# Lecture 8: Header files, functions, and other data types

## A simple header file

In the last lecture we saw how to declare a function in C in the same code in which it is used. For example, if we wanted to write a function that counted the number of white spaces in a string, then we would use it within the same code as

```
typedef char *string;

int CountWhiteSpaces(string inputString);

int main(void) {
  string myString = "hello this is a string";
  printf("The string has %d spaces\n",CountWhiteSpaces(myString));
}

int CountWhiteSpaces(string inputString) {
  int i, count=0;
  for(i=0;i<strlen(inputString);i++)
    if(inputString[i]==' ')
      count++;
  return count;
}
```

Often what we would like to do is to create separate files that contain our functions and have those functions be used by other codes. We can easily do this by creating another file called `whitespace.c` that contains the code for our function, as in

```
// File: whitespace.c
#include "whitespace.h"

int CountWhiteSpaces(string inputString) {
  int i, count=0;
  for(i=0;i<strlen(inputString);i++)
    if(inputString[i]==' ')
      count++;
  return count;
}
```

The included file contains the function prototype and the new type definition

```
// File: whitespace.h

typedef char *string;

int CountWhiteSpaces(string inputString);
```

because this type is needed in both `main.c` and `whitespace.c`. We can then compile this simple code and create an object file for it with

```
$ gcc -c whitespace.c
```

Now we can create another file that has our main code in it that calls the function `CountWhiteSpaces` as:

```
// File: main.c
#include "whitespace.h"

int main(void) {
  string myString = "hello this is a string";
  printf("The string has %d spaces\n",CountWhiteSpaces(myString));
}
```

We can create another object file with

```
$ gcc -c main.c
```

and we can link the two object files with

```
$ gcc main.o whitespace.o
```

We can also compile the files together with

```
$ gcc main.c whitespace.c
```

which will create an executable that when run will produce

```
$ ./a.out
The string has 4 spaces
```

The compiler links the two codes so that the executable `CountWhiteSpaces` that is used in `main.o` is found in `whitespace.o`.

## Header files and the preprocessor

Sometimes you may be working with several source codes and some may include header files which include the same header files. If, for example, you included `whitespace.h` twice in your `main.c` code, it would generate an error because the compiler would read the resulting code as attempting to define the `string` type twice. In order to avoid conflicts like this, you can make sure that a given include file is only included in a particular source code once with the `#ifndef` preprocessor statement. This is a preprocessor directive that is an if statement and is used as follows

```
#ifndef VARIABLE
#define VARIABLE

// Write your header file here.

#endif
```

The preprocessor defines certain variables with the `#define VARIABLE` directive and it can determine if a variable has already been defined with the statement `#ifndef VARIABLE` . Therefore, these preprocessor directives tell the preprocessor only to include the header file if the specified variable has not been defined, since if it has been defined that means that the header file has already been included in some other part of the code. The standard practice is to let the variable of a header file like `whitespace.h` be `_whitespace_h`, since it is doubtful that this variable will be defined anywhere else in your code. The new header file `whitespace.h` will then be given by

```
// File: whitespace.h
#ifndef _whitespace_h
#define _whitespace_h

typedef char *string;

int CountWhiteSpaces(string inputString);

#endif
```

You can also use these preprocessor directives in your code to tell the compiler to compile a different code based on your definitions. As an example, consider the code

```
#define PRINTHELLO

int main(void) {

  #ifdef PRINTHELLO
  printf("Hello world!\n");

  #else
  printf("Goodbye world!\n");

  #endif
}
```

Since `PRINTHELLO` **is** defined in the first line, the preprocessor will convert this code into:

```
int main(void) {
  // PRINTHELLO is defined.
  printf("Hello world!\n");
}
```

and compile it. But if the `#define PRINTHELLO` line is either commented out or removed, the preprocessor will compile the code as

```
int main(void) {
  // PRINTHELLO is not defined.
  printf("Goodbye world!\n");
}
```

It is important to remember that the preprocessor processes comments before it processes any # statements.

## Private functions

In the previous example we created the function `CountWhiteSpaces` which was used in `main.c`. This is because the function was automatically made external to the object file, which is the default. That is, when you define a function in a source file it is automatically linked with other functions that are compiled with that source code. We can alternatively create functions that are private to a given object file so that no other object file has access to that function. As an example, suppose we modifed the `whitespace.c` program so that it contained another function called `IsSpace`, which returns a `1` when a character is a space, and `0` otherwise. But if we want to only use that function within the source code in which it is written, then we declare it as a `static` function. The following source shows our new `whitespace.c` program:

```
// File: whitespace.c
#include "whitespace.h"

/*
 * Local function declarations go inside the source
 * and not the header file!
 *
 */
static int IsSpace(char c);

int CountWhiteSpaces(string inputString) {
  int i, count=0;
  for(i=0;i<strlen(inputString);i++)
    if(IsSpace(inputString[i]))
      count++;
  return count;
}

static int IsSpace(char c) {
  if(c==' ')
    return 1;
  return 0;
}
```

Using the `static` type declaration ensures that when we declare this function as static that it will not conflict with any other declarations of the function `IsSpace`. This way we can define another version of `IsSpace` in `main.c` and be guaranteed that it does not conflict with the definition in `whitespace.c`. This is why we don't put the `static` function declarations in the header file. Since they are not available to be linked with other object files, then there is no need for someone who might be using your object file to know about the statically declared functions.

## Variable scope

### Scope of variables between files

By default if you declare the same global variable in two files then that variable is visible to both files in which it is defined, and therefore to all functions within both of those files. For example, if you declared a global variable within both `main.c` and `whitespace.c`

```
int iglobal;
```

then if you change that variable in the function `IsSpace` then that change will be reflected within the `main` function.

If you wanted to declare the same variable in a bunch of files, but you wanted that variable only to be global to each file, then you would declare it as a static variable with

```
static int iglobal;
```

With this declaration, changes to the variable `iglobal` are only visible to the functions declared in each file.

### Scope of variables between functions

We already saw how if we declare a global variable in a file that it is visible by all of the functions within that file. If we declare variables within functions, we know that by default the scope of these variables is only local to the function that declares them. That is, we can be sure that if we define a variable like `int i;` in `CountWhiteSpaces` that its value will not change when the same variable name is changed in `IsSpace`. So when a value is set within a function, when that function exits that value is lost. C provides the capability of saving the value of a variable in a function even after that function finishes executing with the `static` declaration within a function. For example, we can define a static integer within the `IsSpace` function that counts the number of times the function is used

```
static int IsSpace(char c) {
  static int count=1;
  printf("Function has been used %d times\n",count++);

  if(c==' ')
    return 1;
  return 0;
}
```

Since the `count` variable is a static variable, then it is only initialized once during the first call to the function, and hence each time we operate on that variable its new value will be stored. If we don't declare a variable as `static`, then each time the function is called it reinitializes the value to some nonsensical number. By default `static` variables are initialized to `0`. Because `static` variables are initialized upon the first call to the function, you cannot initialize static arrays in the heap. For example,

```
static float a[10];
```

is okay, but

```
static float *a = malloc(10*sizeof(float));  // NOT OKAY!
```

is not.

## Other data types

### Structures

We have already seen how we can define our own types with the `typedef` statement, and we have also seen how we can use the `static` declaration to store values in functions for further use each time the functions are called. Both of these are different ways of defining a specific type of variable. C provides us with the capability of defining structures which contain a list of variables that we can access by their name. For example, we can define a structure that stores information about a student with

```
struct studentT {
  string name;
  string IDNumber;
  int graduationYear;
  int testGrade;
};
```

So we can define a new student using this structure as a tag with

```
struct studentT newStudent;
```

We can initialize every element of the structure with

```
struct studentT student1 = {"Yoo Doubleusee","994999",2003,8};
```

or we can access each member of the structure individually and set it that way with

```
student1.name = "Yoo Doubleusee";
student1.IDNumber = "994999";
student1.graduationYear = 2003;
student1.testGrade = 8;
```

Structures can contain arrays as well, such as arrays of grades, as in

```
struct gradeT {
  string tests[10];
  string exams[2];
  int assignments[8];
};
```

and these are accessed with

```
struct gradeT student1grade;
student1grade.tests[0]=10;
        ...
```

## Structures and typedef

Rather than using the structure tag every time we want to declare a variable, we can conveniently use the `typedef` statement and declare our own variable type as a `struct`. Recall that the syntax of declaring your new variable with the `typedef` command is

```
typedef existingtype newtype;
```

Applying this to the above `studentT` struct, we have

```
typedef struct studentT {
  string name;
  string IDNumber;
  int graduationYear;
  int testGrade;
} studentTStruct;
```

Which says that now `studentTStruct` is a new type and it is of type `struct studentT`. We can then define new variables with this type with

```
studentTStruct student1;
```

which is identical to typing

```
struct studentT student1;
```

but turns out to be syntactically much more convenient when it comes to pointers.

Once we define a struct, rarely will we ever use the structure tag again, so it is standard practice to leave out the structure tag when we define a new type, as in

```
typedef struct {
  string name;
  string IDNumber;
  int graduationYear;
  int testGrade;
} studentT;
```

This way it makes for a clean type definition of a structure that we can use to define new variables with

```
studentT student1;
```

New structure types are almost always defined in conjunction with the `typedef` declaration.

### Arrays of structs

Just as you can define arrays of type `int` or `float`, you can also define arrays with a new type you created as a structure. For example, if there are 35 students in a class, we can have an array of 35 `studentT` types, each of which corresponds to one of the 35 students, as in

```
studentT students[35];
```

We can use this array in a loop, then, to print out the names, ID numbers, and grades of the students, as in

```
int Nstudents=35;
for(i=0;i<Nstudents;i++)
  printf("Student: %s, ID: %s, Grade: %d",
    students[i].name,
    students[i].IDNumber,
    students[i].testGrade);
```

## Enumerated types

In some instances we have a variable that we can assign a given limited number of values to. For example, we might have a program that asks users which city they visit most often given the following list: Cape Town, Bellville, Goodwood, or Parow. One option would be to define a variable `city` that is of type `int` and assign an integer to each of the cities in the list. That is, if `city==0`, then we know it corresponds to Cape Town. A much cleaner way to do this is to declare an enumerated type that assigns integers to specific names for us. In this case we can define the variable `city` as an enumerated type which can take on the values represented by each city as

```
enum cityTag {
  CapeTown, Bellville, Goodwood, Parow
} city;
```

Just like structures, enumerated types have a tag, which in this case is called `cityTag`, that can be used to define more variables of the same enumerated type, such as

```
enum cityTag city1, city2, city3;
```

But it is more convenient to define a new type and leave out the tag with `typedef`, as in

```
typedef enum {
  CapeTown, Bellville, Goodwood, Parow
} cityT;
```

This defines a new type `cityT`. Enumerated types do nothing other than make a code much more readable, since they really only replace the values in the declaration with integers. As an example, consider the following code, which asks a user to input the city they visit most often:

```
typedef enum {
  CapeTown, Bellville, Goodwood, Parow
} cityT;

int main(void) {
  cityT city;

  printf("Which city? (0=Cape Town, 1=Bellville, 2=Goodwood, 3=Parow): ");
  scanf("%d",&city);

  printf("You entered: ");
  switch(city) {
  case CapeTown:
    printf("Cape Town\n");
    break;
  case Bellville:
    printf("Bellville\n");
    break;
  case Goodwood:
    printf("Goodwood\n");
    break;
  case Parow:
    printf("Parow\n");
    break;
  }
}
```

As you can see, the code can be much easier to read because we can use code like `case CapeTown` rather than `case 0`, which would produce identical results.

We can also assign different values to the names in the enumerated type, such as

```
typedef enum {
  CapeTown=10, Bellville=20, Goodwood=30, Parow=40
} cityT;
```

so that if we declare something like `cityT city=CapeTown`, this is equivalent to declaring `cityT city=10`, only that it is much easier to understand. It is also possible to assign the values as characters since each character can be converted into a unique integer. As an example, we could use

```
typedef enum {
  CapeTown='c', Bellville='b', Goodwood='g', Parow='p'
} cityT;
```

Using this enumerated type, the declaration `cityT city=CapeTown` is identical to `CityT city='c'`.