Started on	Sunday, September 11, 2022, 9:31 PM
State	Finished
Completed on	Sunday, September 11, 2022, 9:31 PM
Time taken	21 secs
Grade	**31.00** out of 31.00 (**100**%)

Question **1**

Correct

1.00 points out of 1.00

The exact sequence of compilation stages processed by a single invocation of `gcc` is:

Select one:

- preprocessing, compiling proper, assembling, linking ✔
- preprocessing, assembling, linking, compiling proper
- preprocessing, assembling, compiling proper, linking
- preprocessing, compiling proper, linking, assembling

Your answer is correct.

The correct answer is: preprocessing, compiling proper, assembling, linking

Question **2**

Correct

1.00 points out of 1.00

The original ANSI (American National Standards Institute) C standard was ratified in 1989 and published in 1990. This standard was ratified as an ISO (International Standards Organization) standard in 1990. There are no differences between these standards and they are commonly known as C89 and sometimes C90. A second major revision occurred in 1995 and this standard is known as C94 and sometimes C95. A third revision occurred in 1999 and is commonly known as C99. A fourth version of the standard was published in 2011 and it is commonly referred to as C11. A fifth version of the standard was published in 2018 and is known as C18.

This course will be using C11 and concentrate on the features compatible with C++ and neglect those incompatible with C++. By default `gcc` doesn't compile for the C11 standard. Which option(s) must be specified for `gcc` to compile for the C11 standard? You must provide all correct options for full credit.

Select one or more:

- `-std=c11` ✔
- `-pedantic-errors` ✔
- `-std=c99`
- `-Wstrict-prototypes` ✔
- `-ansi`
- `-std=c89`

Your answer is correct.

The correct answers are: `-std=c11`, `-pedantic-errors`, `-Wstrict-prototypes`

Question **3**

Correct

1.00 points out of 1.00

Which option should be used with `gcc` for verbose compilation so that detailed information about the exact sequence of commands used to compile and link a program is displayed? Use **this** `gcc` reference page to determine the correct answer.

Select one:

- `-v` ✔
- `-o`
- `-c`
- `-E`
- `-S`

Your answer is correct.

The correct answer is: `-v`

Question **4**

Correct

1.00 points out of 1.00

Which option should be used with `gcc` to compile source files to the C99 standard? Use **this** `gcc` reference page to determine the correct answer.

Select one:

- `-pedantic`
- `-std=c89`
- `-std=c99` ✔
- `-std=c90`
- `-std=c11`
- `-ansi`

Your answer is correct.

The correct answer is: `-std=c99`

Question **5**

Correct

1.00 points out of 1.00

Compile and link the following source file using the C11 standard and write the exact result displayed when the executable program is run.

```c
#include <stdio.h> // prototype for printf
void foo(int *pa, int *pb); // function prototype

int main(void) {
  int num1 = 0x11, num2 = 0x77;

  foo(&num1, &num2);
  printf("%d,%d", num1, num2);

  return 0;
}

inline void foo(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}

```

Answer: 119,17 ✔

The correct answer is: 119,17

Question **6**

Correct

1.00 points out of 1.00

How many different ways can you write a comment in C11?

Select one:

- 3
- 1
- 2 ✔
- 4
- 0
- 5

Your answer is correct.

The correct answer is: 2

Question **7**

Correct

1.00 points out of 1.00

ISO C standards require the preprocessor to replace comments in a source file with single spaces. Make sure to read **this** reference page before answering the question.

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **8**

Correct

1.00 points out of 1.00

How many valid comments does the following C11 source file contain? Make sure to read **this** reference page before answering the question.

```c
// program to compute the square of the first 10 integers
#include <stdio.h>

int main(void /* no arguments */) {
  /*
  //Loop from 1 to 10, printing out the squares
  */
  for (int i = 1; i <= 10; ++i) {
    printf("%d //square is// %d\n", i, i*i);
  }

  return 0;
}

```

Answer: 3 ✔

The correct answer is: 3

Question **9**

Correct

1.00 points out of 1.00

Which option should be used with GNU C compiler **`gcc`** to *only* compile a source file into an object file?

Select one:

- **`-c`** ✔
- **`-S`**
- **`-o`**
- **`-W`**
- **`-E`**

Your answer is correct.

The correct answer is: **`-c`**

Question **10**

Correct

1.00 points out of 1.00

Lectures have emphasized the idea that *compiling is a process associated with an individual source file* while *linking is a collective process requiring the linker to process all necessary object files to generate an executable*.

Suppose we'd like to build a program consisting of two source C files *greeting.c* and *main.c*. The file *greeting.c* consists of these lines:

```
/* greeting.c */
#include <stdio.h> // prototype for function printf

void greeting(void) {
  printf("Hello World!\n");
}

```

Compile but not link *greeting.c* for C11 with the full suite of options. See class handouts on the entire list of options that you must enable.

Source file *main.c* consists of these lines:

```
/* main.c */
int main(void) {
  greeting();
  return 0;
}

```

main.c implements function **`main`** which in turn calls a function **`greeting`** that is implemented elsewhere (in source file *greeting.c* which the compiler is unaware of). Compile but not link *main.c* for C11 with the full suite of options enabled. Does source file *main.c* successfully compile?

Select one:

- True
- False ✔

The correct answer is 'False'.

Question **11**

Correct

1.00 points out of 1.00

The source file *main.c* from the previous question (see below) will not compile - instead the compiler terminates with the error message: "*implicit declaration of function 'greeting'*".

```
1  /* main.c */
2  /*
3  We're declaring the function main because we're
4  providing the compiler the following information:
5  1) name of the function (main),
6  2) type of value returned by the function (int),
7  3) number of parameters or inputs (zero - that's
8     what keyword void in parameter list means), and
9  4) type of the parameter (void)
10 */
11 int main(void) {
12   greeting();
13   return 0;
14 }
15
```

The compiler signals an error because function **`main`** is using an identifier **`greeting`** that is not known to the compiler thereby preventing it from ensuring that this unknown entity called **`greeting`** is being correctly used in function **`main`**. This means that programmers must *declare* all identifiers to the compiler before their first use in a function.

A *function declaration* introduces a reference to a function implemented elsewhere. The function declaration makes known to the compiler the name and return type of the function along with the number and type of inputs (or, in programming terms *parameters*) received by the function. This meaning associated with a function declaration is known as a *function prototype* in C to distinguish from older C standards where the function declaration only specifies the function name and not necessarily the number and types of parameters and the return type. However, C++ being more strictly typed than C, doesn't distinguish between function declaration and function prototype. To be consistent across multiple semesters, we'll use the term function declaration in C to mean function prototype.

Once the compiler has "seen" the declaration, it can ensure the function is being used correctly in subsequent text in the source file. For example, consider a function **`foo`** that requires three integer **`int`** parameters and it is being called with only two **`int`** inputs. If the compiler has seen the function's declaration, it would be able to flag the call as an incorrect use of **`foo`**.

Function **`greeting`** is declared in file *greeting.c*.

```
1  /* greeting.c */
2
3  #include <stdio.h> // declares function printf
4
5  /*
6  We're declaring the function greeting because we're
7  providing the compiler the following information:
8  1) name of the function (greeting),
9  2) type of value returned by function (void - that is,
10    the function doesn't return a value),
11 3) number of parameters or inputs (zero - that's what
12    keyword void in parameter list means), and
13 4) type of the parameter (void)
14 */
15 void greeting(void) {
16   printf("Hello World!\n");
17 }
18
```

Similarly, function **`main`** is declared in *main.c*. Notice that *main.c* doesn't contain a declaration of function **`greeting`**:

```
1  /* main.c */
2  /*
3  We're declaring the function main because we're
4  providing the compiler the following information:
5  1) name of the function (main),
6  2) type of value returned by the function (int),
7  3) number of parameters or inputs (zero - that's
8  what keyword void in parameter list means), and
9  4) type of the parameter (void)
10 */
11 int main(void) {
12   greeting();
13   return 0;
14 }
15
```

Based on this discussion, what change would you make to *main.c* so that it can be successfully compiled? Make sure to test your hypothesis by ensuring that your change enabled you to successfully compile *main.c* and generate a corresponding output object file *main.o*?

Select one:

- In *main.c*, add a function declaration (prototype) for function **`greeting`** before its first use in function **`main`** ✔
- Restrict all C programs from having multiple functions; instead all code is inserted into function **`main`**
- Restrict all C programs to a single source file

Your answer is correct.

The correct answer is: In *main.c*, add a function declaration (prototype) for function **`greeting`** before its first use in function **`main`**

Question **12**

Correct

1.00 points out of 1.00

Recall that a ***function declaration*** introduces a reference to a function object implemented elsewhere. The function declaration makes known to the compiler the name and type of the function along with the number and type of parameters received by the function. This allows the compiler to determine if the function is being referenced correctly in the subsequent text in the source file. For example, consider a function **`foo`** that requires three **`int`** parameters and it is being called with only two **`int`** inputs. Since the compiler has seen the function's declaration, it will be able to flag the call as an error because of the incorrect use of **`foo`**.

Now, consider the updated source file *main.c*:

```c
/* main.c */

/* We're declaring the function greeting because we're
providing the compiler the following information:
1) name of the function (greeting),
2) type of value returned by the function (void -
   that is, the function doesn't return a value),
3) number of parameters or inputs (zero - that's what
   keyword void in parameter list means), and
4) type of the parameter (void)
*/
void greeting(void);

int main(void) {
  greeting();
  return 0;
}

```

And, consider the source file *greeting.c*:

```c
#include <stdio.h> // declares function printf

/*
We're declaring the function greeting because we're
providing the compiler the following information:
1) name of the function (greeting),
2) type of value returned by the function (void -
   that is, the function doesn't return a value),
3) number of parameters or inputs (zero - that's
   what keyword void means), and
4) type of the parameter (void)
*/
void greeting(void) {
  printf("Hello World!");
}

```

Do you agree that function **`greeting`** is being declared in both source files?

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **13**

Correct

1.00 points out of 1.00

Although function **greeting** was declared in both source files *main.c* and *greeting.c*, there is a difference between the two declarations. Function **greeting** is implemented in source file *greeting.c* while *main.c* contains only a declaration. Notice the implementation of function **greeting** in file *greeting.c*:

```c
/* greeting.c */
#include <stdio.h> // declares function printf

/*
We're implementing the function greeting() because we're
providing the compiler the following information:
1) name of the function (greeting),
2) type of value returned by the function (void -
   that is, the function doesn't return a value),
3) number of parameters or inputs (zero - that is what
   keyword void in parameter list means), and
4) type of the parameter (void)
5) the implementation details or the body of the function
*/
void greeting(void) {
  printf("Hello World!\n");
}

```

In source file *main.c*, function **main** is *implemented* while function **greeting** is *declared*:

```c
/* main.c */

/*
We're declaring the function greeting because we're
providing the compiler the following information:
1) name of the function (greeting),
2) type of value returned by the function (void -
   that is, the function doesn't return a value),
3) number of parameters or inputs (zero - that is
   what keyword void in parameter list means), and
4) type of the parameter (void)
*/
void greeting(void);

/*
We're implementing the function main because we're
providing the compiler the following information:
1) name of the function (main),
2) type of the function (int),
3) number of parameters or inputs (one), and
4) type of the parameter (void - that is what
   keyword void in parameter list means)
5) implementation details or body of the function
*/
int main(void)
  greeting();
  return 0;
}

```

Function *implementation* means that memory will be reserved for the instructions of that function.
A function *declaration* simply announces to the compiler the name and return type of the function as well as the number and type of the function's inputs. We need terminology to distinguish between these two declarations. The term **definition** is introduced to distinguish a function's implementation from its declaration. We can then differentiate the meaning behind the terms *function declaration* and *function definition* by suggesting that ***function declaration plus function implementation equals function definition***.

From the above discussion, would you agree that in source file *main.c*, function **greeting** is being ***declared*** but not ***defined***?

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **14**

Correct

1.00 points out of 1.00

Recall that a function declaration is just a reference to a function object defined (that is, implemented) elsewhere. The function declaration makes known to the compiler the name and return type of the function along with the number and type of inputs (or, in programming terms ***parameters***) received by the function. This allows the compiler to determine if the function is being referenced correctly in the subsequent text in the source file.

Since a declaration doesn't actually implement a function but simply tells the compiler the type and name of the function implemented elsewhere, can a source file have multiple declarations of the same function with redundant declarations being ignored by the compiler?

An easy way to answer this question is by inserting additional declarations of function **`greeting`** in *main.c* and then compiling *main.c*.

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **15**

Correct

1.00 points out of 1.00

You've successfully compiled source files *main.c* and *greeting.c* to object files *main.o* and *greeting.o*, respectively. Which of the following commands invoke only the linker so that object files *main.o* and *greeting.o* can be linked to create executable file *test.out*?

Select one:

- `gcc -c main.o greeting.o test.out`
- `gcc -std=c11 -c main.o greeting.o -o test.out`
- `gcc -std=c11 -pedantic-errors -c main.c greeting.c -o test.out`
- `gcc -std=c11 main.c greeting.c -c test.out`
- `gcc main.o greeting.o -o test.out` ✔

Your answer is correct.

The correct answer is: `gcc main.o greeting.o -o test.out`

Question **16**

Correct

1.00 points out of 1.00

Now, let's run a coding experiment. We leave the code in source file *greeting.c* unchanged:

```c
/* greeting.c */
#include <stdio.h> // declares function printf

void greeting(void) {
  printf("Hello World!\n");
}

```

Now, we modify source file *main.c* by adding the definition of function **`greeting`** and also including **`<stdio.h>`**:

```c
/* main.c */

// the original declaration of greeting
void greeting(void);

// main's definition of greeting
#include <stdio.h>

// definition of function greeting
void greeting(void) {
  printf("Greetings from main!");
}

int main(void) {
  greeting();
  return 0;
}

```

Are you able to compile (only) source file *greeting.c* and separately compile (only) source file *main.c*?

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **17**

Correct

1.00 points out of 1.00

Leave the code in source file *greeting.c* from the previous question unchanged:

```c
/* greeting.c */
#include <stdio.h> // declares function printf

void greeting(void) {
  printf("Hello World!\n");
}

```

Recall that the previous question modified source file *main.c* by adding the definition of function **`greeting`** and also including **`<stdio.h>`**:

```c
/* main.c */

/* the original declaration of greeting */
void greeting(void);

/* main's definition of greeting */
#include <stdio.h>

void greeting(void) {
  printf("Greetings from main!");
}

int main(void) {
  greeting();
  return 0;
}

```

It is possible to individually compile (only but not link) both source files:

```
gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c main.c -o main.o
gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c greeting.c -o greeting.o
```

Try to individually compile both source files without the extra typing by adding both filenames to the command-line:

```
gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c main.c greeting.c
```

Each source file will be successfully compiled to their corresponding object files. By default, the compiler will name the object files *main.o* and *greeting.o*.

Are you able to create an executable file by linking together object files *main.o* and *greeting.o*?

Select one:

- True
- False ✔

The correct answer is 'False'.

Question **18**

Correct

1.00 points out of 1.00

The linker is unable to link object files *main.o* and *greeting.o* because it finds two definitions (or implementations) of the function **`greeting`** - one in *main.o* and the second in *greeting.o*. And, further, the linker is not programmed to choose one of the two definitions and discard the other. Instead, the linker is programmed to stop the linking process and flag an error about the presence of multiple definitions.

Let's review our terminology. A ***definition*** is a special kind of declaration that creates an object (function or variable); a ***declaration*** indicates a name that allow you to refer to an object created here or elsewhere.

definition	occurs in only one place	specifies the type of an object; reserves storage for it;
		is used to create new objects
		
		example:
		`int foo(void) {`
		`  // implementation`
		`}`

declaration	can occur multiple times	describes the type of an object; is used to refer to objects defined elsewhere (e.g., in another file) so that the compiler can correctly translate the use of this object example: `int foo(void);`

Based on our discussions so far, would you agree that although there can be multiple declarations of a function in a program's source files, a function can have only a single definition in these source files?

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **19**

Correct

1.00 points out of 1.00

A *header file* is a file containing related C declarations and macro definitions (which will be discussed later) to be shared between several source files. Why do programmers use header files?

1. The first reason is that errors generated by repeated typing or "copying and pasting" of declarations can be removed. Consider a source file *stats.c* defining functions that perform statistical analysis on data.
   Let **`find_median`**, **`find_mean`**, **`find_max`**, **`find_min`**, **`sort_array`**, and **`print_stats`** be some of the functions defined in *stats.c*. Many other source files in the program may wish to call some or all of these functions. Just as with *main.c* (where we had to declare function **`greeting`**), functions must be declared before their first use. One method is to copy and paste the appropriate declaration. However, copying and pasting declarations in many different source files is tedious, time-consuming and error-prone. Instead, it would be better to group these related declarations in a header file called *stats.h* (download **here**) that is somehow included by source files (more on this later) requiring these declarations.
2. Collecting related declarations in a single header file *stats.h* localizes any changes to declarations to that file. When the header file gets updated, source files that include *stats.h* will automatically use the new version when next recompiled. Otherwise, programmers will have to identify, locate and update every source file declaring these functions grouped in header file *stats.h*.
3. Robust and well-designed libraries can be implemented by separating ***implementation details*** (in a source file) from the ***interface*** (the grouping of related declarations in a header file). Header file *stats.h* provides the *interface* (*what* can the statistical package do?) while object file *stats.o* (download **here**) contains the *implementation details* (*how* does the statistical package do what it can do?). Other programmers can use the statistical package without concerning themselves with the implementation details by linking with *stats.o* (notice that users have access to only the object file not the source file *stats.c*).

Suppose you are a client programmer authoring source file *stats-driver.c* (download **here**) wishing to use the summary statistics functions defined in *stats.o* and declared in *stats.h*. In *stats-driver.c*, comment the line with preprocessor **`include`** directive that includes header file *stats.h*. Compile (only) source file *stats-driver.c* like this:

```
gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c stats-driver.c -o stats-driver.o
```

You'll notice that the compiler refuses to compile *stats-driver.c* because summary statistics function **`print-statistics`** is not declared. Look for the declaration of function **`print-statistics`** in *stats.h* and copy the declaration in *stats-driver.c* so that *stats-driver.c* can be compiled. If you want to call the other summary statistics functions from function **`main`** in *stats-driver.c*, each of these functions must be declared. Instead, of declaring these functions individually in *stats-driver.c*, you could just include the header file *stats.h*.

Would you agree that grouping related declarations in a header file is a major convenience to programmers because clutter in source files is reduced, repeated typing of the same declarations in multiple source files is eliminated, changes to declarations are localized to the header file, and robust and well-designed libraries can be implemented?

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **20**

Correct

1.00 points out of 1.00

Since header files are a major convenience, how can programmers have header files inserted into source files so that the compiler can see the declarations? A special program called the ***preprocessor*** was created to perform this and many other basic text editing tasks. The preprocessor is a text editing program that is invoked by the C compiler to edit the original source file and write out a new "preprocessed" source file that that can then be used as input to the C compiler. In addition to inserting files, the preprocessor can also be used to remove code and perform text substitution.

The preprocessor is controlled by special preprocessor directive (or command) lines, which are lines of the source file beginning with the character **#**. The preprocessor typically removes all preprocessor directive lines from the source file and makes additional transformations on the source file as specified by the directives, such as text substitution. Note that the syntax of preprocessor directives is completely independent of the syntax of the rest of the C language. A comprehensive reference of the C/C++ preprocessor can be found **here**.

For now, we'll look at the **`#include`** directive. Programmers request use of a header file in a source file by *including* it with the preprocessing directive **`#include`**. There are two forms of the **`#include`** directive. The form

```
#include <filename>
```

searches for the file *filename* in certain standard places according to implementation-defined search rules. We've used this form in the source *greeting.c* when we had to include the standard C library header file *stdio.h*.

The form

```
#include "filename"
```

will search for *filename* in the directory containing the source file that included the header file.

The general rule-of-thumb is that the **`#include "filename"`** form is used to refer to header files written by the programmer, while the **`#include <filename>`** form is used to refer to standard implementation files.

Now, create a header file *greetings.h* with the following text:

```
/*
greetings.h
declarations for all types of greetings
*/
void greetings(void);

```

Replace the declaration of function **`greetings`** in *main.c* with the preprocessor **`#include`** directive for header file *greetings.h*:

```
#include "greetings.h"

int main(void) {
  greetings();
  return 0;
}

```

Suppose the code in source file *greeting.c* is:

```
/* greeting.c */
#include <stdio.h> // declared function printf

void greetings(void) {
  printf("Hello World!\n");
}

```

Are you able to compile (but not link) *main.c* after replacing the declaration of function **`greetings`** with the preprocessor **`#include`** directive for header file *greetings.h*? Are you able to link *main.o* and *greeting.o* to create an executable?

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **21**

Correct

1.00 points out of 1.00

Source file *stats-driver.c* (download **here**) calls the statistics functions declared in *stats.h* (download **here**) and implemented in object file *stats.o* (download **here**). Note that this object file is only compatible with Linux and *GCC* platform in lab computers and the installation process described in the software installation handout that you used to install WSL/Linux. Compile (only) source file *stats-driver.c* and link the resultant object file with *stats.o* to create an executable program *stats-test.exe*. Execute the program *stats-test.exe* and write the comma separated mean and median values printed to standard output. If your personal computer doesn't yet have an installation of WSL/Linux, complete this question in one of the lab computers.

Answer: 498,489 ✔

The correct answer is: 498,489

Question **22**

Correct

1.00 points out of 1.00

Which option must be used to inform `gcc` to stop after the preprocessing stage without running the compiler proper? Use the GNU C Compiler options **page** to find out the correct answer.

Select one:

- `-x`
- `-c`
- `-S`
- `-E` ✔
- `-o`

Your answer is correct.

The correct answer is: `-E`

Question **23**

Correct

1.00 points out of 1.00

The output generated by the preprocessor stage can be saved to file with the `-o` option:

```
gcc -E main.c -o main.i
```

By viewing file *main.i* generated by the preprocessor stage, can you confirm that comments are replaced by white-space (the space, horizontal tab, new-line) characters and the line `#include "greeting.h"` is replaced by the contents of header file *greeting.h*?

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **24**

Correct

1.00 points out of 1.00

`gcc` implicitly assumes that files with suffix *.c* are C source files and will automatically invoke the C compiler. Assume that we've invoked the preprocessor on a C11 source file *main.c* and redirected its output to a text file *main.i*. How can we now use `gcc` to compile (but not link) using the file *main.i* generated by the preprocessor? Use the GNU C Compiler options **page** to determine the correct answer.

Note that the choices below present `gcc` options of interest to this question only and to avoid the confusion that would result from presenting the entire list of `gcc` options.

Select one:

- `gcc -std=c11 -x c main.i`
- `gcc -std=c11 -E -c main.i`
- `gcc -std=c11 -c -x c main.i -o main.o` ✔
- `gcc -std=c11 -o -x c main.i`

Your answer is correct.

The correct answer is: `gcc -std=c11 -c -x c main.i -o main.o`

Question **25**

Correct

1.00 points out of 1.00

When all source files are compiled, the object files are given to a program called ***linker***. The linker resolves references between the object files, adds functions from the standard run-time library (such as function **`printf`**), and detects some programming errors such as the failure to define a needed function. The output of the linker program is a single executable program, which can then be invoked or run.

Consider the interface (or header) file *greeting.h*

```
/*
greetings.h
declarations for all types of greetings
*/
void greetings(void);

```

the implementation (or source) file *greeting.c*

```
/* greeting.c */
#include <stdio.h> // declares function printf

void greeting(void) {
  printf("Hello World!\n");
}

```

and a driver source file *main.c*

```
#include "greetings.h"

int main(void) {
  greetings();
  return 0;
}

```

First, compile source files *main.c* and *greeting.c* separately:

```
gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c main.c -o main.o
gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -c greeting.c -o greeting.o
```

The separate compilations of *main.c* and *greeting.c* to generate corresponding object files *main.o* and *greeting.o* respectively, should again reinforce the idea that the process of compiling takes as input a single source file.

Now, that we've individually compiled source files *main.c* and *greeting.c* to object files *main.o* and *greeting.o* respectively, let's understand the idea that linking is a ***collective process*** requiring the linker to process all necessary object files together to create an executable program.

Create the executable program *test.out* by calling the linker and supply it only the object file *main.o*:

```
gcc main.o -o test.out
```

Were you able to successfully create the executable program *test.out*?

Select one:

- True
- False ✔

The correct answer is 'False'.

Question **26**

Correct

1.00 points out of 1.00

Let's review the different compilation stages one more time. The exact sequence of compilation stages processed by a single invocation of `gcc` is: ***preprocessing***, ***compilation***, ***assembling***, and ***linking***.

Let's use the statistics program (*stats.h* (download **here**), *stats.o* (download **here**), *stats-driver.* (download **here**)) that was previously introduced to examine the output from each compilation stage beginning with the preprocessing stage. Note that *stats.o* is only compatible with the WSL/Linux and `gcc` installation on your personal computers (using the instructions in the course handout.

Which of the following commands writes the output of the preprocessing stage into file *stats-driver.i*? Examine the output file *stats-driver.i* and confirm that the `#include` (and the `#define`) directives were correctly executed by the preprocessor. Assume as always that you're using C11 standard. Use **this** page to determine the correct answer.

Select one:

- `gcc -std=c11 -pedantic-errors -Wstrict-prototypes -x stats-driver.c -Wall -Wextra -Werror -o stats-driver.i`
- `gcc -std=c11 -pedantic-errors -Wstrict-prototypes -S stats-driver.c -Wall -Wextra -Werror -o stats-driver.i`
- `gcc -std=c11 -pedantic-errors -Wstrict-prototypes -c stats-driver.c -Wall -Wextra -Werror -o stats-driver.i`
- `gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -E stats-driver.c -o stats-driver.i` ✔

Your answer is correct.

The correct answer is: `gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -E stats-driver.c -o stats-driver.i`

Question **27**

Correct

1.00 points out of 1.00

The preprocessing stage's output *stats-driver.i* is then passed to the compiling stage. Since writing a compiler is a non-trivial task, the compilation stage is divided into several phases with well-defined interfaces. Conceptually, these phases operate in sequence, each phase (except the first) taking the output from the previous phase as its input. A common division into phases is described below.

- **Lexical analysis:** This is the initial part of the reading and analyzing the program text in *stats-driver.i*. The text is read and divided into ***tokens*** (read about tokens on pages 27-29 of the text), each of which corresponds to a symbol in the C programming language, e.g., a variable name, keyword, a function name, or number.
- **Syntax analysis:** This phase - also called *parsing* - takes the list of tokens produced by the lexical analysis and arranges them in a tree-like structure called the *syntax tree* that reflects the structure of the program.
- **Type checking:** This phase analyzes the syntax tree to determine if the program violates certain consistency requirements, e.g., if a variable is used but not declared or if it is used in a context that doesn't make sense given the type of the variable, such as using a function that returns nothing as an operand of the addition operator.
- **Assembly code generation:** The program is translated to assembly language for a specific machine architecture.

Using appropriate options to `gcc`, compile the file *stats-driver.i* that was generated by the preprocessing stage but ensure that compilation stops after generating the assembly language version of the C11 source file.

Select one:

- `gcc -Wall -Wextra -Werror -c -x c stats-driver.i -o stats-driver.s`
- `gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -x c -S stats-driver.i -o stats-driver.s` ✔
- `gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -x c -c stats-driver.i -o stats-driver.s`
- `gcc -Wall -Wextra -Werror -S -x c stats-driver.i -o stats-driver.s`

Your answer is correct.

The correct answer is: `gcc -std=c11 -pedantic-errors -Wstrict-prototypes -Wall -Wextra -Werror -x c -S stats-driver.i -o stats-driver.s`

Question **28**

Correct

1.00 points out of 1.00

The next step is to translate the assembly code in *stats-driver.s* into machine language and generate the corresponding object file *stats-driver.o*.

Using appropriate options to **`gcc`**, translate the assembly file *stats-driver.s* that was generated by the compiling stage into the corresponding object file *stats-driver.o*.

Select one:

- `gcc stats-driver.s -o stats-driver.o`
- `gcc -S stats-driver.s -o stats-driver.o`
- `gcc -c assembler stats-driver.s -o stats-driver.o`
- `gcc -c -x assembler stats-driver.s -o stats-driver.o` ✔

Your answer is correct.

The correct answer is: `gcc -c -x assembler stats-driver.s -o stats-driver.o`

Question **29**

Correct

1.00 points out of 1.00

Link object files *stats-driver.o* (generated by the assembler) and *stats.o* (the implementation of the statistics library) to generate an executable *stats-test.out*. Run the executable and report the comma-separated maximum, minimum, mean, and median values (just the values), in that order.

Answer: 994,3,498,489 ✔

The correct answer is: 994,3,498,489

Question **30**

Correct

1.00 points out of 1.00

All C programs must define a single function named **`main`**. This function will be the entry point of the program - that is, the first function executed when the program is started. Returning from this function terminates the program, and the returned value is treated as an indication of program success or failure.

Edit source file *main.c* (from the *greeting* program that was discussed earlier) and amend the name of function **`main`** to **`foo`**. Next, compile *main.c* and supply object files *main.o* and *greeting.o* to the linker. Were you able to create an executable program?

Select one:

- True
- False ✔

The correct answer is 'False'.

Question **31**

Correct

1.00 points out of 1.00

Based on our discussion so far, would you agree that every C program must have ***one and only one*** function called **`main`**.

Select one:

- True ✔
- False

The correct answer is 'True'.

◄ Quiz 1: Introduction to Programming

Jump to...

Lecture 5 [09/12/2022]: Introduction to C Programming Part 3/3 ►