CS-231 - LESSON 02 - PROGRAMMING IN C

GIUSEPPE TURINI

Kettering
UNIVERSITY

TABLE OF CONTENTS

STUDY GUIDE

Study Material:

- These slides and your notes.
- *"The C Programming Language (2nd Edition)"*, ch. 1-2.

Homework:

- Code from scratch all C programs on these slides.
- Solve all exercises in *"The C Programming Language (2nd Edition)"*, ch. 1-2.

Additional Resources:

- *"The C Programming Language (2nd Edition)."*
- *"GNU - The GNU C Reference Manual"*
- *"Eclipse – C/C++ Development Toolkit (CDT) User Guide"*

INTRODUCTION TO PROGRAMMING IN C

C is a general-purpose computer programming language.
It was created in the 1970s by Dennis Ritchie (see course textbook).

C is used in: OS programming, device drivers, and computer networking.
C is used on different computer architectures: from supercomputers to microcontrollers.

The programming language C is:

- An imperative and procedural language (supporting structured programming).
- It supports static scope (lexical scope) and recursion.
- It includes a static type system.
- It is designed to be compiled for performance and low-level memory access.
- It is designed to encourage cross-platform programming.

GETTING STARTED

The only way to learn a new programming language is by writing programs in it!

The first program to write is the same for all languages (*"hello world"*).

```
// File "006B-HelloWorldInC.c"
#include <stdio.h> // Importing standard input/output library (for "printf" etc.).
int main() { // Entry point.
   printf("Hello, World!\n"); // Standard library function to print text on console.
   fflush(stdout); // Standard library function to force print (otherwise buffered).
   return 0; // Default return value for successful execution.
}
```

Note: In Eclipse CDT, first create a C project, then code a C program for that C project.

GETTING STARTED (2)

Your C program must be coded in a source file with extension *".c"*.

Before being able to run it, a C program must be compiled (see your IDE).
If the compilation is successful (no errors), various types of files are created:

- One or more object files (extension *".o"*) are produced as the initial compiler output.
  They consist of function definitions in binary form, but not executable.
- One binary executable file (extension *".exe"*) produced as the final linked output.
  The linker links together object files to produce 1 binary executable file.

Note: The extension *".exe"* for executable files is a convention on Windows.
Other OS may use a different convention for executable file extension.

Note: There are other file extensions related to C programming (*".h"*, *".dll"*, *".lib"*, on Windows), but these will be discussed later in this course or in other courses (see appendices).

GETTING STARTED (3)

We can start by analyzing our first C program (*"hello world"*):

```c
// File "006B-HelloWorldInC.c"
#include <stdio.h> // Importing standard input/output library (for "printf" etc.).
int main() { // Entry point.
   printf("Hello, World!\n"); // Standard library function to print text on console.
   fflush(stdout); // Standard library function to force print (otherwise buffered).
   return 0; // Default return value for successful execution.
}
```

- The *"main"* function is a special function (program execution starts here).
  Later we will discuss the *"main"* version for command-line arguments.
- The *"#include"* directive tells the compiler to include info about external code.
  In this case, "#include <stdio.h>" includes standard input/output library.
- The *"return"* value from *"main"* represent how the C program exited.
  Returning zero means normal exit, and returning non-zero means abnormal exit.

VARIABLES AND ARITHMETIC EXPRESSIONS

Consider this C program *"Fahrenheit-Celsius table"* to print temperature values.

```c
// File "009A-FahrenheitCelsiusTable.c"
#include <stdio.h>
// Print Fahrenheit-Celsius table for fahr = 0, 20, ... , 300.
int main() {
   int fahr, celsius; // Fahrenheit and Celsius temperature values (declaration only).
   int lower = 0; // Lower limit of temperature table (declaration and definition).
   int upper = 300; // Upper limit of temperature table.
   int step = 20; // Step/increment size.
   fahr = lower; // Initialization.
   // Iterative generation of Fahrenheit temperature values (lower to upper).
   while(fahr <= upper) {
      celsius = 5 * (fahr - 32) / 9;
      printf("%d\t%d\n", fahr, celsius);
      fahr = fahr + step;
   }
   return 0;
}
```

VARIABLES AND ARITHMETIC EXPRESSIONS (2)

These are the main programming elements of the C program *"Fahrenheit-Celsius table"*.

- Data types in C.
  - *"char"*, *"int"*, *"float"*, *"double"*, and *"bool"** (see appendices).
- Selection and iteration statements in C.
  - Standard: *"if-else"*, *"while"*, *"for"*, and *"do-while"*.
- Arithmetic, comparison/relational, logical, and assignment operators in C.
  - Standard (see en.wikipedia.org/wiki/operators_in_C_and_C++).
  - Bitwise, pointer, and other operators will be discussed later in this course.
- Formatting data using *"printf"* function.
  - Complete format placeholder specification (see en.wikipedia.org/wiki/printf).

DECLARATION VS DEFINITION

In computer programming, we should use the terms *"definition"* and *"declaration"* carefully.

- **Definition:** It refers to the place in code where the variable/function is created/initialized (and assigned storage). Definition and declaration can happen at the same time; if not, declaration must happen before definition.

- **Declaration:** It refers to the place in code where the variable/function is given a *"type"* (name and type for variables, name and inputs/outputs for functions). Declaration does not initialize a variable/function, and does not allocate storage for it.

See: en.wikipedia.org/wiki/declaration_(computer_programming)

DATA TYPES IN C

C provides 4 (5) basic arithmetic types: **char**, **int**, **float**, **double**, and **bool***.
Additionally, you can use 4 type modifiers: **signed**, **unsigned**, **short**, and **long**.
This is a partial table listing the most-common basic data types in C:

TYPE	SIZE	RANGE	DESCRIPTION
char	8-bit	[-128, 127]	Could be either signed/unsigned.
int	32-bit	[-2147483648, 2147483647]	Basic signed integer type.
float	32-bit	[1.17549e-38, 3.40282e+38]	Single-precision floating-point type.
double	64-bit	[2.22507e-308, 1.79769e+308]	Double-precision floating-point type.
bool	8-bit	{0, 1}	Included in C-99/ANSI-C (stdbool.h).

Note: The complete list of all permissible combinations is included in the appendices.

Note: See the *"data types in C"* code example to query data type information on your platform.

See: en.wikipedia.org/wiki/c_data_types

CONSTANTS IN C

Consider this C program *"Fahrenheit-Celsius table with for and constants"*.

```c
// File "015A-FahrenheitCelsiusTableWithSymbolicConstants.c"
#include <stdio.h>
// Print Fahrenheit-Celsius table for fahr = 0, 20, ... , 300
int main() {
   // Constants.
   const int LOWER = 0; // Lower limit of temperature table (constant).
   const int UPPER = 0; // Upper limit of temperature table (constant).
   const int STEP = 0; // Step/increment size (constant).
   int fahr; // Fahrenheit temperature integer value.
   // Iterative (for) generation of Fahrenheit temperature values (lower to upper).
   for(fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP) {
      printf("%3d\t%6.1f\n", fahr, (5.0F / 9.0F) * (fahr - 32));
   }
   return 0;
}
```

SYMBOLIC CONSTANTS (MACROS) IN C

Consider this C program *"Fahrenheit-Celsius table with for and symbolic constants"*.

```c
// File "015A-FahrenheitCelsiusTableWithSymbolicConstants.c"
#include <stdio.h>
// Symbolic constants (macros).
#define   LOWER   0     // Lower limit of temperature table (macro).
#define   UPPER   300   // Upper limit of temperature table (macro).
#define   STEP    20    // Step/increment size (macro).
// Print Fahrenheit-Celsius table for fahr = 0, 20, ... , 300
int main() {
   int fahr; // Fahrenheit temperature integer value.
   // Iterative (for) generation of Fahrenheit temperature values (lower to upper).
   for(fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP) {
      printf("%3d\t%6.1f\n", fahr, (5.0F / 9.0F) * (fahr - 32));
   }
   return 0;
}
```

SYMBOLIC CONSTANTS (MACROS) IN C (2)

These are the main elements of *"Fahrenheit-Celsius table with for and symbolic constants"*.

- Iteration statement *"for"* in C.

- Symbolic constants using *"#define"* directive in C.
  - Preprocessor directive to define macros.
  - A **macro** is a name that is replaced in the code by a value (or a set of instructions!) before the compilation process begins.
  - No semicolon (;) at the end of a *"#define"* line.

See: learn.microsoft.com/en-us/cpp/preprocessor/hash-define-directive-c-cpp

CHARACTER INPUT/OUTPUT IN C

In C, text input/output (regardless of origin or destination) is handled as a stream.

A text stream is a sequence of characters divided into lines; each line consisting of 0 or more characters followed by a newline character (*'\n'*).

The standard library (in C) includes functions for reading/writing 1 character at a time.

Consider this C program *"copy input to output"*.

```c
// File "016A-CopyInputToOutput.c"
#include <stdio.h>
int main() { // Copy input to output.
   int c; // int type so it can store any value getchar returns (char type is not enough).
   c = getchar(); // getchar returns next character from text stream (usually keyboard).
   while(c != '\n') { // Checking for special character newline ('\n').
      putchar(c); // putchar writes input value in output (usually on screen/console).
      c = getchar(); }
   return 0; // Normal exit.
}
```

ARRAYS IN C

Consider this C program *"Counting digits, and other characters"*.

```c
// File "022A-CountingDigitsAndCharacters.c"
#include <stdio.h>
int main() { // Counts digits and other characters.
   int currChar; // Temporary variable to store current character being read (declaration).
   int numDigits[10]; // Array to store digit counters (declaration).
   int numNonDigits = 0; // Counter for non-digit characters (definition).
   for(int i = 0; i < 10; i++) { numDigits[i] = 0; } // Init array numDigits.
   // Counting digits and non-digits until reaching newline character.
   while((currChar = getchar()) != '\n') {
      if((currChar >= '0') && (currChar <= '9')) { ++numDigits[ currChar - '0' ]; }
      else { numNonDigits++; } }
   // Printing counters.
   printf("Digit counters:");
   for(int i = 0; i < 10; i++) { printf(" %d", numDigits[i]); }
   printf("\nNon-digit counter: %d\n", numNonDigits);
   return 0; // Normal exit.
}
```

FUNCTIONS IN C

Consider this C program implementing a *"Simple power function"*.

```c
// File "024A-SimplePowerFunction.c"
#include <stdio.h>
int power(int base, int exp); // Simple power function (declaration).
int main() { // Testing power function.
   // Testing.
   printf("Testing simple power function:\n");
   for(int i = 0; i < 10; i++) {
      printf("i = %d, \t2^i = %d, \t-3^i = %d\n", i, power(2, i), power(-3, i)); }
   return 0; // Normal exit.
}
// Simple power function (definition). Supports only exponents >0, and small integers.
int power(int base, int exp) {
   int res = 1; // Init result.
   // Iteratively multiplications to compute power (base^exp).
   for(int i = 1; i <= exp; i++) { res = res * base; }
   return res; // Return result (base^exp).
}
```

FUNCTIONS IN C (2)

These are the main elements of *"Simple power function"*.

- Function declarations (function prototypes) specify: function name, formal parameter types and names (local), and return type.
  Remember the difference between *"formal parameter"* and *"actual argument"*.

- Function definitions can appear in any order, and in different source files.
  See also the *"#include"* directive.

- Functions do **not** need to return a value.
  See *"return"* statement.
  Calling code can ignore the value returned by a function call.

See: en.wikipedia.org/wiki/parameter_(computer_programming)

See: learn.microsoft.com/en-us/cpp/preprocessor/hash-include-directive-c-cpp

See: learn.microsoft.com/en-us/cpp/c-language/return-statement-c

FUNCTIONS IN C (3)

These are the main elements of *"Simple power function"* (continued).

- In C, all function call actual arguments are always passed *"by value"*.
  Opposite of passing arguments *"by reference"* available in other languages.
  See also *"how to use pointers"* to pass arguments *"by reference"* in C.

- In C, only arrays are passed to functions *"by reference"*!
  Whenever an array is passed to a function as an input argument (or return value), the value passed to the function is the *"memory address"* (pointer) of the beginning of the array (pointer at array element at index 0).
  There is no copying of array elements, when an array is passed to (or from) a function!

VARIABLES AND SCOPE IN C

So far, all the variables used were local variables...

Local Variable (Automatic Variable): A local variable in a function comes into existence only when the function is called, and disappears when the same function exits.

Local variables are also called automatic variables.

Automatic variables are created and destroyed with function invocation, they do not retain their values from one function call to the next, so they must be explicitly initialized each time before being used.

See: en.wikipedia.org/wiki/automatic_variable

See: *"GNU C Language Intro and Reference Manual - Local Variables"*

See: learn.microsoft.com/en-us/cpp/c-language/c-storage-classes

VARIABLES AND SCOPE IN C (2)

As an alternative to automatic (local) variables, it is possible to define variables that are *"external"* to all functions: variables accessible (by name) by any functions.

External Variable: An external variable must be defined, once, outside of any function.

An external variable must also be declared in each function that wants to access it. This declaration may be an explicit *"extern"* statement, or may be implicit from context.

Before a function can use an external variable, the variable name must be made known to the function (explicitly, using the *"extern"* keyword).

In certain cases, the *"extern"* declaration can be omitted (is implicit): if the external variable is defined in the source file before its use in a function. Usually all *"extern"* definitions are at the top of source files.

See: en.wikipedia.org/wiki/external_variable

See: *"GNU C Language Intro and Reference Manual - Extern Declarations"*

VARIABLES AND SCOPE IN C (3)

Consider this C program for *"Processing char arrays using external variables"*.

```c
// File "032A-ProcessingCharacterArraysUsingExternalVariables.c"
#include <stdio.h>

#define MAXLINE 1000 // Macro: Max num chars in text line.

// External variables.
int max; // Max length seen so far...
char line[MAXLINE]; // Current input text line.
char longest[MAXLINE]; // Longest line seen so far.

// Function declarations (prototypes).
int getline(char l[]); // Input text stored in external var (line) returning its size.
void copy(void); // Copy external var (line) into external var (longest).

...
```

VARIABLES AND SCOPE IN C (4)

Consider this C program for *"Processing char arrays using external variables"* (continued).

```
// File "032A-ProcessingCharacterArraysUsingExternalVariables.c"
...
int main(int argc, char *argv[]) { // Select and print longest text line in input.
   int len; // Local (automatic) var: length of current text line.
   extern int max; // External var.
   extern char longest[]; // External var.
   max = 0; // Init external var.
   // Convert all run (command line) arguments into internal char arrays to find longest.
   for(int i = 1; i < argc; i++) { // Note: Skipping 1st run arg (executable full path).
      len = getline(argv[i]); // Convert current run arg into internal char array (line).
      if(len > max) { // If current line is longer than current longest, update longest.
         max = len; copy(); } }
   if(max > 0) { printf("Longest text line is: %s\n", longest); } // Check validity.
   return 0; // Normal exit.
}
...
```

VARIABLES AND SCOPE IN C (5)

Consider this C program for *"Processing char arrays using external variables"* (continued).

```
// File "032A-ProcessingCharacterArraysUsingExternalVariables.c"
...
int getline(char l[]) { // Stores input line in external var (line) returning its size.
   int c = 0; // Current character being read.
   int i = 0; // Iteration variable scope includes entire function!
   extern char line[]; // External var.
   // Reading input line 1 char at time, storing line in external var (line).
   c = l[i];
   while(c != '\0') {
      line[i] = c;
      i++;
      c = l[i]; }
   line[i] = '\0'; // Adding null character at end of text line.
   return i; // Returning length of text line read.
}
...
```

VARIABLES AND SCOPE IN C (6)

Consider this C program for *"Processing char arrays using external variables"* (continued).

```
// File "032A-ProcessingCharacterArraysUsingExternalVariables.c"
...
void copy(void) { // Copies external var (line) into external var (longest).
   int i = 0; // Iteration variable.
   extern char line[]; // External var.
   extern char longest[]; // External var.
   // Copy characters from (from) to (to) until reaching null character (included).
   while((longest[i] = line[i]) != '\0') { i++; }
}
```

VARIABLES AND SCOPE IN C (7)

These are the main elements of *"Processing char arrays using external variables"*.

- External variables declaration, use, and initialization.
  For external variables declared in different source files, see header files.

- Function declaration (prototype), and definition and external variables.
  See also the *"extern"* keyword.

- Run (command-line) arguments declaration, and use.
  See the different *"main"* function declaration.
  See the configuration of run (command-line) arguments in your IDE.

HEADER FILES IN C

A header file is a file of C code (typically containing C declarations and macro definitions), to be shared between several source files. You request the use of a header file in your program by including it (using the C preprocessing directive *"#include"*). Header files serve 2 purposes:

- **System header files** declare interfaces to parts of OS. Include them in your program to supply definitions and declarations needed to invoke OS calls and libraries.
- **Program-specific header files** contain declarations for interfaces between the source files of a particular program. Create a header file for related declarations and definitions if most of them are needed in multiple different source files.

Including a header file is the *"same"* as copying the header file into the source file!

See: en.wikipedia.org/wiki/Include_directive

See: *"GNU C Language Intro and Reference Manual - Header Files"*

HEADER FILES IN C (2)

In C, the standard is to use extension *".h"* for header files.

Note: In C, including another source file is **not** limited to code usually stored in header files. You can store any C code into a separate source file, and then use *"#include"* directive.

You can use the "#include" directive in 2 ways:

- **Including using <>:** Mainly used to access system header files located in the standard system directories (for example: *"#include <stdio.h >"*). Including an header file using angular brackets (<>), the preprocessor uses predetermined directory paths to search and then access the header file.
- **Including using "":** Mainly used to access program-specific header files located in program-related directories (for example: *"#include "./Include/MyHeader.h""*). Including an header file using double quotes (""), the preprocessor accesses the file using the directory as specified (using relative or absolute path).

ECLIPSE CDT FOR C PROGRAMMING

This section describes some common tasks using the official course IDE for C/C++ programming *"Eclipse IDE for C/C++ Developers"* (also known as *"Eclipse CDT"*).

- Creation of your first C project.
- Compile a C project.
- Configure executable names of a C project.
- Setting run (command-line) arguments of a C project.
- Inspecting variables at runtime (expanding objects/ADTs, checking references, etc.).

See: help.eclipse.org

FIRST C PROJECT IN ECLIPSE CDT

To create a new C project in Eclipse CDT:

- *"File"* menu → *"New"* → *"Project"* → *"C/C++"* → *"C Project"* → *"Next"*.
- Set the project name and folder.
- Select *"Empty Project"* as project type, and *"MinGW GCC"* as compiler (toolchains).
- Click *"Finish"*.
- Now your C project should appear (opened) in the *"Project Explorer"* sidebar.

To create a new C source file in your (currently selected) C project in Eclipse CDT:

- Right-click on the (open) project, using the *"Project Explorer"* sidebar.
- *"New"* → *"Source File"*.
- Select the folder (project folder), file name (extension .c), and click *"Finish"*.
- Now your C source file is ready to be edited, and you can start programming C code.

COMPILE C PROJECTS IN ECLIPSE CDT

To compile (build) an open C project in Eclipse CDT:

- Right-click on the (open) project, using the *"Project Explorer"* sidebar.
- *"Build Configurations"* → *"Set Active"* → select *"Debug"* or *"Release"*.
- Right-click on the (open) project, using the *"Project Explorer"* sidebar.
- *"Build Configurations"* → *"Build Selected..."*.
- The *"Console"* panel shows the result of the compilation (build).

- If the compilation (build) fails, errors and warnings are listed on the console.
- Before compiling again, fix all errors following the tips provided by the compiler.

- If the compilation (build) is successful, an executable is generated.
- The location and name of the executable file can be configured (see appendices).

SET EXECUTABLE NAMES IN ECLIPSE CDT

To change the executable file names in Eclipse CDT:

- Right-click on the (open) project, using the *"Project Explorer"* sidebar.
- *"Properties"* → *"C/C++ Build"* → *"Settings"* → *"Build Artifact"*.
- Select the configuration you want to configure (*"Debug"* or *"Release"*).

- Then you can configure the executable name.
- For example: *"${ProjName}-${ConfigName}"*.

SET COMMAND-LINE ARGS IN ECLIPSE CDT

To set the command-line arguments for a C program in Eclipse CDT:

- Right-click on the (open) project, using the *"Project Explorer"* sidebar.
- *"Run As"* → *"Run Configurations..."* (or *"Debug As"* → *"Debug Configurations..."*).
- Select the proper *"C/C++ Application"* and then its *"Arguments"* tab.
- Write your command-line arguments in the *"Program arguments:"* editbox.

Check your configuration using this C program (*"echo run args"*).

```c
#include <stdio.h> // Importing external code for "printf" etc.
// Echoes command-line arguments.
int main(int argc, char *argv[]) { // Or: int main(int argc, char **argv).
   for(int i = 0; i < argc; i++) { printf("argv[%d]: %s\n", i, argv[i]); }
   return 0;
}
```

Note: In C, the first command line argument is always the full path of the executable file.

REFERENCES

- *"Wikipedia - C Programming Language"*
- *"GeeksforGeeks - Types of C Files After Compilation"*
- *"Wikipedia - C Data Types"*

- *"GNU - The GNU C Reference Manual"*
- *"Microsoft - C Language Reference"*

- *"Nanyang Technological University – How To Install Eclipse for C/C++ Programming"*
- *"Eclipse – C/C++ Development Toolkit (CDT) User Guide"*

APPENDICES

- Static and Dynamic Scope.
- Static and Dynamic Type Checking.
- Types of C Files.
- Dynamic and Static Libraries.
- Data Types in C (Integer Types, Floating-Point Types, and Boolean Types).

STATIC AND DYNAMIC SCOPE

In languages with static scope (lexical scope):

- The name resolution depends on the location in the source code and the static context (lexical context): defined by where the named variable or function is defined.

In languages with dynamic scope:

- The name resolution depends upon the program state when the name is encountered which is determined by the execution context (runtime context, calling context, dynamic context).

Example: With static scope, a name is resolved searching the local static context; if it fails, searching the outer static context, and so on.

With dynamic scope, a name is resolved searching the local execution context; if it fails, searching the outer execution context, and so on (up the call stack).

See: en.wikipedia.org/wiki/scope_(computer_science)

STATIC AND DYNAMIC SCOPE (2)

Most modern languages use static scope (lexical scope) for variables and functions.

Dynamic scope is used in some languages (e.g., Lisp, etc.).

Other languages (e.g., Perl) offer both lexical and dynamic scope.

Static resolution (lexical resolution) can be determined at compile time (early binding).

Dynamic resolution can only be determined at run time (late binding).

See: en.wikipedia.org/wiki/scope_(computer_science)

STATIC AND DYNAMIC TYPE CHECKING

Type checking is the process of verifying and enforcing the constraints of types.
It may occur at compile time (static type checking) or at runtime (dynamic type checking).

A programming language is considered *"strongly typed"* if it strongly enforces its typing rules (allowing only lossless automatic type conversions). Otherwise, it is *"weakly typed."*

Note: Many languages with static type checking provide a way to bypass the type checker.

Some languages allow coders to choose between static and dynamic type checking.
C# (4.0) includes the *"dynamic"* keyword to declare variables to be checked dynamically.

Other languages allow writing code that is not type-safe.
In C, coders can cast a value between any 2 types that have the same size.

See: en.wikipedia.org/wiki/type_system

TYPES OF C FILES

Programming in C involves different file types: some store source code, others are generated:

- **Source File (extension *".c"*):** Stores source code (functions and code logic), and is human readable.
- **Header File (extension *".h"*):** Stores function prototypes and pre-processor statements, allows source code to access externally-defined functions, and is human readable.
- **Object File (extension *".o"*):** Generated by the *"compiler"*, stores function definitions in binary format (but not executable), and is not human readable. The *"linker"* links object files to generate a binary executable file (extension *".exe"* on Windows).
- **Binary Executable File (extension *".exe"*, on Windows):** Generated by the *"linker"* that links together object files, is executable standalone, and is not human readable.

See: www.geeksforgeeks.org/types-of-c-files-after-its-compilation

DYNAMIC AND STATIC C LIBRARIES

Programming in C involves different library files:

- **Dynamic Library File (extension *".dll"*, on Windows):** File storing a library dynamically linked at runtime by the executable.

  Extensions are: *".dll"* on Windows, *".so"* in Linux, and *".dylib"* in OSX.

  They are not included inside the executable file, can be shared among executables (*"shared libraries"*), and can be updated without requiring a rebuild of the executables.

- **Static Library File (extension *".lib"*, on Windows):** File storing a library statically linked at compilation time (linking) by the *"linker"*.

  Extensions are: *".lib"* on Windows, *".a"* in Linux.

  They are included inside the executable file, allow better performance than dynamic libraries, but any update requires a rebuild of the executable.

See: www.geeksforgeeks.org/types-of-c-files-after-its-compilation

INTEGER TYPE CHAR IN C

These are all the admissible combinations for the *"char"* data type in C:

TYPE	SIZE	RANGE	DESCRIPTION
char	8-bit	[-128, 127]	Could be either signed/unsigned.
signed char	8-bit	[-128, 127]	Signed char.
unsigned char	8-bit	[0, 255]	Unsigned char.

Note: See the *"data types in C"* code example to query data type information on your platform.

See: en.wikipedia.org/wiki/c_data_types

INTEGER TYPE SHORT IN C

These are all the admissible combinations for the *"short"* data type in C:

TYPE	SIZE	RANGE	DESCRIPTION
short	16-bit	[-32768, 32767]	Short signed integer type.
short int	16-bit	[-32768, 32767]	Short signed integer type.
signed short	16-bit	[-32768, 32767]	Short signed integer type.
signed short int	16-bit	[-32768, 32767]	Short signed integer type.
unsigned short	16-bit	[0, 65535]	Short unsigned integer type.
unsigned short int	16-bit	[0, 65535]	Short unsigned integer type.

Note: See the *"data types in C"* code example to query data type information on your platform.

See: en.wikipedia.org/wiki/c_data_types

INTEGER TYPE INT IN C

These are all the admissible combinations for the *"int"* data type in C:

TYPE	SIZE	RANGE	DESCRIPTION
int	32-bit	[-2147483648, 2147483647]	Basic signed integer type.
signed	32-bit	[-2147483648, 2147483647]	Basic signed integer type.
signed int	32-bit	[-2147483648, 2147483647]	Basic signed integer type.
unsigned	32-bit	[0, 4294967295]	Basic unsigned integer type.
unsigned int	32-bit	[0, 4294967295]	Basic unsigned integer type.

Note: See the *"data types in C"* code example to query data type information on your platform.

See: en.wikipedia.org/wiki/c_data_types

INTEGER TYPE LONG IN C

These are all the admissible combinations for the *"long"* data type in C:

TYPE	SIZE	RANGE	DESCRIPTION
long	32-bit	[-2147483648, 2147483647]	Long signed integer type.
long int	32-bit	[-2147483648, 2147483647]	Long signed integer type.
signed long	32-bit	[-2147483648, 2147483647]	Long signed integer type.
signed long int	32-bit	[-2147483648, 2147483647]	Long signed integer type.
unsigned long	32-bit	[0, 4294967295]	Long unsigned integer type.
unsigned long int	32-bit	[0, 4294967295]	Long unsigned integer type.

Note: See the *"data types in C"* code example to query data type information on your platform.

See: en.wikipedia.org/wiki/c_data_types

INTEGER TYPE LONG LONG IN C

These are all the admissible combinations for the *"long long"* data type in C:

TYPE	SIZE	RANGE	DESCRIPTION
long long	64-bit	[-9223372036854775808, 9223372036854775807]	...
long long int	64-bit	[-9223372036854775808, 9223372036854775807]	...
signed long long	64-bit	[-9223372036854775808, 9223372036854775807]	...
signed long long int	64-bit	[-9223372036854775808, 9223372036854775807]	...
unsigned long long	64-bit	[0, 18446744073709551615]	...
unsigned long long int	64-bit	[0, 18446744073709551615]	...

Note: See the *"data types in C"* code example to query data type information on your platform.

See: en.wikipedia.org/wiki/c_data_types

FLOATING-POINT AND BOOLEAN TYPES IN C

These are all the floating-point and boolean data types in C:

TYPE	SIZE	RANGE	DESCRIPTION
float	32-bit	[1.17549e-38, 3.40282e+38]	Single-precision floating-point type.
double	64-bit	[2.22507e-308, 1.79769e+308]	Double-precision floating-point type.
long double	128-bit	[2.80027e-312, 2.80027e-312]	Extended-precision floating-point type.
bool	8-bit	{0, 1}	Included in C-99/ANSI-C (stdbool.h).

Note: See the *"data types in C"* code example to query data type information on your platform.

See: en.wikipedia.org/wiki/c_data_types