## **5. Introduction to Eclipse**

Up to this point, we have used a text editor such as emacs (gedit?). First, write a program in C language → compile by gcc → link by gcc → execute the executable file.

(In the case of a simple program, "compile-> link" was done at once.)

With this method, if a compile error or run-time error occurs, you have to go back to the text editor and identify the error part by yourself.

This is fine as long as the program is short, but it is inefficient when it gets long.

Therefore, in this lecture, we will introduce an integrated development environment (IDE), [eclipse](http://www.eclipse.org/).

This is software that allows you to use all the functions of the editor, compiler, and debugger in one application.

(Roughly speaking, eclipse manages text editing, executes the compiler, linker, and debugger from the application program without showing it to the user, then analyzes the message obtained at that time well, and uses it in the form of GUI.)

(If you write a program in Visual Studio of Microsoft Windows, you can think of it as a free Linux version.)

Eclipse originally aims to support programming for the java language. Because of its ease of use and the extensibility of the plug-in format, it is expanded to other languages ​​one after another.

One of them is "An IDE for C / C++ developers", or “EclipseCPP”, which claims to support the C / C++ language.

(This class does not explain how to use eclipse in other languages ​​such as Java. In addition, the eclipse is still being upgraded, as is the case with open software. As of 2022/01, ver. 2020-03 (4.15.0) is used in the Faculty of Engineering Systems.)

(Since eclipse changes every year, this page must be rewritten every year.)

(By the way the [release history](https://www.eclipse.org/downloads/packages/release) as of 2022/01/05 says the version 2021-12 is the latest. The update is quick.)

(Persons who personally want to use the latest version of the C / C++ development environment should visit the release history page and get the latest R (Release) version)

### **05.01. Work concept**

Even when eclipse is used, The process of “editing-> compilation-> error debugging at compilation-> link-> run-time debugging-> completion,” does not change.

In the explanation in the integrated development environment, first, divide the process to run the final executable from writing a program and give each one a name.

**05.01.01. Two types of perspectives and two types of executable files**

There are many small windows on the work screen of Eclipse.

A layout of the small windows for a certain work purpose in eclipse is called a perspective.

Our program is created on theC/C++ perspective.

In the editor window in the C/C++ perspective, you write the C language source.

Of course, it is also possible to use split source files.

While editing the program, the grammar check is automatically performed every time there is any change, and if an error is found, the part that seems to be the error is displayed on the edit screen of the source file at that point.

(This feature is especially useful when you're modifying a program that was already in good shape.)

Furthermore, when you compile the program, it points out compile-time errors. If possible, it will display the part that seems to be an error on the edit screen of the source file.

Once the program is completed by editing the program and there are no compilation errors, the role of the C/C++ perspective (which is to generate an executable file for run-time debugging for first) is settled.

The executable file obtained here is called the executable file is for debug.

The next thing you need is run-time debugging.

It corresponds to the test run of the program.

Let's actually execute the program and check its function.

For this task, we will use the debug perspective.

In the debug perspective, the executable file is executed under the supervision of the debugger (gdb).

The functions of the debugger such as execution tracking and stopping in association with the source file, variable monitoring, etc. can all be monitored on the eclipse GUI.

If you can confirm that there is no problem with run-time debugging, you will enter the final stage of development.

Going back to the C/C++ perspective, this time recompile the same source file(s) with the performance-oriented compile option and generate a performance-oriented executable file different from the previous one.

This performance-oriented executable is called the executable file for release.

All development is completed by obtaining the executable file for this release.

The generated release executable can be executed (normally from the command line) without eclipse.

(The debug executable can also be executed from the command line. If you do not need to pursue performance, you may not create the release executable but use the debug executable in time.)

**Exercise**

**05-01 -01-ex1**: Debug and release executables give the same results (with a few exceptions) on running. So why bother trying to create a release executable again in replace of the debug executable?

**05-01-01-ex2**: Investigate how to run the debugger on the command line without eclipse.

**05.01.02. Project**

A group of work (work to obtain one executable file) from the start of program creation to the generation of the release executable file collectively in eclipse is called a project.

(In this class, we explain eclipse specifically for program development in C language. We do not mention the generic definition of a project.)

If you want to create a different executable file, prepare another project.

In this class, one executable file is created in one section, so a new project will be prepared for each section.

In that sense, the entire class will cover dozens of projects.

Don't worry, eclipse can handle multiple projects at the same time.

**Exercise**

**05-01-02-ex1**: In programming by file division, there are multiple C program source files in one project. How many times does the description of the main() function appear in all the source files? And tell the reason.

**05-01-02-ex2**: Where do you specify program development in C/C++ language when using eclipse?

(It will appear later in this chapter)

**05-01-02-ex3**: What kind of projects are available in Eclipse other than C/C++ language programming? Examine and try to explain each.

(It is a difficult problem for ordinary students.)

**05.01.03. Organizing a workspace**

A workspace is a unit for multiple projects.

(This is also the name of the folder for multiple projects holding the projects as sub-folders.)

Normally, projects related to each other are handled together in one workspace.

In this class, we do not share sources between projects, so there is no problem even if each project has a separate workspace, but we will use one workspace throughout the semester.

Please note that eclipse can handle only one workspace at a time during its use.

(For example, if there is a course that uses eclipse other than this course, it is better to prepare another workspace there and switch between them at the beginning of each class.)

**Exercise**

**05-01-03-ex1**: Check if the workspaces already exist under your "Home" folder in U-Tsukuba Ubuntu system. If so, indicate the purpose for which each workspace is used.

### **05.02. Arrangement of folders and files**

If the program written on eclipse works in cooperation with other files, it is necessary to know where eclipse arranges source files and executable files.

**05.02.01. Workspace**

While working in a workspace, all files handled on eclipse are placed under the workspace folder.

(Of course, there are exceptions to everything, so you can run eclipse with files in a location other than the workspace folder.)

(This class will not handle such cases. If you are interested, do your best. Please learn by yourself)

The workspace is usually placed just under your home folder on Linux by default.

Let’s use the workspace named “eclipse-ide”.

The workspace name and workspace folder name should be the same.

**Exercise**

**05-02-01-ex1**: Please indicate the absolute path of your "home" folder on your Linux environment.

**05-02-01-ex2**: Check the usage of your "home" directory. Show how to actually investigate by yourself, and answer with the result of implementing that method.

(It will be displayed as a message when you log in, but it will not be accepted as an answer by itself.)

**05.02.02. Project**

The first thing to do when you want to start a project is to specify its name.

Since the project name becomes the "project folder name" as it is and also the "executable file name", it is safer to specify the project name using only alphanumeric characters that can be used as the file name and the folder name.

Once the project name is set, eclipse will create a folder with the project name directly under the workspace folder.

Hereafter, this folder will be called the project folder.

Note that if you change the name of a folder that already exists under the workspace folder to a new project name, the set of files in that folder will be registered in the project.

(Around here, the behavior of eclipse is different from the old days. There may have been various things among the developers of eclipse …)

**05.02.03. Source file**

In this class, any source files belong to one of the projects.

In this class, please place the source file(s) directly under the project folder.

(There is also a style of preparing a source folder under the project folder and placing the source file(s) under it.)

(This is a preference = cultural difference.)

It is possible to bring in the source files from the outside after the project starts.

Place the source file you want to bring under the project folder, right-click on the project > "Refresh" to include it in the project tree.

Please note that if you "update" in this way, all the files placed directly under the project folder will be imported into the project even if they are not the source files.

(It is against the eclipse way to put files in the project folder that you don't want to include in the project tree.)

**Exercise**

**05-02-03-ex1**: From the standing point of (the developer of) eclipse, what should be the trigger to automatically start the "update" process?

**05.02.04. The executable file and the current directory**

At runtime, the debug executable file is generated as "project-folder/Debug/project-name".

(Note that the "project-folder" is the same as the "project-name" here.)

The release executable file is generated as "project-folder/Release/project-name".

When the debug executable file and the release executable file are executed from eclipse, the project folder is used as the current folder.

Please be careful about this when you refer to the file etc. from the inside of the program when debugging or executing on eclipse.

When manually executing a program on bash, it is safer to move to the project folder and then execute it until you get used to it.

**Exercise**

**05-02-04-ex1**: Why doesn't eclipse try to run with the directory containing the executable as the current directory? There are good reasons for this. Please state the reason.

(The hint is that eclipse assumes that the developer will go through a two-step process of Debug and Release.)

**05.02.05. Others**

Even if you put some files just under the workspace folder, eclipse will not recognize them.

Even if you create another folder for the siblings of the project folder, eclipse does not care. However, that folder name will no longer be available as a new project name in this workspace.

(If you use this skillfully, you can prepare a "warehouse" directory invisible to eclipse.)

In rare cases, eclipse may behave strangely while you work.

Eclipse creates a hidden folder called “.eclipse” under the home folder to manage the work.

If you want to recover by yourself, you can try deleting this folder from the bash on a terminal.

However, all settings will be lost and the default will be restored.

Source files and project information are saved in the workspace folder, so even if you do this work, they will not disappear.

### **05.03. First-time eclipse**

**05.03.01. Start eclipse and set workspace**

Open the terminal and type eclipse\_cpp from bash to start it.

You should be asked which workspace you use, so set **workspace-ide**.

(If you are not asked, after starting, please switch with Menu: File > Switch workspace)

If the "Welcome" tab is displayed at the beginning, delete that tab.

Then, it becomes the C/C++ perspective which is the basis of eclipse.

(You can check if it is in the C/C++ perspective by gently hovering the mouse pointer over the icon on the far right and it will be displayed as "C/C++")

**05.03.02. The initial setup of eclipse**

Windowpanes (Small sub-windows) in a perspective can be freely laid out. You can change it. If you want to return to the original default layout, run Menu: Windows > Perspective > Reset Perspective.

(Resetting the perspective only corrects the arrangement of the windowpanes, so there is no change in the content.)

Here, let's make the overall settings related to C programming.

The line number should be displayed for the editor.

It seems that it is enabled by default in ver.2020-03. If not, follow this step.

(This is a function commonly found in text editors.)

|  |  |
| --- | --- |
| 05-03-02-A | Displaying line numbers |
| 1 | Menu: Window > Preference |
| 2 | [Prefernce]  (Hereafter, when [Foo] is mentioned, the window called Foo should pop up.)  (Left): General [Unfold] > Editors [Unfolod] > Text Editors [Select]  (Right): Text Editors > Show line numbers > OptIn > Apply and Close |

There is no need to pay particular attention to the close eclipse, but please note that it may take some time to record the work contents.

Menu: File > Exit is the most elegant method to make ends.

### **05.04. First project - Hello Eclipse -**

**05.04.01. Determining the project outline**

Let's set up a project in the form of writing a program directly.

First, consider the project name.

In this class, the rule is "2-digit chapter number - 2-digit section number".

(The reason why the section heading is set to 05.04. instead of 5.4. Is clear here …)

(This kind of rule is called a naming schema.)

(The naming schema is used for large-scale programming. This is a technique for avoiding mistakes and troubles that you may make in the future.)

(You don't have to follow it, but if you do, you may be more likely to survive later.)

Project name: 05-04 -HelloEclipse

Source filename: 05-04-HelloEclipse-Main.c

OK, let's start with these from the next section.

**05.04.02. Register a new project**

|  |  |
| --- | --- |
| 05-04-02-A | Register a new project |
| 1 | Menu: File> New> C/C++ Project |
| 2 | [New C/C++ Project] (Left) All  (Left) All  (Right) C Managed Build  > Next |
| 3 | [C Project]  Project name: 05-04-HelloEclipse  (Please make sure that you have checked "Use default location")  (The grayed-out location should be .../workspace-ide/05-04-HelloEclipse)  Project type: Executable [Unfold]> Empty Project  Toolchain: Linux GCC  > Next |
| 4 | [Select Configurations]  Make sure Debug and Release are OptIn  > Finish |

By finishing the work, you should see a project tree of the project "05-04-Hello Eclipse" in the project view on the left.

Make sure that the icon (indicating the perspective status) at the top right corner is "C/C++".

This provided a project with two settings (Debug and Release).

Click the triangle icon of the project tree at the beginning to expand the project tree.

**05.04.03. Creating a new source file**

|  |  |
| --- | --- |
| 05-04-03-A | Registering a source file |
| 1 | After selecting 05-04-Hello Eclipse in the project view  Menu: File > New > Source file |
| 2 | [New Source File]  Source file: 05 -04-HelloEclipse.c  > Finish |

05-04-HelloEclipse.c is registered in the project tree, and 05-04-HelloEclipse.c is in the editing state in the central editor.

(I think eclipse has already written comment at the beginning of the file)

(This is also a part of the culture in a sense)

After that, just write the C language program source with the central editor.

Try to save the source from time to time with Ctrl-S.

Now, let's enter the program below.

There are some very important keyboard shortcuts in eclipse.

**Ctrl + z** : Cancels the previous action. (Undo)

**Ctrl + y** : Redo the undo-ed operation. (Redo)

**Alt + /** : Present "correct" input candidates at this point. (Content Assist)

**Ctrl + s** : Saving a file

Especially the third is one of the three major merits of using eclipse. Try Alt + / as many as possible.

(For example, if you enter "#inc" and Alt + / ...)

(For example, if you try to enter stdio.h and Alt + / with "std" ...)

If a syntactic defect is found during editing, an icon on the left side of the editor will point out what might be wrong.

This is the second of the three major advantages of using eclipse; static syntax checking.

(The last one is a tracker for online debugging.)

(By the way, I'm just calling these "three major functions" just from my viewpoint, so don't ask Google teachers for that.)

(Eclipse of 2020 version seems that the static syntax check may fail if the file is not saved. You should type Ctrl + s frequently.)

Type the sample program below with the keyboard without "copy and paste".

(Because you won't appreciate these assist functions of eclipse with copy and paste.)

Eclipse also handles formatting, so indentation is automatically performed every time a line break occurs.

(It is really possible to change the indentation style.)

(There are many indentation styles. It's a culture. It's the same world as flower arrangement and tea ceremony.)

|  |
| --- |
| 05-04-HelloEclipse.c |
| #include <stdio.h>  int main(int argc, char \*\*argv){  int i = 2;  int j = 4;  printf("Hello, Eclipse.\n");  i = i \* i;  printf("Second variable = %d\n", j);  printf("First variable = %d\n", i);  return 0;  } |

(nothing behind the printf() function In many cases, but fflush(NULL); is required in rare cases.)

(In eclipse completion, the second argument of the main function is \*\*argv instead of \*argv [].)

(Both are quite correct as C grammar, so it's a matter of faith which one to write.)

(I prefer to use the \*argv[] expression, but eclipse suggests \*\*argv there.)

Get in the habit of saving with Ctrl + s frequently while typing.

(Frequent saving increases the chances of surviving trouble.)

**Exercise**

**05-04-03-ex1**: Explain the type of argv variable when you write "char \*\*argv".

**05-04-03-ex2**: Explain the type of argv variable when you write "char \*arg[]".

**05-04-03-ex3**: "char \*\*argv" and "char \*argv[]" can be regarded as equivalent. Explain carefully why this is the case.

**05.04.04. Creating an executable file for coding and debugging**

The following introduces the functions of eclipse that are convenient for programming.

Outline

When you click the components that are arranged in the outline view of the right, it'll take you to the beginning of the corresponding portion in the editor.

(It doesn't make much sense because there is only the main function now, but it is convenient if the source file becomes long and has many functions.)

Folding/Unfolding blocks

A small "-" is displayed on the first line of the function.

If you click this, the entire function body will be folded.

This feature comes in handy when the source is getting longer.

Refactoring

Refactoring is a batch conversion of variable names (function names are also possible).

For example, in the above source, if you right-click the variable i in the editor and select Refactor > Rename, and enter val1 in the small window that pops up, all the parts corresponding to the variable i in the source will be replaced.

Change the variable j to the variable val2 in the same way.

Since giving appropriate variable names and function names is an important method for programming, this function will also come useful.

No syntax errors in the program?

When the errors disappear, it is finally time to create an executable file for debugging.

Normally, when creating an executable file for debugging, you should tell the compiler and linker various your requests.

However, in the eclipse used here, the general (elementary) requests have already been filled in, so all you have to do is compile and execute.

(Therefore, on the contrary, it becomes difficult to understand what are the common requests and what will be done inside, and black boxing will progress.)

(That's why there was the previous chapter 04. in the class.)

In eclipse, an action to reach the goal of the project is called "building".

Here, the goal is to do compiling and linking so that we can obtain an executable file from the C source files.

Since the source file(s) are stored in the project, you can start "building a project".

There are other ways to build, one for debugging now and one for release later.

In that sense, the work to be done from now on is to "build a project for debugging" and obtain an executable file.

Before creating the debug executable file, save all the files.

In the 2020 environment, it may be better to cancel the automatic build option.

If Menu: Project > Build Automatically is checked, uncheck it.

Are you ready? Let's execute it.

|  |  |
| --- | --- |
| 05-04-04-A | Creating an executable file (of the current target project) |
| 1 | Menu: Project > Build Project |

(If Build Project cannot be selected, the debug executable file has already been updated.)

(You will visit here from Section 05.04.13 in the future. In that case, you are about to create a "release" executable file.)

(That is, this procedure creates a new executable file according to the specified build settings in the target project.)

If the executable file is created successfully, it is displayed in the Console windowpane that the executable file has been created.

(It's displayed in the Console windowpane that eclipse is running behind you on your behalf.)

(In fact, if you type the same string show in the Console windowpane on bash, you can do exactly the same thing.)

If you get an error, it will be displayed here and there in the source file, and at the same time the reasons will be listed in the lower Problem windowpane.

Go back and debug.

**Exercise**

**05-04-04-ex1**: Explain the contents of the Console windowpane line by line.

(You can skip the “make” command line because you do not learn what “make” commands is.)

**05-04-04-ex2**: Show the location of the created executable file with the path and file name.

**05-04-04-ex3**: Try to actually execute the executable file on the command line.

**05.04.05. Runtime debugging settings**

In this section, you will learn how to debug at runtime.

(Although such a short program is not very valuable for run-time debugging ...)

When starting a new debug execution for a project with eclipse (more accurately, an executable file), you need to decide the “debug configuration” before starting.

(It is not necessary to perform "Debug configuration" again at the second run of the run-time debugging.)

|  |  |
| --- | --- |
| 05-04-05-A | Debug configuration (and start of run-time debugging) |
| 1 | Make sure that the project you want to run runtime-debug is selected. |
| 2 | Menu: Run > Debug Configurations |
| 3 | [Debug Configurations]  (Left): Double-click “C/C++ Application”  (This make eclipse find a new setting that can be likely to set up as a new debug configuration)  (Left): If a new component (debug configuration) with the same name as the current project name appears one level below the “C/C++ Application”, click it (here “05-04-HelloEclipse Debug”)  (if it does not appear or something else happens, wait a few tens of seconds and double-click it again.)  (If another one appears, right-click and select Delete to delete it.)  (Right): Name: This is the name given to this configuration. Temporarily, "project name Debug" is suggested, but if possible, use your own name. Here, it is “05-04-HelloEclipse Debug Go”.  (You cannot give the same name as the past configuration. If it happens, you cannot select the last Apply.)  (Right): C/C++ Application: automatically says "Debug/05-04-HelloEclipse". It should be automatically included. This is the target executable file.  (If this field is blank, it means that the executable file cannot be prepared for some reason.)  (Did you create the executable file for debugging actually? If it is blank even though you created it, click the Search Project button. Find the executable file manually. Select Debug on Qualifier.)  (Right): When all are set, click “Apply” to save the settings. |
| 5 | [Debug Configurations]  Now that all the settings have been completed, click “Debug” at the bottom right to start run-time debugging.  (Eclipse allows you to run multiple debugs at the same time, but don't do that at first as it can be confusing.)  (By clicking “Debug”, you start running a single debug run instance under this debug configuration.) |

Finally, we are moving to run-time debugging, so eclipse is moving to the debug perspective.

Before moving to the debug perspective, the Confirm Perspective Switch window may appear.

You can press "Yes" every time, but if you don't want to, you can check "Remember my decision (always use this setting)" here.

The layout of the panes such as the view and editor is going to be changed, and the Debug windowpane appears in front of the Project windowpane on the left, and the perspective icon display on the upper right shows a bug-shaped Debug, so you recognize it is switched to the debug perspective.

**05.04.06. Debugging at runtime**

When you start runtime debug, it will be paused at the first line of the main() function.

(A small arrow should be displayed on the corresponding line in the editor and the line should be colored)

.(The line of the arrow has not been executed yet.)

Also, in the Debug windowpane, a debug tree of the debug instance "05-04-HelloEclipse Debug Go" (the name of the debug configuration you gave in section 05.04.05) should be shown.

(Eclipse can handle multiple debugs at the same time, but it is confusing at first, so make sure that multiple debug instances are not lined up in the Debug windowpane.)

The methods of run-time debugging are two ways.

(1) Step execution

The source program is executed line by line. You can know exactly what is happening, but the longer the program, the longer it will take to run.

(2) Through execution

The program is executed in the same way as the command line execution. However, if the location or conditions specified by the user are met, it will be paused there. The location to pose is called a breakpoint. This is a convenient method when you know the operation of the program to some extent.

Step execution and through execution can be switched at any time. For example, it is possible to set a breakpoint on a certain line, execute it through to that point, and then execute it in steps after pausing.

In the subsequent operations, approaches starting from Menu are shown, but icons and shortcut keys are assigned to frequently used operations. Please try to find it yourself.

For icons, if you hover the mouse cursor, an explanation will pop up, so please refer to that explanation.

(In eclipse, if you gently hover the mouse cursor over the icon, the description of the icon will be displayed in the Hover for about 1 second.)

**Exercise**

**05-04-06-ex1**: Multiple run-time debugs can be executed. Let's think about the situation where multiple, simultaneous run-time debugging is useful.

(Most of the students will graduate without realizing this issue)

**05.04.07. Step execution (step-in, step-over)**

First, let's perform step execution.

Step execution is further divided into two major types and one minor type depending on how you proceed.

Step-in :

Advance the program one line. If that one line is a user-defined function call, move into that function.

Step-over :

Advance the program one line. If that one line is a user-defined function call, it is executed through the function call and reaches the next line just after the function call is completed.

Step-return :

Return to the point where you stepped in (executes the current function to the end and returns to the position immediately after the current function was called). A counter action of the step-in.

In short, if you want to check the contents of a function, step in, if you are no longer interested in the operation of the function and you don't want to check the internal operation anymore, step over.

At the beginning, you may prefer using step-in and step-return.

If you step in too much, you will be warned (when you step into the external library function, you will be told that there is no source code, so there is no way to display the position in the source), so in that case, do step-return.

|  |  |
| --- | --- |
| 05-04-07-A | How to step in |
| 1 | Menu: Run > Step Into |

If you continue stepping in, you may get a message that the source code position cannot be displayed. In that case, execute step return.

(Specifications changed in the system after 2017)

(In the system before 2016, when step-in execution was impossible, it was a specification that automatically switches to step-over)

(It is not that which is bad, It is a difference in specifications.)

(Students are not very happy with the specifications after 2017, but what kind of people would you appreciate? Do you understand?)

(The messages and options on the warning window may give you a hint to think of it)

|  |  |
| --- | --- |
| 05-04-07-B | How to make step over |
| 1 | Menu: Run > Step Over |

On step-in or step-over, proceed as you go, the arrows in the editor window will advance, and when a variable changes, the variable in the Variable windowpane will be highlighted to let you know.

**05.04.08. Debug re-execution**

When you step in or step over to the end, termination is displayed at the top level of the debug instance in the Debug windowpane.

When that happens, this debug execution is complete.

It's easy to run the last debug again.

You can start debugging with the same configuration as the last time by the following method.

|  |  |
| --- | --- |
| 05-04-08-A | Debugging (re) start (re-execute with the same debug configuration as the previous debug configuration) |
| 1 | Menu: Run > Debug |

This will bring you back to section 05.04.06.

(When Debug starts, debug instances that have already been terminated are automatically deleted.)

[Note] If the above operation is performed while the previous debug execution has not been completed, the previous debug execution remains paused. Please note that in this state, another (second) debug execution will be started.

(You'll notice that there are two debug instances in the Debug windowpane.)

(As you get used to it, this can be used, but it's complicated now, so keep one debug instance at a time.)

**05.04.09. Through execution (Run, Break point)**

Now let's do it by "through" execution.

Through execution is also performed at the debug perspective.

First of all, do it "through" without thinking about anything.

|  |  |
| --- | --- |
| 05-04-09-A | Through execution |
| 1 | RunMenu: Run > Run |

It is executed at an unstoppable speed, and the standard output is displayed in the Console windowpane.

Another method is to start debugging in the same way as 05-04-08A and then resume it to the end without stepping in and stepping over.

|  |  |
| --- | --- |
| 05-04-09-B | Start and continue debugging |
| 1 | Menu: Run > Debug (start) |
| 2 | Menu: Run > Resume (continue) |

Next, let's set a breakpoint in through execution.

Consider pausing the program just before executing the first printf().

|  |  |
| --- | --- |
| 05-04-09-C | Breakpoint setting |
| 1 | It is the same for both a / b below.  a. Click the line in the editor pane, then Menu: Run > Toggle Breakpoint  b. Double-click the left edge of the line in the editor windowpane. |

If successful, a small check mark will appear to the left of the line in the editor.

It is also displayed in the breakpoint windowpane.

Let's start through execution in the debug start state (pause state at the first line of the main() function) (05-04-09-B / 1).

Then, this time, resume (05-04-09-B / 2) to reach the breakpoint setting line.

If you resume again (05-04-09-B / 2) here, the execution will proceed to the end and end.

(If you want to try it many times, carry out 05-04-09-B)

You can start step-in or step-over with the paused state.

You can also resume after stepping.

There are two ways to release breakpoints.

|  |  |
| --- | --- |
| 05-04-09-D | Breakpoint Cancellation |
| 1 | Breakpoint Permanent Release: You can do the same thing as setting a breakpoint again.  Temporarily uncheck a breakpoint: In the breakpoint windowpane, uncheck the checkbox for that breakpoint. |

**05.04.10. Before returning to the C/C++ perspective --End of debug execution --**

After confirming the operation by executing run-time debug, let's end (delete) the debug configuration you are currently working on.

|  |  |
| --- | --- |
| 05-04-10-A | End of debugging |
| 1 | Select the debug instance you want to end from the debug windowpane |
| 2 | Menu: Run > Terminate |
| 3 | Right-click on the debug instance labeled “Terminated” > Remove All Terminated |

You do not have to do step 3 of 05-04-10-A above at eclipse of 2020.

(In the 2020 version, when Debug starts, the debug instances in the Terminated state are automatically deleted before the start.)

In eclipse, you can switch from the debug perspective to the C/C++ perspective without finishing debugging. You can switch perspectives at any time.

However, in this lesson, make it a habit to "end all debug executions when switching from the debug perspective to the C/C++ perspective".

(Otherwise, the next time you return from the C/C++ perspective to the debug perspective, you'll have to reconstruct the situation you had in your mind when you left last time, which is easy for Eclipse. But it is not easy for you probably.)

**05.04.11. Switching perspectives**

Let's return from the debug perspective to the C/C++ perspective.

|  |  |
| --- | --- |
| 05-04-11-A | Switching perspectives |
| 1 | Menu: Window > Perspective > Open Perspective > C/C++ |

You can return to the debug perspective in a similar way.

**05.04.12. Checking the operation of the debug executable file**

You can also simply execute the debug executable file.

(It means simply executing without debugging.)

There are two ways to execute the debug executable file.

The first method is to execute it on eclipse.

|  |  |
| --- | --- |
| 05-04-12-A | Executable file execution |
| 1 | Confirm that you are in the C/C++ perspective |
| 2 | Menu: Run > Run |

In this case, the perspective is not switched and the standard output result from the program is displayed in the console windowpane at the bottom center.

If you use this procedure, Eclipse will check the file dependencies before executing the executable file, and will start generating the executable file (“build project”) if necessary. After that, execute the newly prepared executable file.

The second method is command line execution.

The following is the work in the terminal, away from eclipse.

$ cd

$ cd workspace-ipd/05-04-HelloEclipse

$ ls

$ ls Debug

$ ./Debug/05-04-HelloEclipse

**Exercise**

**05-04-12-ex1**: Where do you go when you just "cd" on the bash in the terminal window?

**05.04.13. Generation of release executable file**

From here, we aim to generate a release executable file.

(Since the executable file for debugging involves various mechanisms, it has a structure that makes it very difficult to obtain the maximum speed.)

(For example, in athletics, debugging is a heart rate measuring device or sweat measuring device for athletes. It's like having the athletes run while wearing them. Of course, they can run, but we should remove all the devices from them to let them run at full speed.)

(By the way, the execution speed of the debug executable file created with gcc is not so bad. Actually, the performance degradation is not a big deal. But the debug execution of Microsoft's Visual Studio is in a terrible state. If the gcc's measuring device is something like a watch for athletes, the Visual Studio's measuring device is like a full measurement kit in a backpack with a heavy battery. Well, the analysis can be done in great detail, but, in return, ...)

(Which style do you think is better? After all, it depends on what you like to see.)

In order to create an executable file for release, it is necessary to switch the policy of creating the executable file to focus on performance instead of the conventional focus on debugging.

Since the goal is to create a different type of executable file, you will have to build differently than before.

The build settings for debugging were set in section 05.04.05. At that time, the build settings for release were also set.

(The people who develop eclipse are smart, so when you want to speed up, they've already prepared it in eclipse instead of you, ahead of you, without asking you what to do.)

Now you will use the build settings for the release.

Switch the guideline for creating (building) the executable file of the project from debugging to releasing.

For this, we will change the active build.

Make sure the perspective is C/C++.

|  |  |
| --- | --- |
| 05-04-13-A | Change of active build |
| 1 | Confirm that you are on the C/C++ perspective. |
| 2 | In the project windowpane, check that the project for which you want to generate the release executable file is selected. |
| 3 | Menu: Project > Build Configurations > Set Active > Release |

After making the changes, under the guidelines for creating a new (release) executable file, do the same work as you did in "Debug Build".

In other words, create (build) the release executable file.

The procedure is the same as in section 05.04.04.

However, now the build guidelines are "for release" instead of "for debugging".

When the build starts, you will see the work in the console windowpane.

When the build is finished, there should be one new node called Release along with Debug in the second layer of the project view.

(If the build does not finish successfully and the executable file cannot be created, debug it.)

From here, set up the "(Release) Executable Configuration".

Then, Execute "(the release) executable file " on eclipse.

(Section 05.04.05 was the configuration at the time of run-time debugging the executable file for debugging)

(This time, the configuration at the time of executing the executable file for serious run without test, without debug, but for max speed.)

(Actually, If you are thinking of executing the release only on the terminal without executing it on eclipse, this "execution configuration (for release)” can be skipped.)

|  |  |
| --- | --- |
| 05-04-13-B | Execution configuration (For release) (, and run) |
| 1 | Make sure the project you want to run is selected in the project view |
| 2 | Menu: Run > Run Configurations |
| 3 | [Run Configurations]  (Left): Double-click C/C++ Application  (Now let eclipse search for an executable file that can be newly set.)  (Left): A new component (release configuration) with the same name as the current project name appears one level below the C/C++ Application. Click it (“05-04-Hello Eclipse Release” here).  (If it doesn't appear or another one appears, wait a few tens of seconds and double-click it again)  (If another one appears ， Right-click and select Delete to delete)  (Right): Name: is the name given to this configuration. Temporarily "Project Name Release" is suggested, but if possible, let's name it “05-04-HelloEclipse Release Go-Go”.  (You can't give it the same name as the past configuration. If it has the same name, you can't select the last Apply)  ( Right): C/C++ Application: It should automatically be filled with "Release/05-04-HelloEclipse". This is the target executable file.  (If this field is blank, It means that the executable file is not ready.)  (Did you create the executable file for release? If it is blank even though it is created, set the executable file manually from the Search Project button. As a qualifier you should choose Release.)  (Right): When all the settings are done, click Apply to save the settings. |
| 5 | [Run Configurations]  (Right): Now that all the settings are set, click Run at the bottom right to start execution. |

In release execution, there is nothing to do here during execution, so the perspective remains C/C++.

The execution result should be displayed in the Console windowpane.

The second and subsequent executions can be performed in the same procedure as in Section 05.04.12, unless the execution configuration is changed.

The release executable is used in the same way, except that it cannot be debugged at runtime and the directory in which the executable is stored is different from that for debugging.

Like the debug executable file, the release executable file can be executed from eclipse or from the command line.

To execute it again on eclipse, follow the procedure 05-04-12-A.

At this time, eclipse checks the file dependencies before executing the executable file, and starts generating the executable file if necessary. After that, it executes the newly prepared executable file.

For command line execution, replace "Debug" in the shell command shown in section 05.04.12. with "Release".

$ cd

$ cd workspace-ipd/05-04-HelloEclipse

$ ls

$ ./Release/05-04-HelloEclipselighter.

Unfortunately, you can't feel the speed difference with such a short program, but you can imagine the speed difference from their file sizes.

$ ./Debug/05-04-HelloEclipse

$ ./Release/05-04-HelloEclipse

$ ls -l ./Debug/05-04-HelloEclipse ./Release/05-04-HelloEclipse

**Exercise**

**05-04-13-ex1**: Is there a difference in the dynamic library between the debug executable and the release executable? Check the executable file created in this section.

**05-04-13-ex2**: The compile options for creating a release executable file are usually not the same options for debugging.

Find out what options are used for debugging and release in the eclipse environment currently used by the students, and explain their meanings one by one. (Unknown options can be left unknown, but in that case, specify how to check.)

**05-04-13-ex3**: Check the file sizes of the debug executable file and the release executable file. Try to explain why there is a difference in file size and why.

### **05.05. Bringing external source files into the project -- Hello ESYS again --**

**05.05.01. Project overview**

Here, we will learn how to import the program in section 01.02. as a new project in eclipse.

Project name: 01-02-HelloESYS-Better

Source file: 01-02-HelloESYS-Better.c

**05.05.02. Registering a new project**

This will be the second project in this workspace (workspace-ipd).

Create it in the same way as 05-04-02-A.

Then you should see two projects in the project view: "01-02-HelloESYS-Better" and "05-04-HelloEclipse".

Select “01-02-HelloESYS-Better” and click the triangle icon to expand and display the project tree.

**05.05.03. Importing source files from outside and debug/execute**

Copy the source files you want to import to the project folder (~/workspace-ipd/01-02-HelloESYS).

Save the original C source file as the name of 01-02-HelloESYS-Better.c.

Are you ready? Now, use "Refresh" in eclipse to register the source file in the project.

|  |  |
| --- | --- |
| 05-05-03-A | Registering a newly prepared file in the project folder |
| 1 | Menu: File> Refresh |

If successful, 01-02-HelloESYS-Better.c will be added to the project tree of 01-02-HelloESYS-Better.

Let's create an executable file for debugging as described in section 05.04.04.

To perform run-time debugging, configure run-time debugging as described in Section 05.04.05. Move to the debug perspective, and start run-time debugging.

To create an executable file for release, change the build method and build as described in Section 05.04.12.

**Exercise**

**05-05-03-ex1Add**: fflush() to each printf() in the program source of this section, and modify it so that the output of printf() is output on the console every time.

**05-05-03-ex2**: When adding fflush() of ex1 above, there may be a difference between adding to the right of printf() and under printf(). Find the difference if any.

### **05.06. Bringing in external source files to the project --Simple calculator split source files--**

**05.06.01. Project overview**

How to import the program used for the split source files in section 04.02 as a new project will be explained here.

Project name: 04-02-SC

Source files: 04-02-SC.h, 04-02-SC-BasicFunctions.c, 04-02-SC-ExtraFunctions.c, 04-02-SC-Main.c

**05.06. 02. Registering a new project**

This should be the third project in our workspace (workspace-ipd).

Create a new project with the project name decided in the previous section by following the procedure of 05-04-02-A.

When you have done, you should see three projects in the project view: "01-02-HelloESYS-Better", "04-02-SC", and "05-04-HelloEclipse".

Select 04-02-SC and click the icon at the beginning to expand and display the project tree.

**05.06.03. Importing the split source files from the outside and debugging/executing**

Copy the source files you want to import to the project folder (~/workspace-ipd/04-02-SC).

Here, 04-02-SC.h, 04-02-SC-BasicFunctions.c, 04-02-SC-ExtraFunctions.c, 04-02-SC-Main.c are the files we need to prepare.

Like the work in section 05.05.03., let's register the copied files in the project tree all at once.

If successful, the above 4 files will be added to the project tree of 04-02-SC.

Let's build the debug executable file.

(Eclipse looks in the split C source files, finds out where the main() function is, and automatically compiles and links them.)

(The progress of compile and link procedure is displayed in the console windowpane.)

For run-time debugging, configure run-time debugging as in section 05.04.05., Move to the debug perspective, and start run-time debugging.

During run-time debugging, execution continues across multiple source files, but the user does not have to worry because eclipse automatically switches the source files to be displayed.

There is a call to the user-defined library function in this program, so check the difference between step-in and step-over.

During run-time debugging, the program pauses at the fgets() function (of course) and waits for data from standard input.

Enter a calculation for one line in the console windowpane. After that, the debug execution will continue in the specified form.

To create an executable file for release, change the build method as described in Section 05.04.12.

Let's execute the release executable file both on Eclipse and on the command line.

Make sure that you can do the same thing you did in section 01.02 on the command line.

### **05.07. Bringing external source files into the project --Program to open files--**

**05.07.01. Project overview**

Here we learn how to import and execute the "Program to open files" in section 02.02. as a new project of eclipse.

Project name: 02-02-OpenFile-Full

Source file: 02-02-OpenFile-Full.c

**05.07.02. The registration of a new project**

The procedure is almost the same as 05-05-02-A in Section 05.05.02.

**05.07.03. Importing the source file from the outside**

The procedure is the same as in section 05.05.03.

**05.07.04. Debug Configuration with startup arguments (options)**

For the debug configuration of run-time debugging, If you exactly follow the procedure in section 05.04.05, you will fail on the execution. Let’s fail for the first time.

Name the debug configuration 02-02-OpenFile-Full Debug.

When you actually try this, because the file specification is not passed as an argument (option) at startup, the execution ends up being caught in a trap prepared in the user program.

So let's create another debug configuration with startup arguments.

|  |  |
| --- | --- |
| 05-07-04-A | Debugging configuration with arguments (and starting of runtime debugging) |
| 1 | Make sure the project (02-02-OpenFile-Full) that you want to debug at runtime is selected in the project view. |
| 2 | Menu : Run > Debug Configurations |
| 3 | [Debug Configurations]  (Left): Double-click “C/C++ Application”  (Left): If a new component (debug configuration) with the same name as the current project name appears one level below the “C/C++ Application”, click it (here “02-02-OpenFile-Full Debug”)  (Right): Name: This is the name given to this configuration. Here, set “02-02-OpenFile-Full Debug Arg”.  (Right): C/C++ Application: "Debug/02-02-OpenFile-Full" should be automatically suggested.  (Right): Program arguments under “(x)=Arguments”: Set the command-line options. Here, set “hello.txt”.  (You may find a display of “Working directory” where eclipse tells where it will execute the executable file.)  (Right): When all are set, click “Apply” to save the settings. |
| 4 | [Debug Cofigurations]  (Right): Now that all are set, click Debug at the bottom right to start run-time debugging. |

How is it?

You get caught in another trap you wrote.

This is, of course, because the file hello.txt does not exist.

Let's go back to the C/C++ perspective and create hello.txt.

hello.txt is supposed to be just a text file.

Let's write hello.txt in the project so that it can be edited with the editor of eclipse.

|  |  |
| --- | --- |
| 05-07-04-B | Adding a text file to a project |
| 1 | Menu: File > New > Other |
| 2 | [New]  General [Unfold ]> Untitled Text File > Finish  (An edit screen that looks like “Untitled 1” appears in the editor.) |
| 3 | Write something in hello.txt.  (02-02-OpenFile-Full.c does not need to have the contents in the file, so it does not matter whether the file is empty or something is written.) |
| 4 | Save by specifying the project and file name:  Menu: File > Save As  [Save as]  Specify “02-02-OpenFile-Full” for the parent folder.  Specify “hello.txt” for the file name.  > OK |

This will make a file at ~/workspace-ipd/02-02-OpenFile-Full/hello.txt.

hello.txt should also be added to the project tree of 02-02-OpenFile-Full in the project pane on the left.

Let's go to the debug perspective again and debug at runtime.

The 02-02-OpenFile-Full project now has two types of run-time debugging environments (“02-02-OpenFile-Full Debug” and “02-02-OpenFile-Full Debug Arg”).

Now, select the latter “02-02-OpenFile-Full Debug Arg” by the following work and start debugging.

|  |  |
| --- | --- |
| 05-07-04-C | Changing the debug configuration and starting run-time debugging |
| 1 | Menu: Run > Debug History > 02-02-OpenFile-Full Debug Arg  (The build starts, and pause the execution at the first line of the main function) |
| 2 | Menu: Run > Resume |

This time, the program should proceed normally to the end.

If "All set. Bye!" Is displayed at the end in the console windowpane, it is proof that it worked as expected.

**Exercise**

**05-07-04-ex1**: Add fflush() to each printf() in the program source of this section, and modify it so that the output of printf() is output on the console every time.

**05-07-04-ex2**: When adding fflush() of ex1 above, the difference in behavior during step-in execution when adding to the right of printf() and adding under printf () happens. Describe the difference.

**05-07-04-ex3**: In the program source of this section, define your own function that performs printf() and fflush() together, and replace printf() by calling it. At that time, make it possible to output the line number of the program source on the reading side.

It is better to prepare three types of user-defined functions according to the difference in how to use printf().

With a little thought, user-defined functions can be reduced to two types.

**05-07-04-ex4**: In the definition of a C language function, the number of arguments should be originally fixed, but there are some functions such as the printf() function that can obviously change the number of arguments at the time of calling. If you want to achieve the same thing with a user-defined function, explain how to describe it with an example.

**05-07-04-ex5**: Using the above knowledge of ex4, try to prepare only one user-defined function when creating an ex3 program. (It will be a rather harder question.)

**05.07.05. eclipse: How to read array variables in the debug perspective and the structure of argv called an array of character strings**

When you start debugging at “02-02-OpenFile-Full-Debug Arg”, you will see three variables, argc, argv, and filetoopen in the Variables windowpane (upper right).

It's nice that the value of argc is 2.

What is confusing (because I gave hello.txt as a runtime argument) is argv, which is an array of pointers, and the number of elements is 2 (the number of elements is known by the value of argc).

(Since it is a pointer, the address is included as a value. Since it starts with 0x, it is in hexadecimal notation.)

However, if you look at argv alone, you cannot tell how many arrays will come.

Since there is no help for it, even if a pointer array comes, eclipse initially just displays only the first address of the pointer array as a value.

In this way, when using a pointer variable as an array, eclipse does not display it in the form of an array.

(It is due to the C language specifications. If you refer to it unexpectedly without knowing how many elements are there, a buffer overrun will be waiting for us immediately ...)

By the way, at this moment at runtime, we have a pointer with argv of 2 elements because it is shown in argc. And we know it's an array.

Therefore, give that information to eclipse and ask them to change the display method.

|  |  |
| --- | --- |
| 05-07-05-A | Array display of pointer variables |
| 1 | Right-click argv in the variable pane > Display As Array |
| 2 | [Display As Array]  Start Index: 0  Length: 2  > OK |

This makes the argv tree display 2 nodes in the 2nd layer.

If you click argv[0] and argv[1], you can see the start address of each character string and the string itself.

If you are OK, try counting the number of characters in the string contained in argv [0].

You can see that the address of argv[0] + the number of characters of argv [0] + 1 ('\ 0') should be the address of argv[1].

(The example in my environment where I am writing the manuscript was 0x7fffffffe377 + 85 + 1 = 0x7fffffffe3cd)

(In my case, the 0th argument is "/home/kameda.yoshinari.ft/workspace-ipd/02-02-OpenFile- It was Full/Debug/02-02-OpenFile-Full"; 85 letter)

(It's a hexadecimal number, but you should do the calculation in any format)

(The layout of the memory space is slightly different by compilers/OS, so if it doesn't look like this in your environment, you don't have to worry about it.)

(But most of the time, string arrays are packed and arranged, so this calculation will probably work ...)

Note that we do not touch on filetoopen structure variables.

When you step through the execution of fopen(), many changes occur under the filetoopen variable at the moment when fopen() ends.

If you expand the filetoopen tree, you will see that it has actually a complex structure. It is because it is responsible to hold any status changes for the exchange of data between the OS and the user program that can not be seen usually from anyone.

So this is the typical example of information hiding.

**Exercise**

**05-07-05-ex1**: Create a program that opens two text files (for reading) based on the program in section 05.07.04.

* If neither of the specified files can be opened or the file name is not correct, display which file could not be opened along with the reason.
* During execution, read one line from each file and display it. (1st line of the 1st file, 1st line of the 2nd file, 2nd line of the 1st file, ...)
* When either is finished, output the rest of the remaining file in order.
* Close both files at the end of the program.
* Prepare two text files by yourself (up to 10 lines).

**05-07-05-ex2**: Show the contents of the above filetoopen structure variable using the function of eclipse, and try to explain each member.

### **05.08. Supplementary information**

In the Linux environment of the computer room on the 5th floor of the Engineering System Building, the specified operation may not be performed occasionally when using eclipse.

It seems to be occasionally seen at compile time and through debug execution.

In that case, re-execute the work from the beginning.

At that time, please wait for several tens of seconds before doing it.

(Maybe it's a symptom of the way of mounting your home folder, but I haven't confirmed it.)

Students who like to debug with printf() in eclipse in our Linux environment are encouraged to add fflush(NULL) immediately after each printf().

If you do not add it, the result of printf() might not be displayed on the console immediately when the step execution is invoked.

(Such a problem does not occur in a normal environment.)

(Since fflush() does not cause any particular harm, there is no problem to add fflush() even when you do not face such situations)

**Exercise**

**05-08-ex1**: Examine and explain the function of the fflush() function. Specify the citation source. Also, describe how fflush() makes a difference in using printf(), and explain why you can't feel it in practice in normal cases.