Maciej Sobieraj

Lecture 10

Outline

1. Functions
   1. **Parameterizing the main function**
   2. The basics of disjoint compilation
   3. Examples
2. Quiz

Parameterizing the main function

- So far, the *main* function has appeared in our programs only as a **parameterless** one; that is, declared with the following prototype.

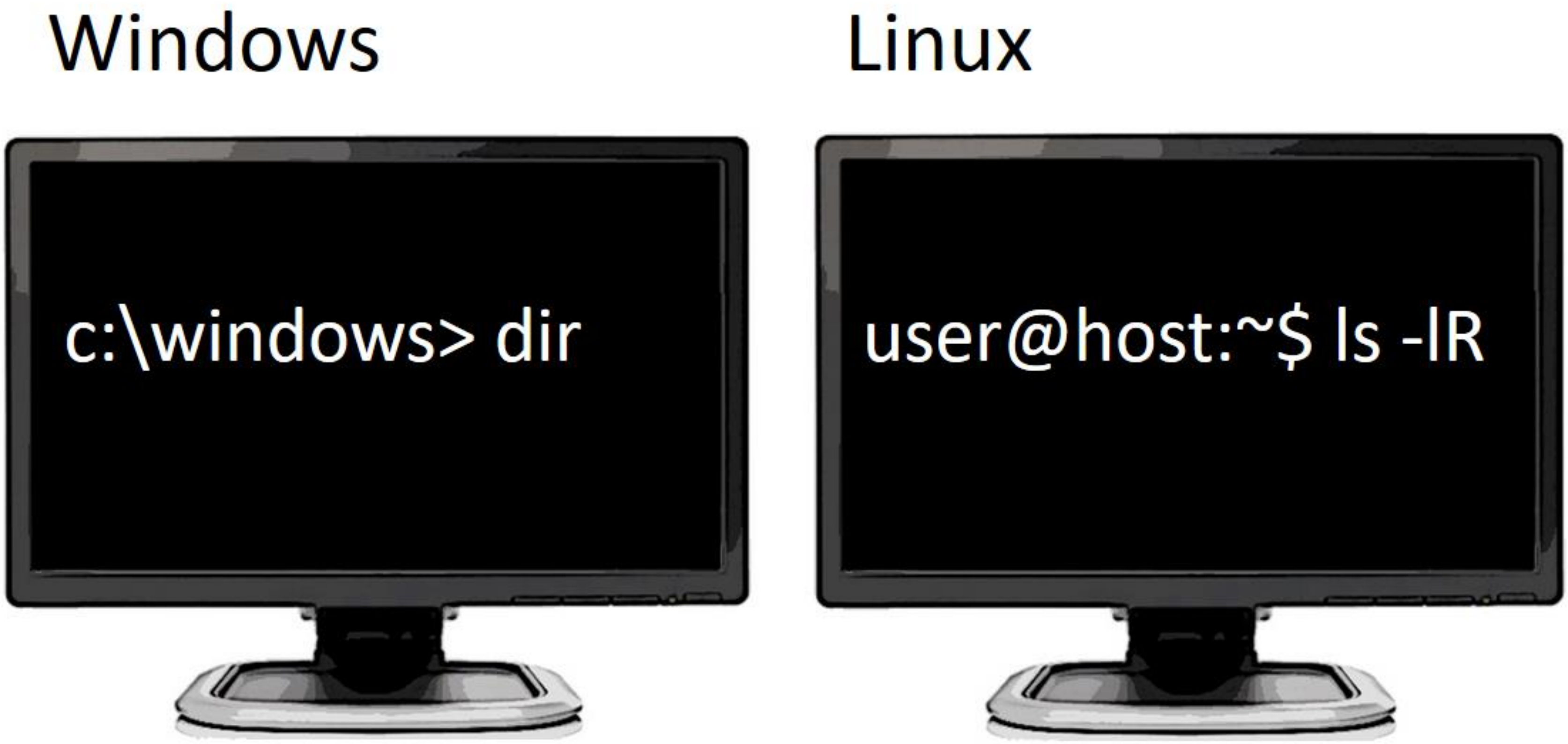
```
int main(void);
```

- This way of declaring the *main* function isn't used often, because we rarely write programs which don't benefit from the possibility of **retrieving** some information from a user at the time of startup.

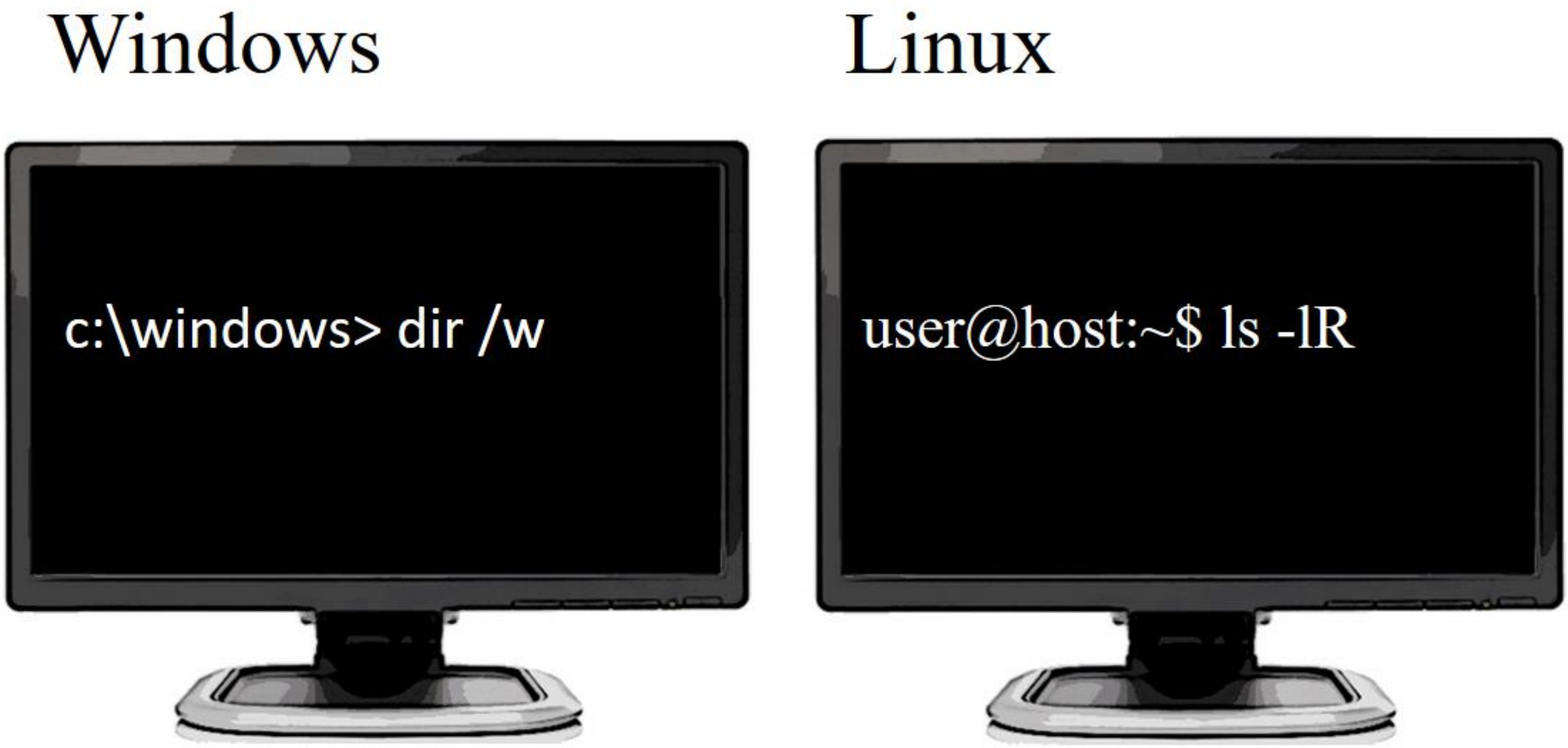
Parameterizing the main function

- If you've ever used, for example, the *dir* command in the Windows console environment or the *ls* command in the Unix console environment, you may have noticed that these commands can be issued either with or without arguments.

Parameterizing the main function



Parameterizing the main function



Parameterizing the main function

- How can it be that the arguments supplied in the **command line** can be retrieved and interpreted by the program? The mechanism for transferring command-line arguments to the running program is integrated in the main function, and to take effect, the *main* function must be **declared in a special way**

```
int main(int argc, char *argv[])
```

Parameterizing the main function

- Now, let's decipher the hidden meaning of the names and their purposes:
  - *argc* (**arg**ument **c**ounter): contains the **number** of arguments passed on to the program plus one; this means that a program run without any arguments will have an *argc* parameter value equal to 1
  - *argv* (**arg**ument **v**alues): an **array of pointers** to strings containing the arguments supplied to the program; they're stored in the following way:
    - *argv[0]* contains the name of the running program
    - *argv[1]* contains the string passed to the program as the first argument
    - *argv[n]* contains the string passed to the program as the *n-th* argument

```
int main(int argc, char *argv[])
```

Parameterizing the main function

- Let's write a simple code whose task will be to demonstrate the mechanism of passing the arguments on to the main function. It'll print all of its arguments in one column.

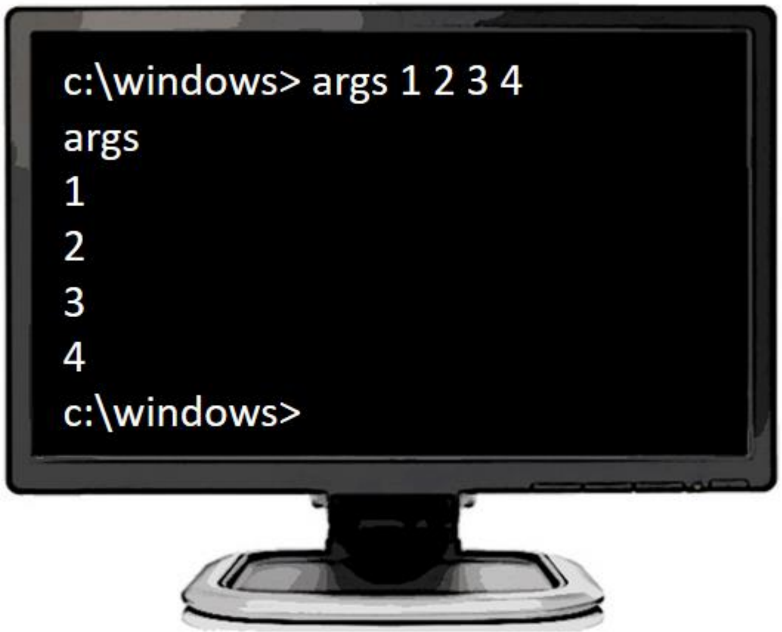
```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;

    for(i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

Parameterizing the main function

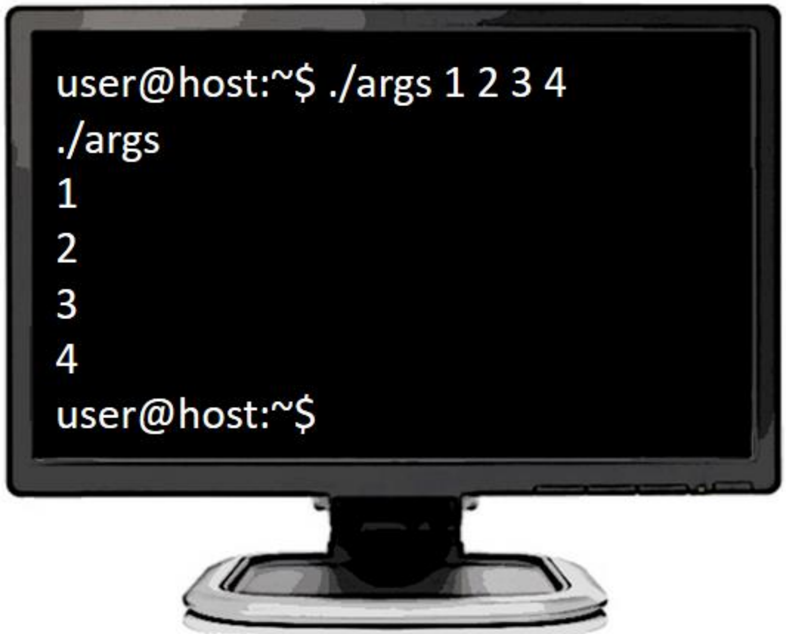
- If you put this program in a file named *args.c* and then compile it, you’ll get an executable file named (probably) *args.exe* (in Windows) or *args* (in a Unix environment).

Parameterizing the main function

Windows

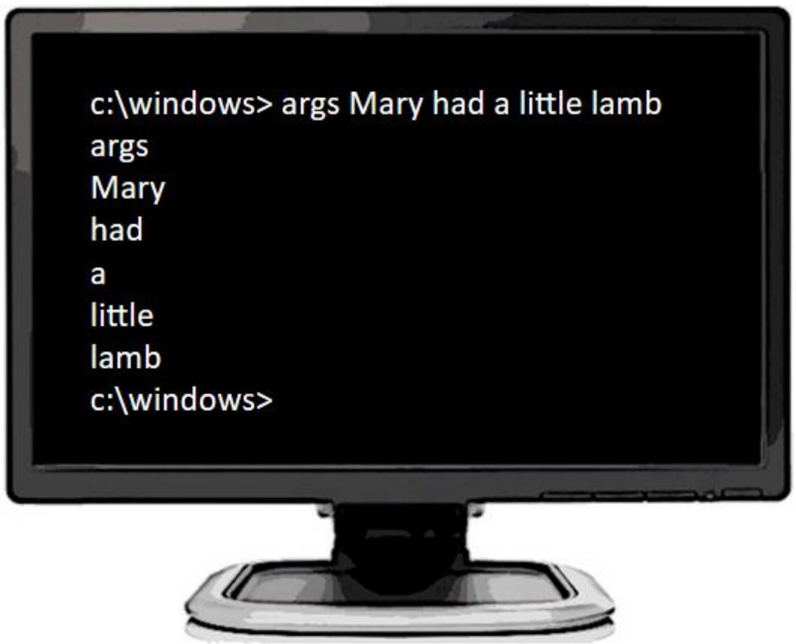


Linux



Parameterizing the main function

Windows



Linux

Parameterizing the main function

Windows

Linux

Parameterizing the main function

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    if(argc == 1) {
        printf("usage: args arg1 arg2 arg3 ...\n");
        return 1;
     }
    if(argc == 2 && strcmp(argv[1],”-v”) == 0) {
        printf("args version 1.0, C language course, 2012\n");
        return 2;
    }
    for(i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

Outline

1. Functions
   1. Parameterizing the main function
   2. **The basics of disjoint compilation**
   3. Examples
2. Quiz

Disjoint compilation: why?

- Let's imagine that you, as good a programmer as you are, decide to write an incredibly complex program, together with your friend. This program is intended to get an *int* number from the user and calculate the **factorial** of that number.

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n$$

Factorials and how to count them

```c
int factorial(int n) {
    int i, product = 1;

    for(i = 1; i <= n; i++)
        product *= i;
    return product;
}
```

Factorials and how to count them

- Let's try to rewrite the code in a somewhat different way.
- The factorial is calculated “backwards”

```
int factorial(int n) {
    int product = 1;

    while(n)
        product *= n--;
    return product;
}
```

Factorials and how to count them

- If your friend is a fan of mysterious coding, he could hide his intentions more effectively by writing something like this

```
int factorial(int n) {
    int product;

    for(product = 1; n; product *= n--) ;
    return product;
}
```

Factorials and how to count them

- Now’s a good opportunity to talk about a programming technique called **recursion**.

5! = 1 * 2 * 3 * 4 * 5

5! = 1! * 2 * 3 * 4 * 5

5! = 2! * 3 * 4 * 5

5! = 3! * 4 * 5

5! = 4! * 5

Factorials and how to count them

- Looking at the this reasoning, we see that:
  - n! = 1 when n == 1
    n! = (n – 1)! * n when n > 1

```
int factorial(int n) {
    if(n == 1)
        return 1;
    else
        return n * factorial(n – 1);
}
```

Factorials and how to count them

- Can you see it? The **factorial function invokes itself**. This is exactly what we call a **recursion**.

```
int factorial(int n) {
    if(n == 1)
        return 1;
    else
        return n * factorial(n – 1);
}
```

A new operator: a three-argument one

- This operator is highly original, because it requires **three arguments**.

expression1 ? expression2 : expression3

A new operator: a three-argument one

- This operator works as follows:
  - it **calculates** the value of *expression1*
  - if the value calculated is a non-zero, the operator **returns** the value of ***expression2***, neglecting completely *expression3*
  - if the value calculated in step 1 is zero, the operator **returns** the value of ***expression3***, omitting *expression2*.

A new operator: a three-argument one

- This means that the result of the following expression:
  - i = i > 0 ? 1 : 0;
- will be calculated in the following way:
  - variable *i* will be assigned a *1* if its previous value was greater than zero, and a *0* otherwise.

A new operator: a three-argument one

  - i = i > 0 ? 1 : 0;
- Note that we can achieve the same effect using a conditional statement:
  - ```
    if(i > 0)
        i = 1;
    else
        i = 0
    ```
- This is somewhat more extensive, although it's undeniably more readable at the same time.
    ```

Back to the disjoint compilation

- We can include it in the file *factorial.c.*

```
int factorial(int n) {
    return n ? n * factorial (n - 1)  : 1;
}
```

Back to the disjoint compilation

- Each author of a piece code, who is planning to share their work results with other programmers, usually prepares at least two source files:
  - the **first file contains the source code** (in our case it's *factorial.c*)
  - the **second file contains the declarations** (not definitions!) of all the entities (symbols, types, variables, functions) intended to be shared with others; this file is called a **header file** and its name should end with the *suffix .h* (in our case it would be *factorial.h*)

Back to the disjoint compilation

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   int n, result;

   printf("Enter n:");
   scanf("%d", &n);
   if(n <= 0 || n > 20) {
      printf("Bad news: you've entered an invalid value.");
      return 1;
   }

   /* we will invoke our friend's function here */

   printf("Factorial of %d is %d\n", n, result);
   return 0;
}
```

Back to the disjoint compilation

- Your friend should run the editor once more and create a file named *factorial.h*.

```
extern int factorial(int n);
```

- Here’s a new word: ***extern*** (*external*). This is a keyword, often called an **attribute**, which can be used along with the declarations of functions and variables. Its presence indicates that the function/variable described in this declaration is defined in a different source file.

Back to the disjoint compilation

- Of course, you can add the prototype directly into the source file *program.c*, but the header files eliminate the need to do so.
- You can also omit the parameter names in the prototype, as you're obliged only to specify their types. This means that we can simplify the declaration to the following form:
  - **extern int** factorial(**int**);

Back to the disjoint compilation

- Now your friend sends you both files: the source (.c) and the header (.h). Then you have to make the following two amendments:
  - add a *#include* directive to inform the preprocessor that it should analyze the new header files
  - encode the correct invocation to the *factorial()* function.
- As you can see, the purpose of a header file can be twofold:
  - firstly, the compiler finds out how to compile the external function invocation;
  - secondly, the programmer can learn how to use the functions in the program.

Back to the disjoint compilation

- The header is often supplemented with a few comments documenting the purpose of the file. Your friend has already done it, so the final version of the *factorial.h* file looks as follows:

```
/***********************************************************
factorial version 1.0, 24 July 2012
author: sauron@mordor.com
the function computes the factorial of its argument
acceptable range of the parameter is [1..20]
the correctness of the parameter isn't checked – be careful!
************************************************************/

extern int factorial(int);
```

Back to the disjoint compilation

- Here's the *program.c* file

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

int main(int argc, char *argv[]) {
   int n, result;

   printf("Enter n value:");
   scanf("%d", &n);
   if(n <= 0 || n > 20) {
      printf("Bad news: you've entered an invalid value.\n");
      return 1;
   }

   result = factorial(n);

   printf("Factorial of %d is %d\n", n, result);
   return 0;
}
```

Back to the disjoint compilation

- You’re accustomed to using the *#include* directive in this form:
  - #include <file.h>
- but something new has appeared:
  - #include "file.h"
- It's not a mistake.

Back to the disjoint compilation

- Here's the importance of this duality:
  - if the file name is surrounded by **angle brackets** < >, it means that the preprocessor should look for the included files in the **standard locations**; in Unix environments, these locations are usually placed inside the */usr/include* directory (the name speaks for itself);
  - if the file name is surrounded by **quotes** " ", it means that the preprocessor should look for the included file in **the same directory** where the original file processed by the preprocessor was located.

Outline

1. Functions
    1. Parameterizing the main function
    2. The basics of disjoint compilation
    3. **Examples**
2. Quiz

Example 1

- Write a program that prints two triangles: one is a normal triangle consisting of backslashes and the other is a Floyd's triangle.
- Remember to escape the backslash with a backslash (not a play on words!).
- A Floyd's triangle consisting of numbers in consecutive order: in the first row, we have only one number: 1; in the second row, two numbers: 2 3; in the third row: 4 5 6 and so on

Example 1

Example input

```
5
```

Example output

```
5

\
\\
\\\
\\\\

    1
    2    3
    4    5    6
    7    8    9   10
```

Example 1

Example input

```
15
```

Example output

```
\
\\
\\\
\\\\
\\\\\
\\\\\\
\\\\\\\
\\\\\\\\
\\\\\\\\\
\\\\\\\\\\
\\\\\\\\\\\
\\\\\\\\\\\\
\\\\\\\\\\\\\
\\\\\\\\\\\\\\

  1
  2   3
  4   5   6
  7   8   9  10
 11  12  13  14  15
 16  17  18  19  20  21
 22  23  24  25  26  27  28
 29  30  31  32  33  34  35  36
 37  38  39  40  41  42  43  44  45
 46  47  48  49  50  51  52  53  54  55
 56  57  58  59  60  61  62  63  64  65  66
 67  68  69  70  71  72  73  74  75  76  77  78
 79  80  81  82  83  84  85  86  87  88  89  90  91
 92  93  94  95  96  97  98  99 100 101 102 103 104 105
```

Example 1

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

void floydsTriangle(int size);
void normalTriangle(int size);

int main()
{
 int size;
 scanf("%d", &size);
 normalTriangle(size);
 floydsTriangle(size);
 return 0;
}
```

Example 1

```
/* triangles.h */
void floydsTriangle(int size);
void normalTriangle(int size);
```

Example 1

```
/* floydsTriangles.c */
#include <stdio.h>
void floydsTriangle(int size)
{
 int i, j, k=1;
 for (i = 0; i<size; i++)
 {
  for (j = 0; j<i; j++)
  {
   printf("%4d", k);
   k++;
  }
  printf("\n");
 }
}
```

Example 1

```
/* normalTriangles.c */
#include <stdio.h>
void normalTriangle(int size)
{
 int i, j;
 for(i = 0 ; i<size ; i++)
 {
  for(j=0; j<i ; j++)
  {
   printf("\\");
  }
  printf("\n");
 }
}
```

Example 2

- Write a program that allows the user to pass the parameters to be executed and compute the results of some mathematical operations.
- Your program should support the following operations:
  - add
  - sub
  - mul

Example 2

- All operations require an additional two arguments. Some examples of program calls inlude:
  - program.exe add 1 3
  - program.exe sub 2 3
  - program.exe mul 2 5
- When there are no parameters, the parameters contain the wrong numbers or a parameter is invalid, the program should print the message "Wrong parameters"

Example 2

Example input

```
add 1 3
```

Example output

```
add 1 3: 4
```

Example input

```
sub 2 3
```

Example output

```
sub 2 3: -1
```

Example 2

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
 if (argc != 4)
 {
  puts("Wrong parameters");
  return -1;
 }
 int number1 = atoi(argv[2]);
 int number2 = atoi(argv[3]);
 int result;
 if (!strcmp(argv[1], "add"))
  result = number1 + number2;
 else if (!strcmp(argv[1], "sub"))
  result = number1 - number2;
 else if (!strcmp(argv[1], "mul"))
  result = number1 * number2;
 else
 {
  puts("Wrong parameters");
  return -1;
 }
 printf("%s %d %d: %d\n", argv[1], number1, number2, result);
 return 0;
}
```

Example 3

- Write a function that computes the square of a given floating-point number and returns its value.
- Separate the declaration of the function from its full definition.

Example output

```
square of 2.00 is 4.00
square of 6.00 is 36.00
square of 2.50 is 6.25
square of 12.12 is 146.89
square of 345.68 is 119493.29
```

Example 3

```
#include <stdio.h>

float square(float);

int main(void)
{
 printf("square of %.2f is %.2f\n", 2.0, square(2.0));
 printf("square of %.2f is %.2f\n", 6.0, square(6.0));
 printf("square of %.2f is %.2f\n", 2.5, square(2.5));
 printf("square of %.2f is %.2f\n", 12.12, square(12.12));
 printf("square of %.2f is %.2f\n", 345.678, square(345.678));
 return 0;
}

float square(float value)
{
 return value*value;
}
```

Example 4

- Write a function that checks whether or not a given string is a valid IP address (in human-readable form, of course).
- This function should return 1 if the address is valid, and 0 if not. Your function should check if: there are 4 parts in the string, separated by dots; each part contains only digits, each number is in the range of 0 to 255, inclusive.
- For converting string fragments to integer va
you can use the strtol, atoi or sscanf functio

Example 4

Example output

```
127.0.0.1 is a valid IP address
127.0.01 is not a valid IP address
127.0..1 is not a valid IP address
127.zero.0.1 is not a valid IP address
127.297.0.1 is not a valid IP address
127.2555.0.1 is not a valid IP address
```

Example 4

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int checkIP(char *);
void printIPValid(char *);

int main(void)
{
 printIPValid("127.0.0.1");
 printIPValid("127.0.01");
 printIPValid("127.0..1");
 printIPValid("127.zero.0.1");
 printIPValid("127.297.0.1");
 printIPValid("127.2555.0.1");
 return 0;
}
```

Example 4

```
int checkIP(char *address)
{
 int result = 1;
 int i;
 int actualCount=0;
 int partsCount=1;
 char actualPart[5];

 for (i = 0; i<strlen(address); i++)
 {
  char c = address[i];
  if (c >= '0' && c <= '9' || c == '.')
  {
   if (c == '.')
   {
    if (actualCount<1) /* for cases like: "127.0..*/
    {
     result = 0;
     break;
    }
    partsCount++;
    if (partsCount>4)
    {
     result = 0;
     break;
    }
    actualPart[actualCount] = '\0';
    int partValue = atoi(actualPart);
    if (partValue>255 || partValue<0)/*second condition is not needed*/
    {
     result = 0;
     break;
```

Example 4

```
int checkIP(char *address)
{
 int result = 1;
 int i;
 int actualCount=0;
 int partsCount=1;
 char actualPart[5];

 for (i = 0; i<strlen(address); i++)
 {
  char c = address[i];
  if (c >= '0' && c <= '9' || c == '.')
  {
   if (c == '.')
   {
    if (actualCount<1) /* for cases like: "127.0..*/
    {
     result = 0;
     break;
    }
    partsCount++;
    if (partsCount>4)
    {
     result = 0;
     break;
    }
    actualPart[actualCount] = '\0';
    int partValue = atoi(actualPart);
    if (partValue>255 || partValue<0)/*second condition is not needed*/
    {
     result = 0;
     break;
```

Example 4

```
    }
    actualCount = 0;
   }
   else
   {
    if (actualCount < 3)
    {
     actualPart[actualCount] = c;
     actualCount++;
    }
    else
    {
     result = 0;
     break;
    }
   }
  }
  else
  {
   result = 0;
   break;
  }
 }
 if (partsCount != 4)
  result = 0;
 return result;
}
```

Example 4

```
void printIPValid(char * address)
{
 if (checkIP(address))
  printf("%s is a valid IP address\n", address);
 else
  printf("%s is not a valid IP address\n", address);
}
```

Example 5

- Write a function that checks which of two given matrices is greater. To simplify the function parameter list, you can assume that both matrices are equal in size and are square. This function should return:
  - 1 if the first matrix is greater than the second;
  - -1 if the first matrix is smaller than the second;
  - 0 if both matrices are equal - this means they have exactly the same values.

Example 5

- For this task, we assume that a matrix is smaller than another matrix when the first element which is different is smaller in that matrix.
- We analyze matrices from left to right and from top to bottom.

Example output

```
Both matrices are equal.
matrixA is smaller than matrixB
matrixA is greater than matrixB
Both matrices are equal.
matrixA is smaller than matrixB
matrixA is greater than matrixB
```

Example 5

```c
#include <stdio.h>

int matrixCompare(int, int *, int *);
void printMatrixCompare(int, int *, int *);

int main(void)
{
 int matrixA[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
 int matrixB[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
 int matrixC[3][3] = { 1, 4, 3, 4, 5, 6, 7, 8, 9 };

 printMatrixCompare(3, *matrixA, *matrixB);
 printMatrixCompare(3, *matrixA, *matrixC);
 printMatrixCompare(3, *matrixB, *matrixA);
 printMatrixCompare(3, *matrixB, *matrixC);
 printMatrixCompare(3, *matrixC, *matrixA);
 printMatrixCompare(3, *matrixC, *matrixB);
 return 0;
}
```

Example 5

```
int matrixCompare(int pSize, int *matrixA, int *matrixB)
{
 int i, j;
 for (i = 0; i < pSize; i++)
 {
  for (j = 0; i < pSize; i++)
  {
   if (*matrixA!=*matrixB)
   {
    if (*matrixA < *matrixB)
     return -1;
    else
     return 1;
   }
   matrixA++;
   matrixB++;
  }
 }
 return 0;
}
```

Example 5

```
void printMatrixCompare(int pSize, int *matrixA, int *matrixB)
{
 int result = matrixCompare(pSize, matrixA, matrixB);
 if (result == 0)
  puts("Both matrices are equal.");
 if (result == -1)
  puts("matrixA is smaller than matrixB");
 if (result == -1)
  puts("matrixA is greater than matrixB");
}
```

Outline

1. Functions
    1. Parameterizing the main function
    2. The basics of disjoint compilation
    3. Examples
2. **Quiz**

Quiz

What happens if you try to compile and run this program with the following command?

```
prog MARY HAD A LITTLE LAMB

#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
   printf("%d", argc + strlen(argv[1]));
   return 0;
}
```

- the program outputs 6
- the program outputs 10
- the program outputs 8

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int fun(int n) {
   if(n == 0)
      return 0;
   return n + fun(n - 1);
}
int main(void) {
   printf("%d",fun(3));
   return 0;
}
```

- the program outputs 1
- the program outputs 3
- the program outputs 6