project that depends upon first DLL project. Then save both projects under same workspace named App8.

Go to **File | New | Project | Dynamic Link Library** menu option to create a DLL project. Name this project libobject. Now rename the libobject project files. We'll rename `main.h` file to `dllmain.h` and `main.cpp` to `dllmain.cpp` file. To do this, close all open editor files and right-click on the file name in the project tree as shown in the following screenshot:



Enter new file name in the dialog box shown in following screenshot:



This will avoid ambiguities in file names. Now replace code inside `dllmain.h` file with the following code.

```
#ifndef __DLLMAIN_H__
#define __DLLMAIN_H__

/* To use this exported function of dll, include this header
 * in your project.
 */

#ifdef BUILD_DLL
#define DLL_IMP_EXPORT __declspec(dllexport)
#else
#define DLL_IMP_EXPORT __declspec(dllimport)
#endif

#ifdef __cplusplus
extern "C"
{
#endif
    void DLL_IMP_EXPORT SayHello(void);
#ifdef __cplusplus
}
#endif

class base {
public:
    void Set(int width, int height) {
        m_width = width;
        m_height = height;
    }
    virtual int Area() = 0;
protected:
    int m_width, m_height;
};

class DLL_IMP_EXPORT Rectangle : public base {
public:
    int Area();
};

class DLL_IMP_EXPORT Triangle : public base {
public:
    int Area();
};

#endif // __DLLMAIN_H__
```

A DLL on Windows requires special decoration in order to export it from a dynamic link library. This decoration statement changes while it is exported and at the time it is imported. Decoration `__declspec(dllexport)` is used to export functions from a DLL and `__declspec(dllimport)` is used to import function from another DLL. Decorations instruct linker to export or import a variable/function/object name with or without name mangling. A preprocessor define `DLL_IMPORT_EXPORT` is used to indicate compiler whether a function or a class is being exported or imported.

C++ allows function/method overloading. It is achieved by introducing name mangling in the generated code. Name mangling is a process in which a function name is converted to a unique name based on function parameters, return type, and other parameters. Name mangling is compiler dependent and as a result any DLL written in C++ can't be used directly with another compiler.

C++ introduces name mangling by default for all functions. We can stop name mangling using `extern "C"` keyword and are using it to stop name mangling for the exported `SayHello()` function. By stopping name mangling we can use a DLL written in C++ and compiled with one compiler to be used with another compiler.

We have defined a class `base` and this base class has a member function `Set()` and it sets two internal variables. There is a pure virtual function named `Area()` that must be redefined derived classes. A **pure virtual function** is a function that has not been implemented in the base class. If a pure virtual function is called in any app it may result in a crash.

However, this base class is not decorated with `DLL_IMPORT_EXPORT`. This means it will not be exported in DLL and no outside app can use this class.

In order to use feature of the base class we'll create two derived classes. Class `Rectangle` and `Triangle`, these are derived publicly from the base class. We have used inheritance of classes here. These classes are declared with decoration `DLL_IMPORT_EXPORT`. Thus these two classes will be exported in the resulting DLL.

Now replace code inside the `dllmain.cpp` file of the `libobject` project with the following code:

```
#include <windows.h>
#include <iostream>

#include "dllmain.h"

void SayHello(void) {
    std::cout << "Hello World!" << std::endl;
}
```

```
int Rectangle::Area() {
    return (m_width * m_height);
}

int Triangle::Area() {
    return (m_width * m_height / 2);
}

extern "C" DLL_IMP_EXPORT BOOL APIENTRY DllMain(HINSTANCE hinstDLL,
DWORD fdwReason, LPVOID lpvReserved) {
    switch (fdwReason) {
        case DLL_PROCESS_ATTACH: // attach to process
            // return FALSE to fail DLL load
            break;
        case DLL_PROCESS_DETACH: // detach from process
            break;
        case DLL_THREAD_ATTACH: // attach to thread
            break;
        case DLL_THREAD_DETACH: // detach from thread
            break;
    }
    return TRUE; // successful
}
```

Code in the `dllmain.cpp` file mainly defines all the code of publicly exported function. There is a `DllMain()` function. It may be used to do any initialization or de-initialization for the DLL.

Next create a console app named App8. Now rename workspace as App8 and save workspace as App8. This console app will use functions defined in `libobject.dll`. Replace code inside the `main.cpp` file of App8 with the following code:

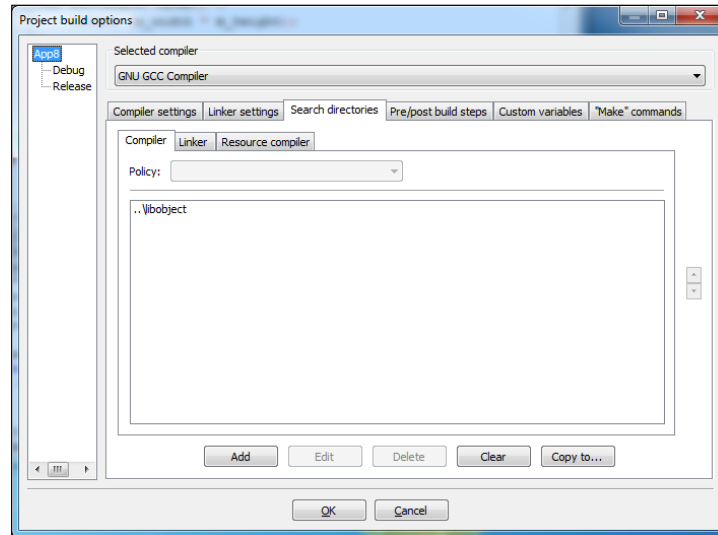
```
#include <iostream>

#include "dllmain.h"

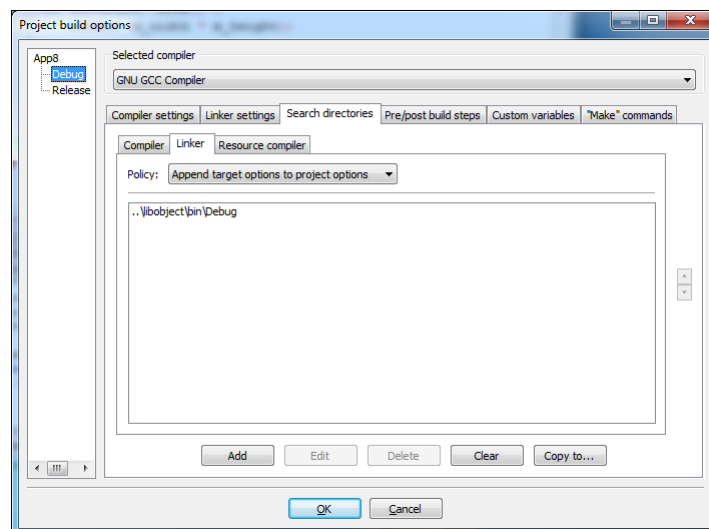
int main() {
    Rectangle rect;
    rect.Set(10, 20);
    Triangle trgl;
    trgl.Set(5, 6);
    std::cout << "Rectangle(10, 20).Area() = " << rect.Area() <<
std::endl;
    std::cout << "Triangle(5, 6).Area() = " << trgl.Area() <<
std::endl;
    return 0;
}
```



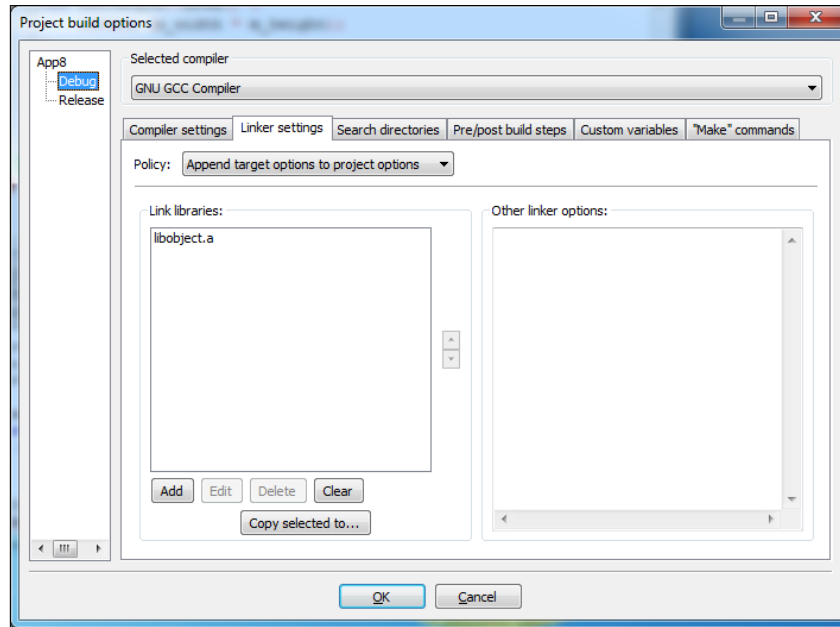
Next, we have to prepare our App8 project to use this DLL. To do so go to **Project | Build options** menu option. Select App8 in the project tree and then click on **Search directories** tab. Then add `..\libobject` directory to the list in the **Compiler** tab. This instructs compiler to search for header files in that directory:



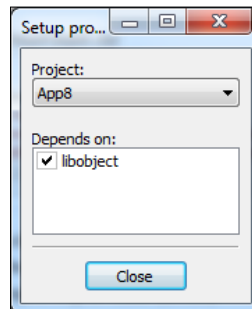
We also need to point linker to the directory where we have kept import library of `libobject.dll` file. To do so select the **Debug** target and click on the **Search directories** tab. Then click on the **Linker** tab and add `..\libobject\bin\Debug` folder to the list:



We have to instruct linker to find references of symbols found in `libobject.dll` file. To do so click on the **Linker settings** tab and add `libobject.a` to the **Link libraries** list.



We'll set up project dependencies in this step. Go to **Project | Properties...** menu option and then click on the **Project dependencies...** button. Click on the `libobject` and then click on the **Close** button. Finally click **OK** button to close the **Project/targets** options window. This completes preparation of the `App8` console app.

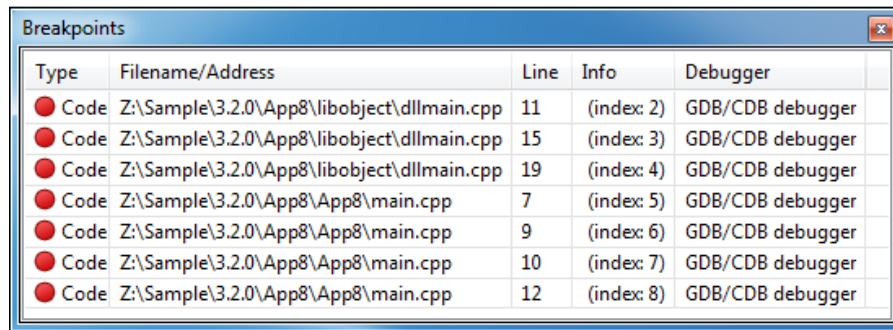


Now go to **Build | Build workspace** menu option. This will build the `libobject` project first and subsequently `App8` will be compiled.

In order to learn debugging multiple projects we'll set breakpoints at the following line number:

- Line number 11, 15, 19 in the `dllmain.cpp` file, `libobject` project
- Line number 7, 9, 10, 12 in the `main.cpp` file, `App8` project

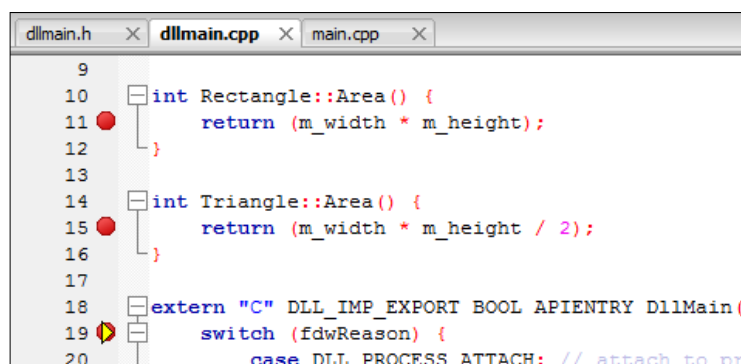
Breakpoints can be verified from **Breakpoints** window shown in the following screenshot:



Type	Filename/Address	Line	Info	Debugger
Code	Z:\Sample\3.2.0\App8\libobject\dllmain.cpp	11	(index: 2)	GDB/CDB debugger
Code	Z:\Sample\3.2.0\App8\libobject\dllmain.cpp	15	(index: 3)	GDB/CDB debugger
Code	Z:\Sample\3.2.0\App8\libobject\dllmain.cpp	19	(index: 4)	GDB/CDB debugger
Code	Z:\Sample\3.2.0\App8\App8\main.cpp	7	(index: 5)	GDB/CDB debugger
Code	Z:\Sample\3.2.0\App8\App8\main.cpp	9	(index: 6)	GDB/CDB debugger
Code	Z:\Sample\3.2.0\App8\App8\main.cpp	10	(index: 7)	GDB/CDB debugger
Code	Z:\Sample\3.2.0\App8\App8\main.cpp	12	(index: 8)	GDB/CDB debugger

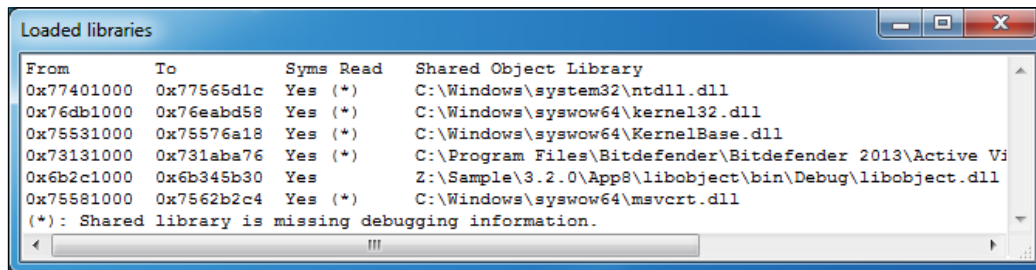
Note that DLLs can't run as a standalone process and require a host application to load them into memory. In order to debug a DLL we have to debug the host application that loads and runs it. Alternatively we can specify a host application (in our case `App8.exe`) for debugging by navigating to **Project | Set programs' arguments...** menu option.

We'll use first approach and let our host app to load `libobject.dll`, then use it to debug both `libobject.dll` and `App8.exe` file. Ensure that `App8` project is activated in the project tree and then click on the debug/continue button in debugger toolbar:

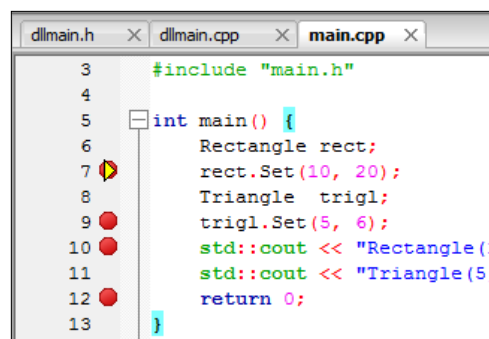


In the preceding screenshot execution has stopped at line number 19 of the `dllmain.cpp` file. Whenever `DllMain()` is exported it becomes the first function to be called during the loading/unloading of any DLL. As a result execution stops there.

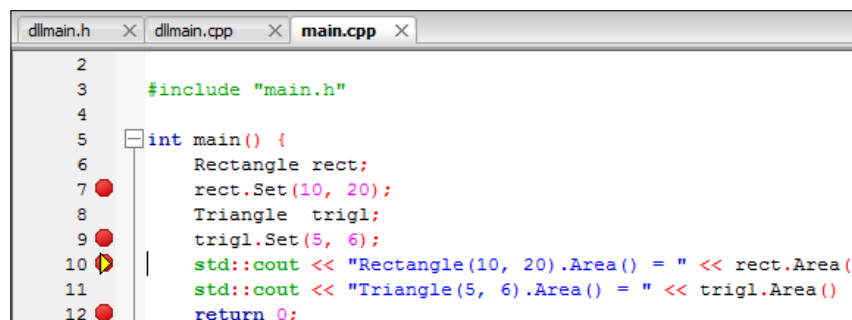
Loaded libraries window in the following screenshot confirms that `libobject.dll` has been loaded in memory and this library can be debugged:



Click on the **Continue** button to continue. Execution will now pause at line number 7 of the `main.cpp` file.

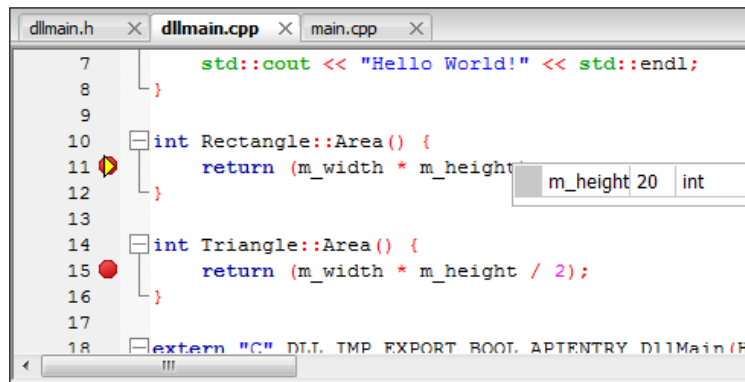


Click on the **Continue** button twice. Execution will stop at line number 10 of the `main.cpp` file as shown in the following screenshot:



Click on the **Continue** button again and execution will stop at line number 11 of `dllmain.cpp` file.

Debugger is now debugging `libobject` project's source file, which is a separate project. If cursor is hovered `m_height` variable debugger will evaluate this variable and show its value.



It is evident that we can debug both DLL project and console app project at the same time. Larger projects can be debugged using a similar method. With this example we conclude our multiple app debugging session. Click on the **Stop** button to stop debugging.

## Summary

In this chapter we learned app debugging with Code::Blocks using GNU GDB debugger. We learned various debugging related tools provided by Code::Blocks. Subsequently we learned debugging single and multiple apps.

In the next chapter we'll discuss app development for Windows.