

Systems Programming

1 Introduction to C

Secure Systems Group, LIT Secure and Correct Systems Lab
Johannes Kepler University Linz
`stefan.rass@jku.at`

1 Introduction to C

- The Compilation Process
- Projects Composed from Multiple Files

- Developed at Bell Laboratories in 1972 by Dennis Ritchie
Many of its principles and ideas were taken from the earlier language B and B's earlier ancestors BCPL and CPL.
- CPL (Combined Programming Language) supported both high level, machine independent programming and control over the behavior of individual bits of information, but was too large for use in many applications.
- In 1967, BCPL (Basic CPL) was created as a scaled down version of CPL
- In 1970, Ken Thompson, while working at Bell Labs, developed the B language, a scaled down version of BCPL written specifically for use in systems programming.
- Finally in 1972, a co-worker of Ken Thompson, Dennis Ritchie, returned some of the generality found in BCPL to the B language in the process of developing the language we now know as C.

- In 1983, the American National Standards Institute (ANSI) formed a committee, X3J11, to establish a standard specification of C. In 1989, the standard was ratified as ANSI X3.159-1989 "Programming Language C." This version of the language is often referred to as **ANSI C**, **Standard C**, or sometimes **C89**.
- In 1990, the ANSI C standard (with a few minor modifications) was adopted by the International Organization for Standardization (ISO) as ISO/IEC 9899:1990.
- This version is sometimes called **C90**. Therefore, the terms "C89" and "C90" refer to essentially the same language. We follow C90 in this course. The "latest" standard to address the C language was ISO 9899:2011, issued in 2011.
- This standard is commonly referred to as "C11" It was adopted as an ANSI standard in December 2011. C18 is only an error correction of C11 – no changes.

We will use C18 (ISO/IEC 9899:2018) in this course

- One level of abstraction "above" assembly
- Language constructs that immensely facilitate "structured programming", e.g. loops, conditional branching, etc.
- Automatic [stack](#) management
- Integrated [heap](#) management
 - Think of this as a sophisticated version of the memory allocator in the last assembly example
 - But [memory management](#) is still the responsibility of the [programmer](#)!
- Strongly [typed](#), but with unchecked type conversions! [Source-level portability](#) between processor architectures and operating systems
- The [standard C Library](#)
and many other standards-compliant libraries (e.g. POSIX)

- Both **high-level** and **low-level language**
- Better control of low-level mechanisms
- **Performance better** than Java (but this is not a universal truth any more!)
- **Java hides many details** needed for writing OS code (as does Python)

But:

- Not object oriented
- **Memory management** responsibility
- Explicit **initialization** and **error detection**
- More room for **mistakes**
- C exposes many details only needed for writing OS code
- Runs on almost all and even very small/slow CPUs

Fig. 1.1: hello-world.c



```
#include <stdio.h>
int main() {
    /* Stream comment;
    ...multi-line (C90) */
    printf("Hello World\n\tNot again :-)\n");
    return 0; // comment until end of line (C99)
}
```

Compile (and link):

```
$ gcc hello-world.c -o hello
```

Run:

```
$ ./hello
Hello World.
    Not again!
```

Summarizing the example

- **#include** <stdio.h>
 - Include header file "stdio.h"
 - No semicolon at end
 - Small letters only – C is case-sensitive
- **int** main(**void**) {...}
 - Program entry point
 - Is the first (user) code executed

- `printf("message you want printed");`
Prints a message
 - `\n` = newline
 - `\t` = tab
 - `\` in front of other special characters within `printf`
 - `printf("Have you heard of \"The Rock\" ? \n");`

Screen **output** works easiest with the `printf()` function. It takes:

- A format string, containing %-markups that the system replaces by numbers, strings, etc.
- The values to be printed (any number of parameters, separated by comma).
- Be careful: each %-placeholder must correspond to one parameter after the format string.

Example 1.1:

- `printf("file %s processed up to %d%%", fileName, progress);` will replace the %s by the string that fileName contains, and will replace %d by an integer stored in the variable progress. The double %% prints out the percent symbol directly.
- `printf("the symbol %c has value %f\n", chr, floatValue);` will replace %c by an ASCII character whose value comes in by the variable chr, and will replace %f by a floating point value stored in floatValue.



Screen `input` works likewise by the function `scanf()`, which takes:

- A format string of the same syntax as for `printf()`,
- followed by a number of variables, each corresponding to the respective `%`-placeholder.
- The only difference is that we need to supply addresses for the variables to enable write-access by `scanf()` → see Example 1.2.

Technically, we need to supply pointers → Chapter 5

Example 1.2: `printf("type in your age: "); scanf("%d", &age);` will read an integer (`%d`) into the variable `age`, more precisely, write it to the `address` that is obtained by prefixing the variable with the `&`-symbol. Details will follow later. ◇

Remarks:

- The detailed specification and explanation of I/O functions requires much more preparation than we have at this point, especially the concept of pointers that we will cover later.
- For the time being, take these examples as templates, unless the code that you will be asked to write does not already contain the respective I/O instructions.
- Full details will thus follow in Chapter 6

In C, compilation starts by running the **preprocessor** on the source code

- The preprocessor is a simple program that replaces patterns in the source code with other patterns the programmer has defined using **preprocessor directives** (see before).
- These directives are used, among other things, to save typing, to increase code readability, to control inclusion of header files, etc
- The pre-processed code is often written to an intermediate file



Compilers usually do their work in two passes

- The first pass **parses** the pre-processed code into a "parse" tree
- In the second pass, the **code generator** walks through the parse tree and generates either assembly code or machine code for the nodes of the tree.
- If the code generator creates assembly code, the **assembler** must then be run.
- The end result in both cases is an **object module** (a file that typically has an extension of .o or .obj).



The linker combines a list of object modules into an **executable program** that can be loaded and run by the OS:

- When a function in one object module makes a reference to a function or variable in another object module, the linker **resolves these references**
- The linker also makes sure that all the external functions and data you claimed existed during compilation **do exist** exactly **once**
- The linker also adds a special object module to perform start-up activities The linker can search through special files called libraries in order to resolve all its references
- A library contains **a collection of object modules** in a single file



- C compilers generally accept so-called **preprocessor directives**, to take control of the compilation process within the source file.
- All preprocessor directives start with a **#**, such as **#include**, **#define**, etc.
- Preprocessor directives have no end-of-instruction token like a semicolon ; or others. They are terminated by the line break (and not earlier)

Typical uses of preprocessor directives:

- **#define**: This defines a **macro**, which is a piece of code that gets **literally copied into** the actual source code **before compilation**. A typical use is the **declaration of constants**, such as

- **#define** BUFFER_SIZE 256
 - **#define** M_PI 3.14159265358979323846264338327950288

or the declaration of expression templates, acting **similar as functions**, such as

- **#define** min(X, Y)((X) < (Y)? (X): (Y))


```
x = min(a, b);           x = ((a) < (b)) ? (a) : (b));  
y = min(1, 2);           y = ((1) < (2)) ? (1) : (2));  
z = min(a + 28, *p);      z = ((a + 28) < (*p)) ? (a + 28) : (*p));
```

Remark: the `(a) ? (b) : (c)` code is known as **ternary if**: it evaluates the Boolean expression `a`, and if true, evaluates to the value of (the expression) `b`, and else comes back with (the value of) the expression `c`.

- **#include**: this copies the entire content of the file (given as the argument) into the location of the **#include** statement in the source code. We will go into deeper details later at several occasions (e.g., slide 4-11).
- **#if**, **#elif**, **#else**, **#endif**, **#ifdef**: conditional compilation. Pieces of code enclosed within an **#if** ... **#endif** directive are compiled if (and only if) a certain condition is met.

The most common use-cases are:

- Prevention to include a file multiple times: many of the standard libraries `#define` a certain constant (e.g., `__STDIO_H` in case of `stdio.h`, or similar), and use `#ifdef` to check if the symbol exists and hence the file has been included already. If so, the whole file (or section) will be skipped.

- Compilation specifically for a platform, for example

```
#if defined(WIN32) || defined(_WIN64) || ...  
    some Windows-specific code...  
#elif defined(__APPLE__)  
    some Mac-specific code...  
#endif
```

- Many other directives exist (not relevant for us here)

Fig. 1.2: `preprocessor.c`



```
#include <stdio.h>
#define DANGERLEVEL 5
    /* C Preprocessor - substitution on appearance.
       Somewhat like Java 'final' */
int main(void) {
    float level = 1;
    /* if-then-else as in Java */
    if (level <= DANGERLEVEL) { /* replaced by 5 */
        printf("Low on gas!\n");
    } else {
        printf("Good driver!\n");
    }
}
```

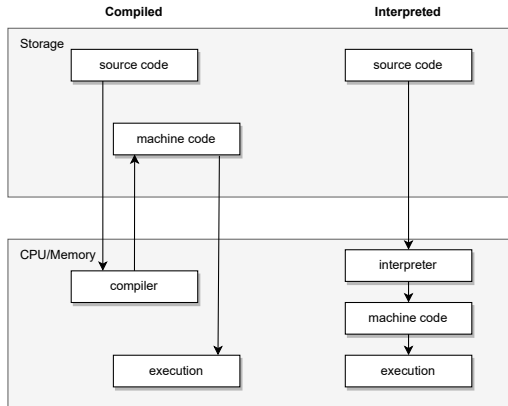
Exercise: What if we wrote `#define DANGERLEVEL 5;` (i.e., with the terminating semicolon)?

- A compiler translates source code directly into assembly language or machine instructions
→ The eventual end product is a file or files containing **machine code**
- Some languages (such as C and C++) are designed to allow pieces of a program to be compiled independently
 - These pieces are eventually combined into a final executable program by a tool called the **linker**
 - This process is called **separate compilation**
Attention: This is different from creating/using libraries!
Makefiles are a standard tool here → see Appendix A
 - **Exercise:** Why is this a useful feature?
- Modern compilers can insert information about the source code into the executable program. (`gcc` option `-g`)
→ This information is called **debug information** and is used by **source-level debuggers**

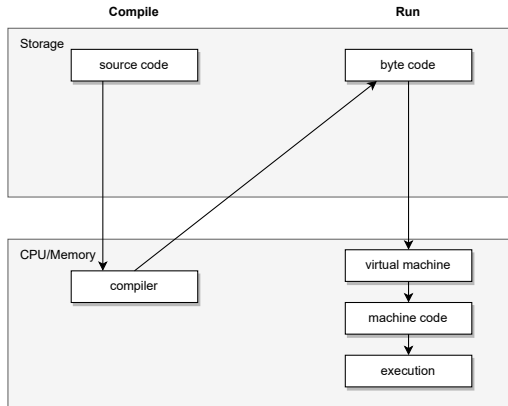
- When you make an external reference to a function or variable in C, the linker, upon encountering this reference, can do one of two things:
 - If it has not already encountered the definition for the function or variable, it adds the identifier to its list of "unresolved references"
 - If the linker has already encountered the definition, the reference is resolved
- If the linker cannot find the definition in the list of object modules, it searches the libraries
 - Libraries have some sort of indexing, so the linker doesn't need to look through all the object modules in the library – it just looks in the index
 - When the linker finds a definition in a library, the entire object module (but not the entire library) is linked into the executable program
- Because the linker searches files in the order you give them, you can pre-empt the use of a library function by inserting a file with your own function, using the same function name, into the list before the library name appears
⇒ Think **very hard** before you actually do this. . .

- When a C executable program is created, certain items are "secretly" linked in
- One of these is the [start-up module](#), which contains initialisation routines that must be run any time a C program begins to execute
- These routines [set up the stack](#) and [initialise](#) certain variables in the program
- Initialisation of the [memory manager](#)
- The dynamic linker used to find and access the [standard C library](#) (→ Appendix C) at run-time is also included in the executable
- The linker always searches the [standard library](#) for the compiled versions of any "standard" functions called in the program (e.g., `printf()`, `scanf()`)
- Because the standard library is always searched, you can use anything in that library by simply including the appropriate header file in your program
- If you are using an add-on library, you must explicitly add the library name to the list of files handed to the linker

In case of **C** (compiled):



In case of **Java** (**interpreted**):



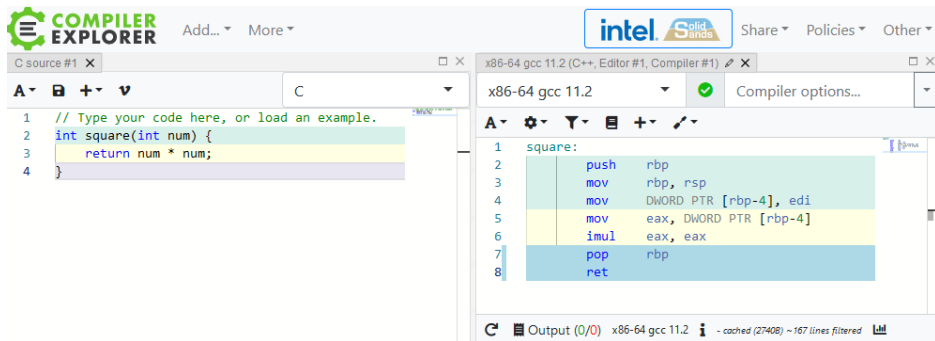
- In some occasions, it can be useful (even necessary) to take a look at what the compiler has produced as assembly code, embodied in the program binary. The insights from this are manifold, such as (but not limited to):
 - Seeing how functions are called and parameters are passed. We will go deeper into this on slides 4-15ff
 - Seeing which other procedures and symbols are added by the linker to our code (such as library calls, and others)
 - Seeing addresses of where functions are located in our code. Especially when writing shared libraries, it is important to have the functions at addresses that are "Position Independent". The address of (our own) functions may be fixed, but addresses of shared library code is looked up in a Procedure Linkage Table (PLT), mapping function symbols to their "Position Independent Code" (PIC) → not covered here any further
 - ...
- It is **recommended** to occasionally take a disassembly of your programs to see how your C code was translated into machine code. There are several tools available for this purpose, such as

- **Linux:** `$ objdump -d /path/to/binary.`

Example 1.3: For the hello world example from slide 1-7, we get, upon disassembling via
`$ objdump -d bin/Debug/HelloWorld`



- **Online** (platform independent): Compiler explorer (<https://godbolt.org/>). **Exercise:** visit the website, copy and paste the "Hello World" program, and study the resulting disassembly.



The screenshot shows the Compiler Explorer interface. On the left, the 'C source #1' tab is active, displaying the following C code:

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     return num * num;
4 }
```

On the right, the 'x86-64 gcc 11.2 (C++, Editor #1, Compiler #1)' tab is active, showing the disassembly for the 'square' function:

```
1 square:
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     imul    eax, eax
7     pop     rbp
8     ret
```

At the bottom, the output section shows 'Output (0/0) x86-64 gcc 11.2' with a status of '- cached (27408) ~ 167 lines filtered'.

- Many projects have pieces of code shared between several modules, externalized and put into separate files
- This eases maintenance and reduces redundancy (makes the code cleaner)
- To make a set of routines (functions) available at several other places, it is enough to
 - 1) collect the **prototypes** of these functions in a **header-file**, extension `.h` → as in Example 1.4

Example 1.4 (Project composed from multiple source files):

mypgm.h

```
void myproc(void);  
extern int data;
```

hw.c

```
#include <stdio.h>  
#include "mypgm.h"  
  
int main(void){  
    myproc();  
    printf("%d", data)  
}
```

mypgm.c

```
#include <stdio.h>  
#include "mypgm.h"  
  
int data;  
  
void myproc(void)  
{  
    data = 2;  
    ... /* some code */  
}
```



- 2) compile the implementation of these functions separately → makefiles are helpful here (see Appendix A)

Example 1.5 (Generating individual source code files; Example 1.4 cont'd):

```
$ gcc -c mypgm.c  
$ gcc -c hw.c
```



3) tell the linker to link the shared library/resources to the actual code

Example 1.6 (Link object files to executable; Example 1.5 cont'd):

```
$ gcc -o hw mypgm.o hw.o
```



- Standard libraries, which contain functions whose prototypes are in the usual includes like `stdio.h`, `stdlib.h`, etc., are automatically linked, by the linker's default configuration.

- **#include** directives tell the preprocessor to import functions, typically prototypes, into other code.
 - File names in angular brackets, such as:
#include <header.h>
cause the preprocessor to search for the file in the "include search path". The mechanism for setting the search path varies between machines, operating systems, and C implementations.
Typical use: standard/OS libraries location
 - File names in **double quotes**, such as:
#include "local.h"
tell the preprocessor to search for the file in an "implementation-defined way." What this typically means is to search for the file **relative to the current directory**. If the file is not found, then the include directive is reprocessed as if it had angular brackets instead of quotes.
Typical use: parts of this project/program

The following files are often/typically included → see Appendix C for more/details

- `stdio.h`: Standard input/output, e.g. `FILE`, `stdin`, `printf`, ...
- `stdlib.h`: Standard library, e.g. `malloc`, ...
- `unistd.h`: POSIX standard library, e.g. system call wrappers `read`, `write`, `chdir`, `fork`, ...
- `errno.h`: defines macros for reporting and retrieving error conditions using the symbol `errno`. Certain library functions store a value in `errno`, which acts like an `int` variable when used for error/exception handling. To this end, various common error codes are defined in this file.
- `string.h`: String functions, like `strcpy`, `strcat`, `memset`, `atoi`, ...
- `sys/types.h`: Various data types, e.g. `size_t`, `time_t`, ...
- `stdint.h`: Data types of fixed/guaranteed length, e.g. `uint8_t`, ...
- and many more → check out for yourself what is there ([Exercise](#))