Functions

Up to this point:

- **We have learned about "tools" available for us to use in programming:**
  - We learned about variables to hold information to use in our program
    - We can input into variables from users: *scanf("%d", &n);*
    - We can save function return values into variables *int n=strcmp(word1, word3);*
    - We can save the outcome of an operation into a variable *int n=2*3;*
    - We can hardcode values into variables *int n=3;*

  - We learned about data structures to hold multiple bits of information (specifically arrays)
    - We also learned about 2D arrays to hold arrays as elements
    - Note that even though we don't talk about higher dimension arrays in 1310, you could do 3D arrays, 4D arrays etc (using the same approach that we did to go from 1D arrays to 2D arrays)

  - We learned about operators to manipulate values (like arithmetic operators, assignment operators, relational operators and logical operators)
    - Remember we can store the results of the outcomes when using operators in variables:
      - int n=2*3;
      - int n=!(2+3==4);
      - n+=4;

  - We learned about if statements to make decisions
    - We can nest our if statements to make decisions within decisions

  - We learned about loops to repeat actions
    - We can nest our loops to repeat actions within repeating actions

  - We can also use if statements within loops and loops within if statements

- **Today, we will learn about making our own functions**
    - We have been using functions made by other people

    - We will now start making our own
      - All the tools we have learned so far (mentioned above) will still be used just as we have been using them
      - We are not going to stop thinking in the same way we have been doing since the beginning of the semester
        - Getting information and manipulating that information to arrive at some desired program outcome
      - What will change is we are going to start "compartmentalize" different conceptual tasks within in our program into functions
        - For example, instead of just writing a whole program in main that does one large task, we will break up the large task into smaller tasks (put into functions) and then put all those smaller tasks together to solve the larger task
        - Like putting puzzle pieces together

- Today, I will just introduce the concepts and give a few examples

- **WE WILL BE USING FUNCTIONS FOR THE WHOLE SEMESTER (and you should be using them for the rest of your programming life)**

    - The next step is to organize and modularize your problem-solving technique and approach
    - Everything that we have done so far was to get you comfortable with the idea of a programming language and using a program language to communicate your thoughts to the computer
        - I had a student ask once why didn't we just start with functions-I figure you need to get comfortable with the language first
        - As an analogy, I would assume most of you guys would want to achieve some level of fluency in English before you were asked to start writing a well-structured essay in the language…

Note: if you still struggling to understand concepts from the "tools" we have learned up to this point-don't worry, we will still be using them throughout the semester so you will still get plenty of practice. For example, if you are still struggling with loops, we will be using loops for the whole semester so you will still get practice with it.

Functions:

Note: I feel it is easier to understand functions when actually doing the sample code-this first part is just conceptual information

- So far, we have been using functions that have been created by other people
- We will now learn how to make our own functions
    - In programming, a function is a collection of programming statements (each themselves doing a task) that are combined to perform a specific task
        - Sometimes you need to give an input (or inputs) to your function in order for it to work (called ***parameters***)
        - Your function can have an output which we signify with a ***return type*** (void means nothing is returned)
- Other languages, like Java, call this concept ***methods***
- Before we start, we will give a visual of what a function really is
- Today, I will focus more on the concept with a few examples

Function analogy:

Input into the function (ingredients)

Inputs (arguments)

2. We name these wrapped up actions and refer to all these actions as the single overall name

Name: Make cake

1. Several individual actions are wrapped up together.

1. Mix ingredients
2. Put in oven
3. Take out of oven
4. Put frosting

Output from function (cake result of actions on the inputs)

Output (return value)

Function:

2. We name these wrapped up actions and refer to all these actions as the single overall name

Input (a vector):
1 2 1

1. Several individual actions are wrapped up together.

```
int wallet_total (std::vector <int> money)
{
  if(money.size()==0)
  {
   std::cout<<"Nothing in your wallet yet."<<std::endl;
   return 0;
  }
  int total=0;
  for(int i=0;i<money.size();i++)
  {
   total+=money[i];
  }
  return total;
}
```

Note: this is in C++, not C. The concept is the same

4 Output (an integer-the return type of this function is an int)

- Remember, we can save the output (if want) in a variable to use later
  - We can also just use it directly (as we have done before)

There are 2 ways to write functions in your code (you will see me doing both in this class):

1

```
#include <stdio.h>

void exp_function(int base, int power)
{
	int i=0;
	int answer=1;

	for(i=0;i<power;i++)
	{
		answer=answer*base;
	}

	printf("answer: %d\n", answer);
}

int main (int argc, char **argv)
{

	exp_function(3, 4);
	exp_function(5, 3);

}
```

2

```
#include <stdio.h>

void exp_function(int base, int power);

int main (int argc, char **argv)
{
	exp_function(3, 4);
	exp_function(5, 3);
}

void exp_function(int base, int power)
{
	int i=0;
	int answer=1;

	for(i=0;i<power;i++)
	{
		answer=answer*base;
	}

	printf("answer: %d\n", answer);
}
```


- Note that we can either:
  - Have the definition defined before use it OR
  - Have the declaration before we used it (with the definition given later)
- Note that different languages do not necessary follow this same expectation

Motivations for functions:

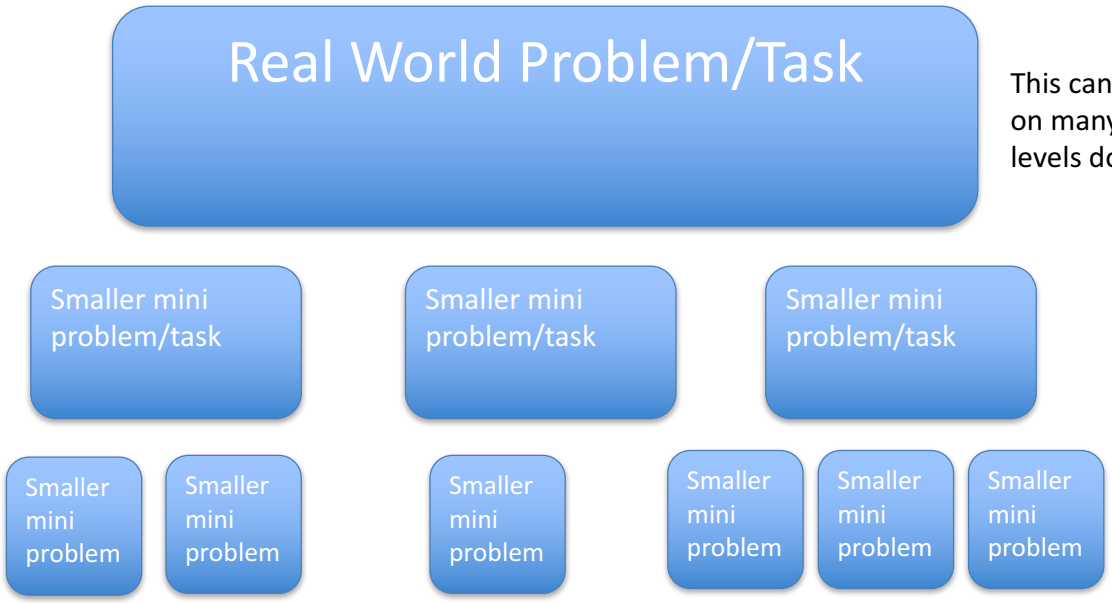
- Why do we even have functions?
  - Why do we need them?
    - Reuse code
    - Modularize code
    - Make problems easier to deal with

Motivation for Functions

- Before we start, lets look at a general problem solving framework:

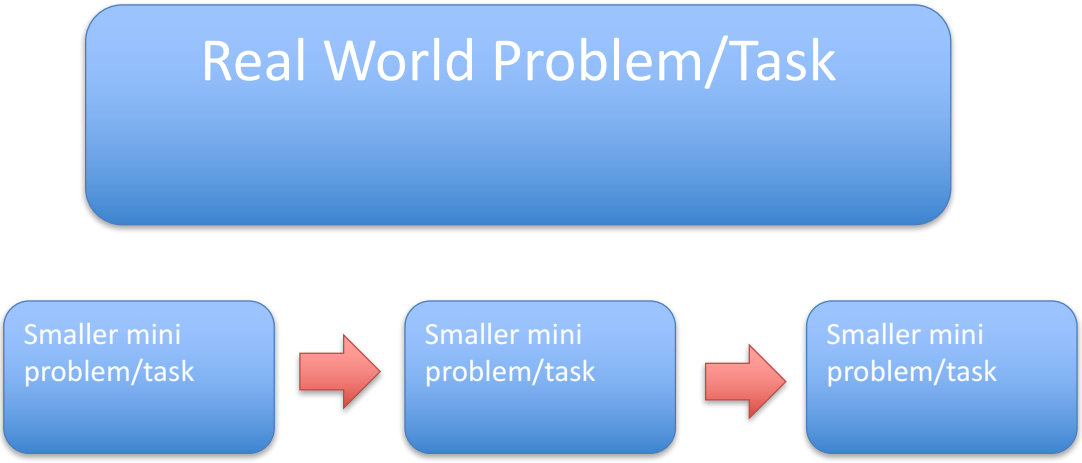
Split the problem into smaller subtasks. Solve those smaller subtasks. These will be our **functions.**

Motivation for Functions



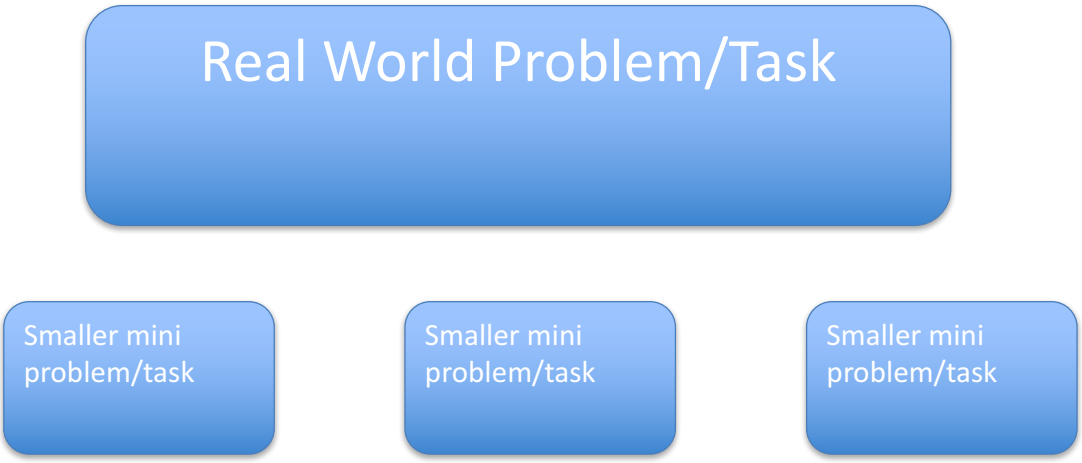
This can go on many levels down

Motivation for Functions



Sometimes we need to solve these tasks in a specific order (i.e. the input of one function relies on the output of another one.)

Motivation for Functions



Other times, we just need them available (they might not even be used in a specific program run). You see this with menu options.

Create a program that allows you to add money to your wallet. The program should keep adding money until it hits a specified goal.

Split the problem into smaller subtasks. Solve those smaller subtasks. These will be our **functions.**

Calculate total money in wallet

If goal was hit, did you go over?

Note: this is in C++, not C. The concept is the same

Write lines of code that do this task.

Write lines of code that do this task.

```
int wallet_total(std::vector <int> money)
{
   if(money.size()==0)
   {
    std::cout<<"Nothing in your wallet yet."<<std::endl;
    return 0;
   }
   int total=0;
   for(int i=0;i<money.size();i++)
   {
    total+=money[i];
   }
   return total;
}
```

```
bool check_total(int total, int goal)
{
 bool b;
 if(total<=goal)
 {
  std::cout<<"You hit your goal without going over."<<std::endl;
  b= true;
 }
 else
 {
  std::cout<<"You went over your goal."<<std::endl;
  b= false;
 }
 return b;
}
```


Create a program that allows you to add money to your wallet. The program should keep adding money until it hits a specified goal.

We can now solve the problem and use the functions…

```
int main(int argc, char **argv)
{
 int goal;
 int add;
 std::vector <int> wallet;

 std::cout<<"Enter your goal amount:"<<std::endl;
 std::cin>>goal;

 while(wallet_total(wallet) <goal)
 {
  std::cout<<"Enter amount to add to wallet: "<<std::endl;
  std::cin>>add;
  wallet.push_back(add);
 }

 check_total(wallet_total(wallet),goal);
}
```

Calculate total money in wallet

Note: this is in C++, not C. The concept is the same

If goal was hit, did you go over?

Create a program that allows you to add money to your wallet. The program should keep adding money until it hits a specified goal.

Notice by using the functions how much cleaner, direct and easier to understand our code is. We wrap up lines of code in a function, give it a name and use it by the name

```
int main(int argc, char **argv)
{
 int goal;
 int add;
 std::vector <int> wallet;

 std::cout<<"Enter your goal amount:"<<std::endl;
 std::cin>>goal;

 while(wallet_total(wallet) <goal)
 {
  std::cout<<"Enter amount to add to wallet: "<<std::endl;
  std::cin>>add;
  wallet.push_back(add);
 }

  check_total(wallet_total(wallet),goal);
}
```

```
int wallet_total(std::vector <int> money)
{
  if(money.size()==0)
  {
   std::cout<<"Nothing in your wallet yet."<<std::endl;
   return 0;
  }
  int total=0;
  for(int i=0;i<money.size();i++)
  {
   total+=money[i];
  }
  return total;
}
```

Note: this is in C++, not C. The concept is the same

```
bool check_total(int total, int goal)
{
 bool b;
 if(total<=goal)
 {
  std::cout<<"You hit your goal without going over."<<std::endl;
  b= true;
 }
 else
 {
  std::cout<<"You went over your goal."<<std::endl;
  b= false;
 }
 return b;
}
```


A final consideration for today is the following:

- Notice the fact that solving a problem usually means reducing a problem to information/data and ways to manipulate (actions on) that information/data
- We store the information/data in variables
  - If we have many variables, we can keep them in data structure
- We can then perform actions on these variables
  - It can be a few lines of code
  - If the lines of code combine to do one task, we can store them in a function.

- I will give a possible way on how to think to split up problems
  - You don't have to do it this way, just an idea

Motivation for Functions

Real World Problem/Task

1. **What info/data are we dealing with?**
2. **Is there a lot? Should we use a data structure?**

Smaller mini problem/task

Smaller mini problem/task

Smaller mini problem/task

Motivation for Functions

Chef Francois has four spots for ingredients in his pantry. Create a program that allows him to enter four ingredients and then allow his sous chef to check if a certain ingredient is in the pantry.

Think about

1. **What info/data are we dealing with?** Strings (ingredient names)
2. **Is there a lot? Should we use a data structure?** Yes-an array

Enter ingredients

Check if an ingredient is available

Let user know if ingredient is available or not

Do we need a function?
Yes

Do we need a function?
Yes

Do we need a function?
No

Motivation for Functions

Thinking in terms of info/data and doing stuff with that:

Enter ingredients	Check if an ingredient is available	Let user know if ingredient is available or not
Info/data type: strings Stored in array. **Actions:**Go through array and input ingredients.	**Info/data type:** strings Stored in array. **Actions:** Go through array and check if something is present	No need for function
What do we need to do this? An array and size (parameter)	**What do we need to do this?** An array, size, word looking for (string) (parameters)	
Return? Nothing	**Return?** Nothing	
Function declaration: void enter_info(char a [][20], int size)	**Function declaration:** void check (char food_name, char [][20], int size)	

Enter ingredients

Info/data type: strings
Stored in array.
Actions: Go through array and input ingredients.

What do we need to do this?
An array and size (parameter)

Return? Nothing

Function declaration:
void enter_info(char a [][20], int size)

This is our input (parameter) to the function-what do we need so our function works?

This is our return type for the function-what does the function give us back when it is done?

What goes in-what do we need to do this? (parameter)

Enter ingredients

Actions:
go through array
Input ingredients

What comes out (return)

Create a program to get the a number from a user. Calculate the square root of this number.

Get user input

Calculate square root

Output this value to screen

Do we need a function?
We already have one
in the C standard library

It is declared in the
header **stdio.h:**

int scanf(const char
*format, ...)

Do we need a function?
We already have one
in the C standard library

It is declared in the
header **math.h:**

double sqrt (double x)

Do we need a function?
We already have one
in the C standard library

It is declared in the
header **stdio.h:**

int printf(const char
*format, ...)

- There is such a thing as function overkill
  - If you have 20 functions and they each have one line of code, you are probably not breaking up the problem effectively
  - If you have a function that has one single function inside, it might be overkill
    - That doesn't mean you won't ever have simple functions with few lines of code
      - For example, a function that squared a number would be pretty simple
- A good rule of thumb is making sure each function does a meaningful task
  - Just printing out a word is not a good reason for a function
  - Different people will see different ways to break up a problem

Final notes:

- Solving a smaller task is a much simpler undertaking
- It makes it easier to work in groups (which you will inevitably be doing in the future)
  - Each person takes a different function
- We can re-use code
  - If we have to solve a task, maybe there is already a function available for us to use

Anatomy of a function:

- Let's discuss the general anatomy of a function

- Declarations
- Definitions
- Parameters
- Arguments
- Return Types

There are 2 ways to write functions in your code (you will see me doing both in this class):

1

```
#include <stdio.h>

void exp_function(int base, int power)
{
        int i=0;
        int answer=1;

        for(i=0;i<power;i++)
        {
                answer=answer*base;
        }

        printf("answer: %d\n", answer);
}

int main (int argc, char **argv)
{

        exp_function(3, 4);
        exp_function(5, 3);

}
```

Function definition- basically all lines of the actual function

2

Function declaration: Overview of the function

```
#include <stdio.h>

void exp_function(int base, int power);

int main (int argc, char **argv)
{
        exp_function(3, 4);
        exp_function(5, 3);
}

void exp_function(int base, int power)
{
        int i=0;
        int answer=1;

        for(i=0;i<power;i++)
        {
                answer=answer*base;
        }

        printf("answer: %d\n", answer);
}
```

```
#include <stdio.h>

void exp_function(int base, int power);

int main (int argc, char **argv)
{
        exp_function(3, 4);
        exp_function(5, 3);
}

void exp_function(int base, int power)
{
        int i=0;
        int answer=1;

        for(i=0;i<power;i++)
        {
                answer=answer*base;
        }

        printf("answer: %d\n", answer);
}
```

Argument-when you actually use the function and put values in-
in this example, 3 and 4 are arguments for the first function call
5 and 3 are arguments in the second function call

Return Type-What type of value the function returns

Parameter -part of the definition. Think of them as reminders to the user that they need to actually give values when using the function.

A quick way to know if it is a parameter is whether or not a variable type (such as int) is present before a variable name.


```
#include <stdio.h>

void exp_function(int base, int power);

int main (int argc, char **argv)
{
        exp_function(3, 4);
        exp_function(5, 3);
}

void exp_function(int base, int power)
{
        int i=0;
        int answer=1;

        for(i=0;i<power;i++)
        {
                answer=answer*base;
        }

        printf("answer: %d\n", answer);
}
```

Remember: main is also a function-it has a return type (int) and parameters. Sometimes, you will see main written without a return type or parameters:

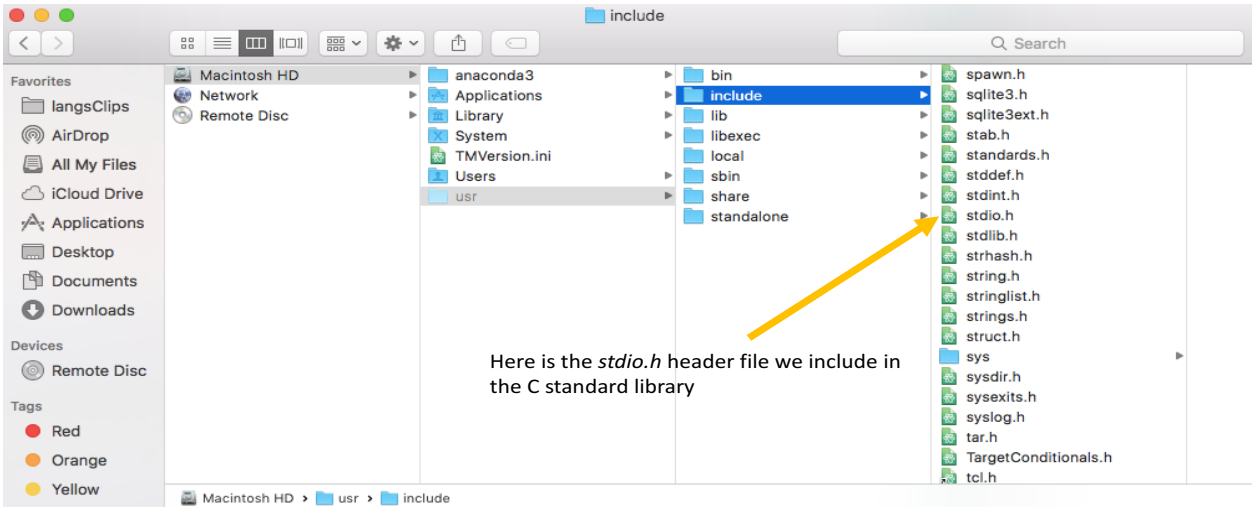
```
#include<stdio.h>

main(void)
{
   printf("Hello World");

}
```

Keeping our functions:

- Remember when we were using functions made by other people, we gained access to them by including the header
    - The header contains the function declaration
    - That allows us access to the function-for example, when I want to access *printf* and I include the *stdio.h* header, the header is located in my computer (and by including it, the program starts knowing to look for information with that specific file)



- For now, we will be creating our functions in the same file as the main (as shown previously in the lecture)
- In the future, you will be making your own headers and including them instead

Before we code:

- It doesn't matter what you named your parameters (just use that same name in the function)
    - Remember, when inside your function, you can't "see" other variables in other functions (like in main)
    - This is something called "scope"-we'll talk about it next lecture
- The name of the argument DOES NOT have to match the name parameter
- Output of the functions is given by the return statement
- You can save the return value (int n= add_num(3);) in a variable or use it directly
    - For example, remember I could print out the return value of the function or use it when adding to another int (so the return value is part of an operation with +): int n=10+add_num(3);
- I also tried out functions individually to make sure they did what they were supposed to do

Class Code:

Simple programs showing functions

Print out numbers

Sample Run:

```
Enter a number: 4
New value (printed in the function): 6
New value: 6
------------------------
```

```
#include <stdio.h>

/*This function takes an int, adds 2 and returns the input+2*/
int add_two(int x)
{
  int num=x+2; /*num does NOT exist outside of this function-I can't use num in main for example*/
  printf("New value (printed in the function) : %d\n",num);
  return num; /*I'm returning the VALUE of num, NOT the actual variable itself*/
}


int main(int argc, char **argv)
{
   int number=3;
   int new_value=add_two(number); /*the add_two function is passing the VALUE of number (3 in this case) NOT the actual variable number.  Inside the add_two function, we can't access the number variable here (but I could make a new variable called num in the function if I wanted*/
   printf("New value: %d\n", new_value); /*I could also do: printf("New value: %d\n", add_two(number)); to use the return value once*/
}
```

2d arrays of chars (printing out strings)

Sample Run:

```
Enter a word at least 5 letters long: dog
The word must be at least 5 letters. dog is only has 3 letter(s). Enter again: cat
The word must be at least 5 letters. cat is only has 3 letter(s). Enter again: basketball

Enter a letter to check for: b
Total letters: 2
```

Candy!

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

/*enter word of length 5.  Remember that arrays passed into functions can be modified directly so we don't need to return anything-whatever change we do the array will stay.  If I tried to modify an integer passed in, the modification would only stay during the function run-when I returned to main, it wouldn't stay.  This idea has to do with a concept called pass by value vs pass by reference-you will learn about it in 1320 (but feel free to look it up now if you are interested!)*/
void enter_word(char user_input[])
{
 printf("\nEnter a word at least 5 letters long: ");
 scanf("%s",user_input);

 while(strlen(user_input)<5)
 {
  printf("The word must be at least 5 letters. %s is only has %lu letter(s). Enter again: ", user_input,
strlen(user_input));
  scanf("%s",user_input);
 }
}

/*This function takes a string and a letter to look for and returns the number of instances that the letter occurs.  For example, if we had int n=check_letter("cat in the hat", 't'); the value of n would be 3 since 3 occurs in the given string 3 times*/
int check_letter(char str[], char letter)
{
 int i, counter=0;

 for(i=0;i<strlen(str);i++)
 {
  if(str[i]==letter)
  {
   counter++;
  }
 }

 return counter; /*returns the value of counter (the number of times we found the letter)*/
}


int main(int argc, char **argv)
{
 char answer[20];
 char letter;

 enter_word(answer);

 printf("\nEnter a letter to check for: ");
 scanf(" %c",&letter);
```

```
  int total_letters=check_letter(answer, letter);
  printf("Total letters: %d\n",total_letters);

  if(total_letters<5)
  {
    printf("Candy!\n");
  }

  else
  {
    printf("Potato!\n");
  }

}
```