Introduction to C

Programming Fundamentals

Dr Kevin Chalmers

School of Computing
Edinburgh Napier University
Edinburgh

k.chalmers@napier.ac.uk

Edinburgh Napier UNIVERSITY

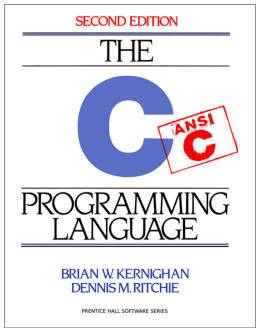
Notes

Outline

Notes

References

- Kernighan & Ritchie (1988), The C Programming Language, 2nd Edition
  - Co-authored by the creator of C
- `www.cplusplus.com` - contains library documentation for C and C++



Notes

Outline

Notes

A Brief History of C

- Developed by Dennis Ritchie at the AT&T Bell Labs in the early 1970s
- Initially designed as a programming language to rewrite the UNIX operating system
- Still one of the most popular languages today (if not the most popular)
- Many languages have inherited its syntax
- Most operating systems are written using C

Figure: Dennis Ritchie, C pioneer, and Ken Thompson, UNIX pioneer at Bell Labs (Wikimedia Commons)

Notes

C Specifications

- C is a *high-level* programming language, i.e, offering a level of abstraction from the hardware and a *"human-friendly"* syntax
  - Most other languages you might have experience of are also high-level, e.g. Java, C#. Their syntax is inherited from C.
- However, due to its age and closeness to the CPU and memory, C is lower-level than other modern languages
- Its high performance and low footprint make it a language of choice for operating systems, some hardware drivers, graphics, real-time programming, etc.

Notes

Hello World

- It is customary to test a programming environment using a program printing Hello World. In C, it would look like this:

Hello World in C

```c
// Hello World example in C
#include <stdio.h>

int main(int argc, const char **argv)
{
    printf("Hello, World!\n");
    return 0;
}
```

- Those already familiar with Java or C# will recognise the general syntax, albeit with some differences

Notes

Hello World

- Comment lines start with `//`. These allow you to describe your code and are ignored by the compiler
  - `// Hello World example in C`
- Lines starting with # are messages to the preprocessor (we will look at this later in the module). `#include` tells the preprocessor to include an existing library. In this case STanDard Input-Output
  - `#include<stdio.h>`
- The program requires a *main* function which is the start of the program. A function takes the form *`return-type`* `function-name(`*`parameters`*`)`. Our main function returns an `int` and takes two parameters representing the command line instructions (see example in workbook).
  - `int main(int argc, char **argv)`

Notes

Hello World

- The body of a function (a block of code or instructions for the computer) starts and ends with curly-brackets { }
- The `stdio.h` library header has given us access to commands to a number of in-built functions. One of these is `printf`. `printf` allows displaying of messages on the command line.
- Here we print the message *Hello, World!*.
- The `\n` character at the end denotes a new line should be printed at the end.
- Notice that the message is surrounded by `" "` which denotes a *string of characters*.
  - `printf("Hello, World!\n");`
- Notice that the end of each expression has a semi-colon `;`. This is used to denote the end of an instruction statement to the computer.
- Our main function returns an `int` value which is often used to determine if an application has successfully completed. `0` means that it did
  - `return 0;`

Notes

Hello World

- So what did our program actually do. Let us break it down line by line
  1. Include input-output functionality
  2. Start the main function
  3. Print *Hello, World!* to the screen
  4. Return from main function
- Breaking down programs into a list of English statements is useful for new programmers - it allows you to think of the instructions as if talking to another person.
- This technique is often called *pseudocode*. We will look at this a number of times in the module.

Notes

Questions?

Notes

Outline

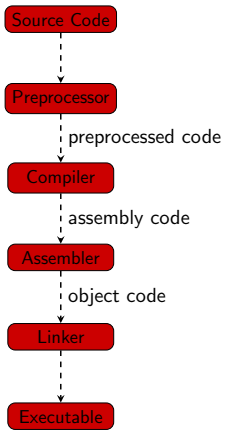
Notes

Building

- In order to run our *Hello World* program on a computer we need to build it
- Building a C program is necessary to translate the source code, understandable by a human, into a series of instructions understandable by a machine
- Building involves a several steps
  - Preprocessing
  - Compiling
  - Assembling
  - Linking

Notes

Building Steps



- The preprocessor executes all the preprocessor directives, creating a modified version of the source file
- The compiler generates assembly code for the target architecture (x86, x64, ARM)
- The assembler generates binary object code
- The linker links all the required object files with other binaries (e.g. operating system) and generates an executable
- You can tell the compiler to output the intermediate output (see the workbook)

Notes

Intermediate Output

- A C program consists of one or more source code files (.c extension) and zero of more header files (.h extension)
- C source is *"easily"* readable by a human
- Assembly code contains low-level instructions for your computer architecture
- Assembly code is understandable by a machine

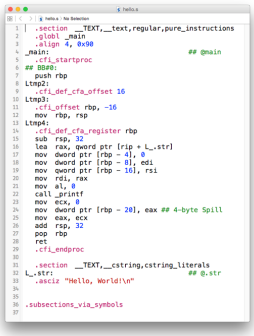


Figure: Intermediate Output from a Compiler

Notes

Building: Hands On

- This module has a bottom-up approach - you will learn how to build a program without assistance from an IDE (*Integrated Development Environment*) like Microsoft's Visual Studio or Eclipse.
- Source files are actually *plain text* files, with an extension denoting their language:
  - `.c` - C source file
  - `.cpp` - C++ source file
  - `.h` - header file (either language)
- It is therefore simple to build programs using a *command-line interface* (CLI)
- It is also important as you may sometimes need to work in a CLI only environment

Notes

Building Hello World

- Building our Hello World application from the Linux command line is quite simple. You can use either the GNU `gcc` compiler or the LLVM `clang` compiler:
  - `gcc hello.c -o hello`
    - `gcc` is the compiler we are using, so we are executing this program on the command line
    - `hello.c` is the file we are compiling
    - `-o hello` tells the compiler to name the *output* file *hello*. This is the executable's filename
- Other compilers are available. Microsoft Windows generally uses Microsoft's `cl` compiler. Apple normally uses `clang`. The principals are similar across the compilers however.
- The compiler also check if your code is syntactically correct. *What would happen if we removed a semicolon from our Hello World?*

Notes

Outline

Notes

Demo

Notes

Outline

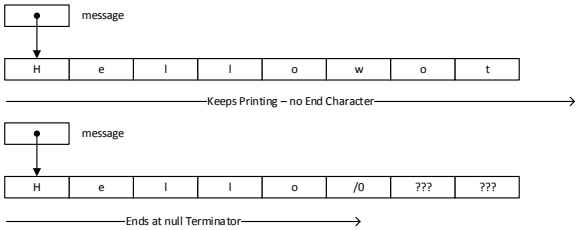
Notes

String I/O

- Our Hello World example displays a string to the command line: `"Hello, World!"`
- We want to be able to work with different strings in the future to make more useful C programs
- C is an old language, and its handling of strings is a lot more rudimentary than moderner languages and platforms
- String handling functionality is included in the `string.h` header file
- In C, string are treated like 1-dimensional *arrays* of characters

Notes

String Output



- C will treat strings as a block of memory and print each character stored until it reaches a null terminator \0
- The absence of a null terminator will provoke the unwanted output of additional memory content (actually a major issue for secure code)
- In some situations the compiler will add the null terminator for you

Notes

Null Terminators

- There is more than one way to declare strings in C, as presented in the workbook
  - `char msg_1[5] = {'H','e','l','l','o'};` - is a strict array of five characters with no null terminator. `printf` would provoke unwanted output
  - `char msg_2[8] = " World!";` - is a string constant matching exactly 8 characters. The null terminator is added by the compiler
  - `char msg_3[9] = "Goodbye!\0";` - is a string constant with an explicit null terminator. Note that in the future C++ will only allow explicit null terminators
  - `char msg_4[] = "Compler worked out my size!"` - the compiler will work out the necessary size of the array
  - `char *msg_5 = "Compiler worked out my size too!"` - similar to the above, but we are declaring the string as a pointer to an array of memory as opposed to a specific block

Notes

Formatted Output

- To print a string variable on screen, we would use the `printf("%s\n", msg)` expression, where `msg` is a string
- The `%s` part of the statement basically says *treat the parameter as a string of characters*
- Other output formatting exists, for instance `int` types (whole numbers) are represented with `%d`, `char` (single characters) with `%c`, and `float` (decimal numbers) with `%f`

Example of Formatted Output

```
int age = 36;
char[] name = "Kevin";
printf("My name is %s and I am %d years old.\n", name,
    age);
```

- This would print `My name is Kevin and I am 36 years old.`

Notes

Formatted Output

Example of Formatted Output

```
int age = 36;
char[] name = "Kevin";
printf("My name is %s and I am %d years old.\n", name,
    age);
```

- This would print `My name is Kevin and I am 36 years old.`
- The `%s` has been replaced by the value stored in `name` - *Kevin*
- The `%d` has been replaced by the value stored in `age` - *36*
- Note the order of the values in the message match the order they are listed in `printf`

Notes

Questions?

Notes

String Input

- There are a number of ways to capture input in C. Among them are `fgets` and `scanf`. We will focus on the former
- The `fgets` (File GET String) function uses the following form
  - `fgets(string, length, file-stream)`
    - `string` - the location in memory to read into - a `char` array or pointer
    - `length` - the maximum number of characters to read - determined by the size of the area in memory allocated
    - `file-stream` - the location to read data from. It is a `FILE*` (a pointer to a file). `stdin` is the file we will use at the moment. `stdin` stands for STanDard INput, and represents the command line. Therefore, the command line is treated like a file in C.

Notes

String Input: Example

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h> // We include this one for the atoi() function

int main(int argc, const char * argv[])
{
    // Declaring the memory space for our strings
    char name[50];
    char age_str[50];
    int age;

    // Displaying a message and reading from the console
    printf("Please enter your name: ");
    fgets(name, 50, stdin);
    printf("Please enter your age: ");
    fgets(age_str, 50, stdin);

    // Converting the age from string to int
    age = atoi(age_str); // This function is for conversion of string to int

    // Result output
    printf("\nYour name is %s and you are %d years old.\n", name, age);

    return 0;
}
```

Notes

String Input: Example Continued

- The output of this program would look like this:
  - Please enter your name: Kevin
  - Please enter your age: 36
  - Your name is Kevin
  - and you are 36 years old.
- Note that we have also printed the new line captured when we entered our name at the keyboard
- `fgets` captured this character before adding the null terminator. A solution is to change the new line character to a null terminator

Notes

Removing a New Line Character

Removing new line character from a read string

```c
// Gets length of string and replaces new line
int len = strlen(name);
// Check if last character is a new line
if (len > 0 && name[len - 1] == '\n')
{
    // Replaces the new line with a null terminator
    name[len - 1] = '\0';
}
```

Notes

Questions?

Notes

Outline

Notes

Basic Constructs

- The last sample code introduced an `if` statement. We will be looking at `if` statements in more detail in other material delivered in this module.
- The next few slides introduce a number of new concepts that we will visit in more detail in other forms.
- The main aim is to describe the most fundamental aspects of programming
  - sequence - expressions are executed in sequence (one after another) by the computer
  - selection - an expression can be made such that a choice from two or more path ways can be taken. This choice requires the testing of a value to determine the path to take through the code. This is what the `if` statement is
  - iteration - a block of code (collection of expressions) can be executed a number of times. This concept is often called `looping`

Notes

Conditionals

- `if` statements allow a choice to be made between two separate code pathways

`if` statement

```
if (condition)
{
    // This code runs if
    // condition equals true
    // (non-zero in C)
}
else
{
    // This code runs if
    // condition equals false
    // (zero in C)
}
```

- C, unlike modern languages, does not have a *boolean* type, so there are no values `true` and `false`

Notes

Conditional Example

`if` statement example

```
int x = 5;
if (x < 10)
{
    // This code will run
    printf("x is less than 10\n");
}
else
{
    // This code will not run
    printf("x is not less than 10\n");
}
```

Notes

Concatenated `if` Statement

- `if` statements can be joined together to create more choice. For example

Concatenated `if` statement example

```
int x = 10;
if (x < 5)
{
    // This code will not run
    printf("x is less than 10\n");
}
else if (x < 20)
{
    // This code will run
    printf("x is less than 20\n");
}
else
{
    // This code will not run
    printf("x is greater than 20\n");
}
```

Notes

Loops

- C supports a number of loop types. These loops are fairly common across programming languages (especially C based ones)

  `while(condition)` - the commands in the block are executed as long as the condition is true. If it is false at the start then the code block will never run

  `do ... while(condition)` - the commands in the block are executed at least once and will be repeated as long as the condition is true

  `for(value; condition; command)` - `for` loops perform an initialisation (`value`), test for a condition at the start of each iteration (`condition`), and perform a command at the end of each operation (`command`). Usually used to loop a fixed number of times.

Notes

Switch / Case

- case statements allow us to branch on the value of a variable, for example if we were working on a menu with multiple choice

case statement example

```
switch (variable)
{
    case value_1:
        // code
        break;
    case value_2:
        // code
        break;
    default:
        // code
        break;
}
```

Notes

Functions

- Functions in C allow us to structure our program and break our code in different, reusable sections. We already mentioned the general syntax:
  - `return-type function-name(params)`
- Examples
  - `int main(int argc, char **argv)`
  - `char* read_name()`
  - `int add(int a, int b)`
- Notice we try and give our functions names that relate to what they do

Notes

Questions?

Notes

Outline

Notes

Summary

- After this lecture, you should be able to:
  - Create basic programs in C with basic text input and output
  - Compile and execute those programs using a command-line interface on Windows
  - Understand the different steps taken by the compiler
  - Understand the concepts of string input and output in C
- Unit 1 of the workbook covers these areas in more detail. Practice what has been discussed.

Notes

To do...

content...

Notes

Notes

Notes