## **4. Programming by divided files**

When you start to reuse program parts by using subroutines, some parts will continue to be used without being rewritten again.

(To be more precise, we use subroutines to create such situations and make things easier.)

It is safer not to inadvertently edit program parts that will not be rewritten in the future.

(This is important for improving program development efficiency.)

In some cases, it is conceivable that multiple programmers share different functions for writing.

(The basic form is to decide the person in charge for each function and write it in a separate file for each person in charge.)

For this purpose, programming by dividing the file is useful.

If you write the program by dividing it into files, an object file will be generated for each divided source file. This is the work of split compilation.

Collect the completed object files and finally create one executable file. This is a work called linking.

The programming style can be roughly divided into two styles depending on when to collect these multiple object files.

(1) Make object files from source files when needed and link them (normal linking work)

(2) Collect object files, create a library once, and write a program later using it. (linking work using a library)

In this section, you will learn both.

### **04.01. Source file division of C language program**

In C language, as a general rule, source file division is used to separate functions.

You can write multiple functions in one source file.

(The ultimate form of this is the sample program in section 03.04.)

You can write only one function in one source file.

There are no written rules, but it is better to put functions with similar functionalities in the same file.

After writing the program source divided into files, compile each source file and create an object file.

This is called split compilation. Separate compilation is performed for every source file.

For each split compilation, each source file must contain enough information to compile.

Therefore, at the beginning of each source file, the header file is included according to the function used in it.

**[Birth of header file]**

When the source file is divided, some functions that you do not know (do not understand) may appear which are not defined in the source file.

Let's look at the simple example below.

The operator\_add() function is used in the source file (split-main.c) that contains the main() function. There are no clues about its format or function in this file.

In split compilation, the "type" and "function body" of a function are considered separately.

The "type" of a function is defined by the name of the function, the number and type of arguments, and the return type of the function itself. This is called a function prototype.

The "type" that shows how to use the operator\_add() function is going to be shared everywhere. In other words, operator\_add() function in both split-main.c and split-sub.c must be used in the same expression.

(For function arguments, if the type and order are the same, the variable names do not have to be the same.)

| Original (full.c) | Splitted (split-main.c) | Aplitted (split-sub.c) |
| --- | --- | --- |
| #include <stdio.h>  float operation\_add(float v1, float v2) {  return v1 + v2;  }  int main(int argc, char \*argv[]){  float val1=14.2;  float val2=3.4;  printf("result = %f\n", operation\_add(val1, val2));  return 0;  } | #include <stdio.h>  float operation\_add(float v1, float v2) ;  int main(int argc, char \*argv[]){  float val1=14.2;  float val2=3.4;  printf("result = %f\n", operation\_add(val1, val2));  return 0;  } | float operation\_add(float v1, float v2) {  return v1 + v2;  } |

The header file is a collection of these function prototypes.

Here, function prototypes are collected in a file called 04-01-SplitHead.h.

Then, the split source will be as follows.

| Original (04-01-Full.c) | Split with a header file (04-01-SplitHead-Main.c) | Split with a header file (04-01-SplitHead-Sub.c) | The header file (04-01- SplitHead.h) |
| --- | --- | --- | --- |
| #include <stdio.h>  float operation\_add(float v1, float v2) {  return v1 + v2;  }  int main(int argc, char \*argv[]){  float val1=14.2;  float val2=3.4;  printf("result = %f\n", operation\_add(val1, val2));  return 0;  } | #include <stdio.h>  #include "04-1-SplitHead.h"  int main(int argc, char \*argv[]){  float val1=14.2;  float val2=3.4;  printf("result = %f\n", operation\_add(val1, val2));  return 0;  } | float operation\_add(float v1, float v2) {  return v1 + v2;  } | float operation\_add(float v1, float v2) ;  // Name of the option variables does not make sense, so just show types is sufficient like:  // float operation\_add (float, float);  // |

If you look at 04-01-SplitHead-Main.c, you can see that there are two ways to include the header file.

If the header file is enclosed in <>, the compiler will search for the specified header file according to the rules set in the compiler.

The header file of the library function corresponds to this (example: stdio.h, math.h, etc.)

On the other hand, if the header file is enclosed in "", the user explicitly has to tell which directory at compile time to find it.

The only exception is the current directory “.”, which is automatically checked to find the library.

Now 04-01-SplitHead-Main.c, 04-01-SplitHead-Sub.c, and 04-01-SplitHead.h are placed in the working directory for split compilation and linking.

$ gcc -Wall -c 04-01-SplitHead-Main.c (Making the object file 04-01-SplitHead-Main.o)

$ gcc -Wall -c 04-01-SplitHead-Sub.c (Making the object file 04-01-SplitHead-Sub.o)

$ gcc -Wall -o 04-01-Split-Go 04-01-SplitHead-Main.o 04-01-SplitHead-Sub.o (Linking three object files to make the executable file 04-01-Split-Go)

$. /04-01-Split-Go (Execute the file)

(Actually, the linker creates an executable file by adding some extra object files that are not explicitly specified here.)

(("Compiler" and "Linker" have completely different tasks. I think it would be easier to understand if they had different command names.)

**[Dynamic library]**

In fact, in modern executable environments, 04-01-Split-Go is not the perfect form to be executed.

The moment you try to run 04-01-Split-Go, some dynamic libraries are added to 04-01-Split-Go. As a result, the executable file is ready to be executed as an application program.

$ ldd 04-01-Split-Go

You can use the command ldd to find out which dynamic libraries will be added.

(There are many dynamic libraries required.)

(The dynamic library provided directly by the OS does not have a corresponding library file. Only the address is shown.)

(The right end is where it is placed in the actual memory space. How many bits does the address represent?)

**Exercise**

**04-01-ex1**: How many bits is the Ubuntu Linux you are using? Show it along with the survey method.

**04-01-ex2**: Investigate the number of bits of the address that Ubuntu Linux you are using can handle.

**04-01-ex3**: Investigate how many bytes of main memory can be logically handled by the Ubuntu Linux you are using.

### **04.02. Simple calculator / split compilation version**

Let's apply the same method as in section 04.01. to the simple calculator program in section 03.04.

03-04-SimpleCalculator.c (repost) (code/03-04/03-04-SimpleCalculator.c)

Follow the steps below to do it yourself.

(An example of the answer is posted at the end. If you look at it immediately, you will not get a learning effect, so do not look at it.)

Let's make the following policy regarding the source file division of the simple calculator program.

(1) A source file containing four functions corresponding to the four basic operations of addition, subtraction, multiplication, and division. If you write it once, you will never rewrite it.

(2) A source file containing functions corresponding to other new operations.

(3) A header file that summarizes the function prototypes of (1) and (2) above.

(4) A source file containing the main() function. The user interface is also prepared here.

The following is an actual division example. It doesn't have to be this way.

04-02-SC-BasicFunctions.c: 4 functions corresponding to the basic 4 operations of addition, subtraction, multiplication and division (code/04-02/04-02-SC-BasicFunctions.c)

04-02-SC-ExtraFunctions.c: Functions corresponding to new operations (code/04-02/04-02-SC-ExtraFunctions.c)

04-02-SC.h: Common header file (code/04-02/04-02-SC.h)

04 -02-SC-Main.c: main () function (code/04-02/04-02-SC-Main.c)

The following is a compilation example.

$ gcc -Wall -c 04-02-SC-BasicFunctions.c

$ gcc -Wall -c 04-02-SC-ExtraFunctions.c

$ gcc -Wall -c 04-02-SC-Main.c-Wall-

$ gcco 04-02-SC-Go 04-02-SC-Main.o 04-02-SC-BasicFunctions.o 04-02-SC-ExtraFunctions.o

$ ./04-02-SC-Go

**[Digression] Multiple “Include”s Prevention**

04-02-SC.h contains unfamiliar macro instructions.

The macro "#ifndef FOO" considers that the condition is met if the macro "FOO" is undefined, and put all the lines up to "#endif". “ifndef” means “if not defined”.

If the macro “FOO” is defined, all lines up to “#endif” will be ignored.

Here, a macro called \_04\_02\_SC\_H is prepared as an argument of ifndef. This is what I decided on my own. It's a weird name, so no one else will define a macro with the same name.

By doing this, even if #include "04-02-SC.h" exists twice in one source program, its contents will be expanded only once.

The same gimmick can be seen, for example, with the \_MATH\_H macro in /usr/include/math.h.

**Exercise**

**04-02-ex1**: If you do not prepare 04-02-SC.h, state what you will need in the other three files.

**04-02-ex2**: List two or more advantages of preparing 04-02-SC.h.

**04-02-ex3**: Extend the above simple calculator program as follows.

- Operator "exp". "A exp B" finds the numerical value AB. A and B may be floating-point numbers.

- Operator "log". "A log B" finds the numerical value of logAB. A is a positive floating-point number and B is a floating-point number greater than or equal to 0.

- If an invalid numerical value is given during the calculation, display that fact and do not perform the calculation.

- For the functions corresponding to these operators, describe their bodies in 04-02-SC-ExtraFunctions.c.

- Rewrite other files if necessary.

**04-02-ex4**: Archive the files using the zip command.

- It is assumed that you are in the same directory as the above "Compile example".

- On bash, execute "zip 04-02-ex4.zip 04-02-SC-\* .c".

**04-02-ex5**: Check the contents of the zip file.

- On bash, execute "unzip -l 04-02-ex4.zip".

(Note that if you use "unzip 04-02-ex4.zip", it will actually be expanded)

### **04.03. Simple calculator ・ Production and use of homemade library**

When programming with split compilation, there might be some source files that may be fixed.

It is a waste of time to create an object file for each of these source files at every compilation.

For example, 04-02-SC-BasicFunctions.c and 04-02-SC-ExtraFunctions.c will not be changed in the future.

When creating an executable file by modifying only 04-02-SC-Main.c, there is no need to recreate 04-02-SC-BasicFunctions.o and 04-02-SC-ExtraFunctions.o.

$ gcc -Wall -c 04-02-SC-Main.c

$ gcc -Wall -o 04-02-SC-Go 04-02-SC-Main.o 04-02-SC-BasicFunctions.o 04-02-SC-ExtraFunctions.o

$ ./04-02-SC-Go

This process is repeated every time 04-02-SC-Main.c is rewritten. It is troublesome to specify 04-02-SC-BasicFunctions.o and 04-02-SC-ExtraFunctions.o one by one at the time of linking.

(With an increasing number of user-defined functions on a simple calculator, it is hard to specify the object files to be linked. Any of you may forget to specify some files. Typos more likely)

Here, Let’s think of handling all of these "object files that correspond to the split source files that are no longer modified".

That is, the **library**.

**[Creating a library]**

Creating a library is surprisingly easy.

Suppose we are at the last state of section 4.2.

To create a library, use the command ar. Here, we will create a static library.

$ ar r libSC.a 04-02-SC-BasicFunctions.o 04-02-SC-ExtraFunctions.o

Let's confirm that we have done it.

$ ls (confirm that libSC.a is created)

$ ar t libSC.a (confirm that the contents of libSC.a are composed of two object files)

If libSC.a is successfully generated , 04-02-SC-BasicFunctions.c, 04-02-SC-ExtraFunctions.c, 04-02-SC-BasicFunctions.o, 04-02-SC-ExtraFunctions.o are not required anymore.

(You can delete them, but move them to another directory in case you need them later.)

($ Mkdir old\_src)

($ mv 04-02-SC-BasicFunctions.c 04) -02-SC-ExtraFunctions.c 04-02-SC-BasicFunctions.o 04-02-SC-ExtraFunctions.o old\_src)

From now on, all you need is 04-02-SC.h and libSC.a.

These two make up your own homemade library.

04-02-SC-Main.c describes a simple calculator using the homemade library. After that, create an executable file according to the following procedure.

$ gcc -Wall -c 04-02-SC-Main.c

$ gcc -Wall -o 04-02-SC-Go 04-02-SC-Main.o **-L. -lSC**

$ ./04-02-SC-Go

"-lSC" is the order of using the library called "libSC.a". The name of the static library must always start with "lib" and end with ".a".

“-L.” Means to refer to the working directory “.” when searching for libraries.

(Do not put a space between -l and SC. Do not put a space between -L and “.”.)

Suppose 04-02-SC-Main.c is the only file being edited. The above work can be simplified a little more.

(To be precise, make gcc compile and link together.)

(Gcc has both compiler and linker functions to enable this "collective request".)

$ Gcc -Wall -o 04-02-SC-Go 04-02-SC-Main.c -L. -lSC

You guys have created your own homemade library and become a programmer who uses it.

(Wow, congratulations!)

The library created here is called a static library.

In the static library, the necessary instructions are actually embedded in the executable file at the time of linking.

On the other hand, the dynamic library only checks the library when the executable file is generated. The code you need at runtime will actually be attached later on execution.

**[Quite aside]**

By the way, do you feel like you have seen something similar?

Remember when you wrote a simple program that used math library functions.

| math-sample.c |
| --- |
| #include <stdio.h>  #include <math.h>  int main(int argc, char \*argv[]){  double angle = 45.0;  printf("sin(%g[degree]) is %f\n", angle, sin(angle / 180.0 \* M\_PI));  return 0;  } |

$ gcc math-sample.c -lm

$ ./a.out

$ ldd ./a.out

Check what are similar and what are different.

In this example, the math library is referenced as a dynamic library.

There is also a static library of the math library. The function is the same.

Let's check the contents.

$ less /usr/include/math.h

$ ar t /usr/lib/libm.a

Although the size is different, both libSC.a and libm.a have the same structure.

(When searching for a file with a specific file name under a certain directory [for example, under /usr], it is normal to search with $ find /usr -name libm.a on Unix.)

Still not for sure? Let's play around more.

(Unfortunately, the following cannot be performed in an environment where /usr/lib/libm.a does not exist.)

Replace the existing math library (libm.a) with your own sugaku library (libsugaku.a).

(Just copy libm.a and rename it.)

$ cp /usr/include/math.h sugaku.h

$ cp /usr/lib/libm.a libsugaku.a

| math-sample-sugaku.c |
| --- |
| #include <stdio.h>  #include "sugaku.h"  int main(int argc, char \*argv[]){  double angle = 45.0;  printf("sin(%g[degree]) is %f\n", angle, sin(angle / 180.0 \* M\_PI));  return 0;  } |

$ gcc -o math-sample-sugaku math-sample-sugaku.c -L. -lsugaku

$ ./math-sample-sugaku

$ ldd ./math-sample- sugaku

We don't use the math library libm.a anymore. We are using libsugaku.a instead.

$ ldd a.out

$ ldd math-sample-sugaku

**Exercise**

**04-03-ex1**: Use unix commands to clearly indicate that the above math-sample-sugaku does not use the OS standard math library.

**04-03-ex2**: Describe the difference between dynamic and static libraries.

**04-03-ex3**: In the current OS, many dynamic libraries are used at the time of linking. Describe the advantages of dynamic libraries compared with static libraries and consider why dynamic libraries have become so widely used in current operating systems.