Fundamentals of Programming 1

Libraries

Arkadiusz Chrobot

Department of Computer Science

January 15, 2020

Outline

Libraries

The source code of large computer programs is usually divided into separate files that are generally called *translation units* and in case of compiled programming languages – *compilation units*. The objective of this partition is twofold. It improves the legibility of the code by grouping elements of program that serve the same purpose and it allows for reusing the code for creating different software. The translation units that contain functions, variables, data types and constants for using in many different program are called *libraries*. They are not only a "containers" for elements of a program, but also allow the programmer to manage the code, by deciding which of the elements should be available outside the library and which should be hidden inside.

Interface and Implementation of the Library

Elements of software which are gathered in a library and are available outside, for public use, create the library *interface*. Elements closed inside the library form its *implementation*. To make this distinction possible a programming language has to provide special *mechanism for hiding the code*. To explain the necessity of separating an interface from an implementation, two points of view on the library structure have to be taken into consideration. The first one in associated with the programmer who uses the library. She or he wants the library to be easy to apply in her or his code. The programmer wants also to have a clear information which elements of the library she or he can use in her or his code and how. When a new version of the library is released, perhaps with an extended interface or changed implementation, the programmer's software should compile with this version of library without any modifications. The second point of view is associated with the programmer who creates the library.

Interface and Implementation of the Library

Continued

She or he wants to have the possibility of introducing changes or corrections to the library code, but she or he doesn't want to make harder the job of the programmer who uses the library by making her or him to modify her or his program. To fulfill requirements of those two types of programmers the elements of the library that are likely to stay the same in subsequent versions should be made a part of library interface. If a function is such an element then the way it is invoked must be the same in new releases of the library. This means that the name of the function, the types, order and even number of its parameters have to stay unchanged. The interface is responsible for cooperation between the library and other units of translation, so it can only be extended in next versions of library. It cannot be rebuilt or narrowed down. The creator of the library should put all the elements that are likely to change in next version in its implementation. Those elements decide about the internal workings of the library and should be hidden to the programmer that uses the library.

Interface and Implementation of the Library

Continued

A similar strategy is followed by the car manufactures. In a new model they try to preserve the layout of the elements necessary for controlling the auto, like the steering wheel, clutch, gauges etc. while changing the elements that decide about the performance or look of the car, like the engine, gearbox, chassis, body, etc.

Libraries in the C Languages

The C language allows the programmers to create and use static and dynamic libraries. The former are included in the program during compilation. The latter are loaded into the computer's memory only when a program references them, for example by calling one of the functions they contain. A single dynamic library loaded in the memory may be shared by more than one program.
The static and dynamic libraries are crated in the same way, but the way of using the latter depends on the operating system. Therefore only the static libraries are covered in the lecture.

Declaration and Definition of a Variable

For the seek of simplicity the introduction of a variable to the program is called a declaration. In reality, if a variable is used only in a single translation unit, then when declared it also defined. If a variable is used in many translation files, then its declaration may be separated from its definition. Moreover there can be many declarations of the variable, but only one definition has to exist. The declaration is information for the compiler, that a variable is defined in some other translation unit, but is used in the file that contains the definition. The declaration, on the other hand informs the compiler that it has to put in the output code instructions that allocate the memory for the variable (i.e. literally create it) and initialize it. The declaration of a variable starts with the `extern` keyword and that's basically what differentiates it from definition of the variable. The declared variable must be defined somewhere in the program, usually in another translation unit. Otherwise the program won't compile. The next slide presents an example in which a variable is declared and defined in two separate compilation units.

Declaration and Definition of a Variable

Example

File 1: external.c

```
int foo;
```

File 2: program.c

```
#include<stdio.h>

extern int foo;

int main(void)
{
        printf("%d\n",foo);
        return 0;
}
```

Declaration and Definition of a Variable

Comment

The `external.c` file contains the definition of the `foo` variable, which is used in the `program.c` file. Since it is a global variable, its initial value is zero.

Static Variables

If the variable was to be *global*, but unavailable outside the translation unit in which it is defined, its definition should be prefixed with the `static` keyword. The keyword has been already introduced in the lecture on functions. If it is used with a local variable it makes the variable persistent between subsequent calls of the function where it is defined. Moreover, the initial value of the variable (i.e. before the first call of the function) is zero. In case of the global variables, the `static` keyword narrows down their scope to file where they are declared. It is safe for the functions defined in the same translation unit, to directly (i.e. without using parameters) reference those variables, because no other code can influence the state of those variables.

Declaration and Definition of Function

Similarly to variables also functions can be defined and declared in separate translation units. The declaration of a function consist of its header ended with a semicolon. It is also allowed to leave out the names of function's parameters. Only their types have to be present in the declaration. If a function is defined in separate translation unit, then its declaration can be prefixed with the `extern` keyword, however it is not mandatory. If the function should be available only in the file where it id defined, then its header should start with the `static` keyword. It can be concluded, that the `static` keyword in the context of libraries allows for hiding details of their implementation, which should be unavailable outside. The next slide shows a declaration and definition of a single function placed in a two different files.

Declaration and Definition of Function

Example

File 3: external_2.c

```c
#include<stdio.h>

void print(int variable)
{
        printf("%d\n",variable);
}
```

File 4: program_2.c

```c
extern int foo;
extern void print(int);

int main(void)
{
        print(foo);
        return 0;
}
```

Declaration and Definition of Function

Comment

The `print()` function is defined in the `external_2.c` file. Because it calls the `printf()` function, it is necessary to include the `stdio.h` header file in this translation unit. The `print()` function is invoked in the `program_2.c` translation unit, where the function is also declared. The code in the aforementioned file references also the `foo` variable, which is defined in the `external.c` file.

Header Files

Using variables and functions defined in different files requires repeating their declarations in every translation unit where they referenced or invoked. This inconvenience can be mitigated by placing the declarations in a *header file*. Such a file contains a source code, just like other translation unit, but its name has a different extension: `.h`. The header file may contain declarations of functions and variables defined in at lease one translation unit with the `.c` extension, but it shouldn't contain their definitions or any other statements that force the program to allocate memory. The other "safe" statements beside declarations of files and variables are type definitions (for structures, unions and enumerated types) and preprocessor macros, which will be described in the lecture. The next slide presents an example of a header file.

Header Files

File 5: header.h

```
#ifndef HEADER_H
#define HEADER_H

extern int foo;
extern void print(int);

#endif
```

Header Files

Comment

The exemplary header file contains declarations of a function and a variable from the previous examples. They are surrounded by directives of preprocessor that make sure that the declarations form the file will be placed in a source code only once, even if the header file itself may be included many times in the program. The `#ifndef` directive makes the preprocessor to make some actions, provided that the marker that follows the directive is **not defined**. The marker is called an *include guard*, *macro guard* or *header guard* and it can be any legal identifier, but usually its definition in based on the header file name, just like in the example. If the header guide is not defined that the preprocessor copies all the code between the `#ifndef` and `#endif` to the place in program where the header file is included. The code contains the definition of the header file. It resembles definition of a constant but the value is missing so, only the name (the marker) is defined. If the header file is included in the source code more than once the preprocessor will detect the definition of the header guard and will take no actions.

Header Files

Inclusion of a Header File to a Program

The inclusion of the header file to the file with the `.c` extension or other header file is done by the preprocessor. It is a program that takes a part in the compilation process by preparing the source code for the compiler. If the header file doesn't contain any macros, the preprocessor copies its content to the file where it is included, provided it hasn't been already copied. The `#include` directive is used for including the header file in other file with the source code. If the header file is in the preprocessor default directory for header files, then the directive is followed by the name of the file embraced by angle brackets. If the file is kept in the same directory as the rest of program's files, then its name should be embraced by quotation marks instead of angle brackets. Providing its path could be necessary if it is in other nonstandard directory. It is recommended that the header file is included in the file containing the definitions of functions and variables. It allows the compiler to check the validity of declarations and definitions. If the latter file uses macros or types defined in the header file, then the header file must be included.

Header Files

Inclusion of a Header File to a Program — Remark

An important conclusion should be drawn from the description of the header file inclusion procedure: If the program doesn't need any definition or declaration contained in a header file then the header file shouldn't be included in its source code. Otherwise the content of the file will only increase the size of the executable output file. Next slide demonstrates the usage of the header file in a program's source code.

Header Files

Example

File 6: external2.c

```c
#include<stdio.h>
#include"header.h"

void print(int variable)
{
        printf("%d\n",variable);
}
```

Header Files

The previous slide contains the modified `external_2.c` file. A directive is added that includes the `header.h` file. It is assumed that both files are in the same directory. It is not necessary to include the header file into the `external_2.c` file, but it allows the compiler to check compatibility of declaration and definition of the function.

Header Files

Example

File 7: external.c

```c
#include"header.h"

int foo;
```

Header Files

Comment

The previous slide contains modified version of `external.c` file. The inclusion of the header file in this case is also not necessary, but it also allows the compiler to check the compatibility of the declaration and definition of the variable.

Header Files

Comment

Plik 8: program2.c

```c
#include"header.h"

int main(void)
{
        print(foo);
        return 0;
}
```

Header Files

Example

Header files along with related source files (the files with the `.c` extension in names) form libraries. The program from the previous slide demonstrates the usage of a static library. The included header file contains declarations of the `foo` variable and the `print()` function defined in another source file. Thus the `main()` function can use them as their would be declared in the same translation unit as the function.

Libraries — Summary

A library in the C language consists of a header file and at least one source file. The header file can contain definitions of macros and types as well as declarations of functions and variables. The source files (already compiled or not) contain the definitions of those functions and variables. It is also possible to build a library consisting of only one source file and many header files that at least contain definitions of function and variables declared in the source file. Finally, a library with many header files related to many source files is also possible.

Preprocessor Macros

The header files handling isn't the only job of the preprocessor. It hes its own programming language, consisting of directives, which isn't translated to the machine language, but is used for controlling the compilation process. Some of the directives of the processor have been already introduced in the lecture. For example, the `#define` directive was often used to define constants. Such constants are also a preprocessor macros. The names of the constant used in the source code, are substituted by the preprocessor with its value. However, the macros can be something else than simple constant definitions. A macro can be a whole subroutine with its own parameter list. When used in the program it is not called like a function, but it is expanded, which means that its code is copied to the place in the code where its name is listed together with its arguments. The advantage of the macro over a function is that it do not requires the program to allocate a stack frame, so it saves the time and memory. The disadvantage is that the preprocessor that expands the macro doesn't check the validity of the macro code, which may lead to hard to discover defects in the code.

Preprocessor Macros

Macros definitions are usually placed in header files, but it is also possible to put them in source files. The second approach is used in all examples in the lecture, to simplify the code of exemplary program. While expanding the macro the preprocessor doesn't check its validity. It only prepares the source code for the compiler, which performs such an activity. This means that if there is a syntax error, it will be discovered by the compiler in other place in the source code than the definition of the macro. Thus it is more difficult to discover bugs in macro than in a function. Also the macros are more prone to convey logical errors to the code. The next slides present a program containing a definition of a simple macro, which may cause unexpected problems in some cases. The corrected version of the macro will also be introduced.

Preprocessor Macros

Example

```c
#include<stdio.h>

#define MULTIPLY(X,Y) X*Y

int main(void)
{
    int a=2, b=3;
    printf("The macro result: %d\n",MULTIPLY(a,b));
    return 0;
}
```

Preprocessor Macros

Comment

A macro named `MULTIPLY` is defined in the program, which multiplies arguments passed to it by the `X` and `Y` parameters. The definition of the macro does not contain a semicolon. It may however be necessary to used it in the location of code where the macro is expanded. Please also observe that the parameters of the macro don't have the type specifications. The names of the macro and parameters are capitalised, but it is only a matter of convention. Most C programmers however follows it. In the example the parameters of macro are substituted in the `main()` function by two arguments – the `a` and `b` variables. The value of the former is `2`, and the latter `3`. As expected the result of the macro is `6`. A small modification of the program's code may lead to an incorrect result from the macro, which is demonstrated in the next slide.

Preprocessor Macros

Example

```c
#include<stdio.h>

#define MULTIPLY(X,Y) X*Y

int main(void)
{
    int a=2, b=3;
    printf("The macro result: %d\n",MULTIPLY(a+1,b-1));
    return 0;
}
```

Preprocessor Macros

Comment

The modified version of the program will display `4` on the screen as a result. The expected value is `6`. The error is caused by the lack of arguments evaluation when the macro is expanded. The `x` and `y` parameters in the `x*y` expression are substituted by the `a+1` and `b-1` arguments, so the macro is expanded to `a+1*b-1` expression. Finally, when the program runs the expression is once more time evaluated to the `2+1*3-1` form and hence the wrong result. To correct the defect each of the parameters should be embraced by parentheses when applied in macro code. Unfortunately, this solution fails if the arguments are of incompatible type with the expression (for example they are strings). Such an error is detected by the compiler.

Preprocessor Macros

Example

```c
#include<stdio.h>

#define MULTIPLY(X,Y) (X)*(Y)

int main(void)
{
    int a=2, b=3;
    printf("The macro result: %d\n",MULTIPLY(a+1,b-1));
    return 0;
}
```

Preprocessor Macro

The preprocessor macros may contain more than one statement. Each of the statements can be placed in a separate line of the source code, provided that some additional requirements are met. If the macro has more than two lines of code, then all the lines between the first and the last one have to be ended with the backslash character: \. The backslash is used for informing the preprocessor that a line of macro code is followed by another one. Very often the code of the macro body is enclosed in a `do…while()` loop that has `0` as a condition and **isn't ended** with the semicolon. Such a loop is executed only once. The loop allows the programmer to put a semicolon after the macro expansion statement in the code wherever it seems to be necessary.

Makra preprocesora

Przykład

```
#include<stdio.h>

#define SWAP(A,B) do {\
    int tmp;\
    tmp = (A);\
    (A) = (B);\
    (B) = tmp; \
} while(0)

#define PRINT(A,B,C) printf("%s: %d, %s: %d, %s: %d\n",#A,(A),#B,(B),#C,(C))

int main(void)
{
    int a=2, b=3, c = 4;
    if(a>b)
        SWAP(a,b);
    else
        SWAP(a,c);
    PRINT(a,b,c);
    return 0;
}
```

Preprocessor Macros

Comment

The usage of the `do…while(0)` in the code of the `SWAP` macro makes it possible to end its expansion statement twice – before and after the `else` keyword in the conditional statement. There is also another macro defined in the exemplary program. Its name is `PRINT` and it displays names and the values of the variables on screen. The names of the variables substituted for the macro parameters are obtained with the use of `#` (the hash) operator. For example the `#A` expression means that everything that stands between the hash and the following comma will be converted to a string. Processor has another such an operator denoted by double hashes: `##`. This operator concatenates two strings ("glues" them together). Its application is demonstrated in the next slide.

Processor Macros

Example

```c
#include<stdio.h>

#define VARIABLE(SCOPE) int SCOPE##_variable

VARIABLE(global);

int main(void)
{
    VARIABLE(local) = 6;
    printf("global_variable: %d, local_variable: %d\n",
                                    global_variable, local_variable);
    return 0;
}
```

Preprocessor Macros

Comment

The `VARIABLE` macro is applied in the program to declare and define two variables of the `int` type, namely the `global_variable` and `local_variabl`
Moreover, the latter variable is also initialized. The code in not easy to follow, but it demonstrates the usage of the `##` operator.
If needed a definition of a macro may be excluded from the whole of code or a part of it with the use of `#undef` directive. It should be followed by the name of the macro to exclude. The macro won't be available for the code that follows the directive.

Assertions

Macros are often used for finding defects and verifying the code. In the C language there is a macro called `assert` defined in the `assert.h` header file. It is an implementation of the assertion concept introduced to computer science by Robert Floyd. The assertion is an expression that should always true. The assertions are evaluated to check the correctness of some parts of a code. For example, an assertion can be evaluated before and after a loop or a function is performed. If the value of such assertion is false, then there is at least one defect in the code that needs to be removed. The `assert` macro takes one argument, which is an expression. If the expression is false then the macro displays an appropriate message and terminates the program. Usage of the macro is demonstrated in the next slide.

Assertions

Example

```c
#include<stdio.h>
#include<assert.h>

int main(void)
{
    int i;
    for(i=1;i<10;i++) {
        assert(i<=5);
        printf("%d%%%d=%d ",5,i,5%i);
    }
    return 0;
}
```

Assertions

Comment

In the example form the previous slide the `assert` macro is used for checking if the divisor in the expression is not greater than the dividend. If the macro recognizes that the condition is not met, it will display a message and terminate the program. Please also observe the formating string in the `printf()` function call. It needs to display the reminder operator symbol which is `%`. Because it is a special character in the formatting string, it has to be doubled. That's why the string looks complicated. The programmers tend to switch off assertions when they deliver the final version of the program. While it is controversial it is worth to know how to do it. It is sufficient to define a `NDEBUG` macro at the beginning of the program's main translation unit or as an option of the compiler. In the former case the macro should be defined before the `assert.h` file is included. The next slide contains an example explaining the usage of the macro.

Assertions

Example

```c
#include<stdio.h>
#define NDEBUG
#include<assert.h>

int main(void)
{
    int i;
    for(i=1;i<10;i++) {
        assert(i<=5);
        printf("%d%%%d=%d ",5,i,5%i);
    }
    return 0;
}
```

Macros and Debugging

There are also other macros defined in the C language which may be useful in the process of debugging a program. Using them don't require including any header files. Some of them are described here:

- `__DATE__` evaluated to the compilation date,
- `__TIME__` evaluated to the compilation time,
- `__FILE__` evaluated to the name of the compiled file,
- `__LINE__` evaluated to the line number in the compiled file.

Conditional Compilation

Some of the preprocessor directives allow for controlling the process of compilation, i.e. they make it possible to decide whether some part of the code should be compiled. Some of the directives used in the code that applied the header guard. The table describes more of them.

`#ifdef`	Is similar to `#ifndef`. If the marker that follows it is defined the preprocessor processes the statements that follow the directive. If not, they are omitted.
`#else`	It is similar to the `else` keyword in the conditional statement.
`#elif`	It combines the `#else` and `#if` statements.

Examples of the conditional compilation usage are show in the next slide.

Conditional Compilation

Example

```c
#include<stdio.h>
#include<assert.h>

#ifndef NDEBUG
#define PRINT_VERBOSE(X) printf("The variable %s has a value of: %d\n",#X,(X))
#else
#define PRINT_VERBOSE(X)
#endif

int main(void)
{
    int i;
    for(i=1;i<10;i++) {
        PRINT_VERBOSE(i);
        assert(i<=5);
        printf("%d%%%d=%d ",5,i,5%i);
    }
    return 0;
}
```

Conditional Compilation

Comment

A `PRINT_VERBOSE` macro is defined in the program, that displays a message concerning a value of a specified variable. The information is valuable in the process of debugging, should there be an error involving the variable. However in the everyday usage of the program it is redundant. That's why, when the `NDEBUG` is defined, an empty definition of the `PRINT_VERBOSE` macro is applied. The next slide contains an example of such a case.

Conditional Compilation

Example

```c
#include<stdio.h>
#define NDEBUG
#include<assert.h>

#ifndef NDEBUG
#define PRINT_VERBOSE(X) printf("The variable %s has a value of: %d\n",#X,(X))
#else
#define PRINT_VERBOSE(X)
#endif

int main(void)
{
    int i;
    for(i=1;i<10;i++) {
        PRINT_VERBOSE(i);
        assert(i<=5);
        printf("%d%%%d=%d ",5,i,5%i);
    }
    return 0;
}
```

Functions With a Variable Number of Arguments

The C languages allows for creating functions that take a variable number of arguments. An example of such a function is `printf()`. The list of such arguments is handled by preprocessor macros, which are defined in the `stdarg.h` header file. To accept a variable number of arguments the function has to be defined accordingly. It parameter list has to contain at least one regular parameter. A three dots (…) should be the last element of this list. Those dots inform the compiler that they can be replaced by any number of argument when the function is invoked. The dots cannot be preceded by an array parameter.

Functions With a Variable Number of Arguments

The access to the additional arguments is provided by five macros. The first one is the `va_start` macro which accepts two arguments on its own. The first argument is a previously declared and defined variable of the `va_list` type. It is a list of the additional arguments. The second argument of the `va_start` macro is last regular parameter of the function. The `va_start` macro initializes the `va_list`. The `va_arg` macro returns the value of the function's argument which type name is passed to the macro as its second argument. The first argument of the macro is the list of additional arguments. The `va_end` macro takes the list of additional arguments as its only argument and signals that the processing of the list is finished by the program. Finally, the `va_copy` macro is used when a copy of additional arguments list is required. It takes two variables of the `va_list` type as its arguments. The second one is the original list and in the first one the copy of the list will be stored.

Functions With a Variable Number of Arguments

Example

```c
#include<stdio.h>
#include<stdarg.h>

double average(unsigned int counter, ...)
{
    va_list arguments_list;
    int next=0, sum=0, i=counter;

    if(counter==0)
        return 0.0;

    va_start(arguments_list, counter);
    while(i--) {
        next = va_arg(arguments_list,int);
        sum += next;
    }
    va_end(arguments_list);
    return (double)sum/counter;
}
```

Functions With a Variable Number of Arguments

Example

```
int main(void)
{
    double result = average(5,1,2,3,4,5);
    printf("The average is: %.2f\n",result);
    return 0;
}
```

Functions With a Variable Number of Arguments

Comment

The `average()` function in the example counts the arithmetical average of some integer numbers passed to the function as its arguments. Only the first argument is passed to the function by a regular parameter. Its value defines how many additional arguments the function has. It is not required that the argument should have the same type, but such a case is demonstrated in the exemplary program.

The `inline` Functions

The **`inline`** functions are an alternative for the preprocessor macros. Similarly to them they could be expanded, but this time by the compiler, which check the validity of such expansion. In other words the **`inline`** function is like a regular function, but is potentially expanded instead of being invoked. Usage of such a function offers a better performance of the program, but increases the size of the executable output file and the time needed for compilation. It should be stated that the C language standard doesn't require the **`inline`** functions to be expanded at all. They should only offer a better performance than the regular functions. It is sufficient to add the **`static inline`** keywords at the beginning of the function's header to make it an **`inline`** function. An example of a program with an **`inline`** function is presented on the next slide.

The inline Functions

Example

```c
#include<stdio.h>

static inline void swap(int *first, int *second)
{
    int tmp = *first;
    *first = *second;
    *second = tmp;
}

int main(void)
{
    int a = 1, b = 2;
    swap(&a,&b);
    printf("a: %d, b: %d",a,b);
    return 0;
}
```

Thanks

Many thanks to Grzegorz Łukawski, PhD and Leszek Ciopiński, MSc for helping me to complete the Polish version of this slides.

Questions

?

The End

Thank You for Your attention!