Programming with C

Terry Marris December 2010

17 Structures

In the previous chapter we looked at searching through arrays and sorting them into some kind of order. Now we move on to structures in C. But first we look at *typedef*.

17.1 Typedef

A *typedef* creates a new data type name for an existing type. For example, the declaration

```
typedef char *String;
```

makes *String* another name for *char **.

We can define a variable of type *String*

```
String name;
```

But we have to explicitly allocate memory for it.

```
String name;
name = malloc(25);
```

17.2 Structures

A *structure* allows us to group data items together. For example, a person *has* a name, a gender and an age. We can write:

```
struct person {
  char name[25];
  char gender
  int age;
};
```

struct introduces the structure. Our structure has three *members*: *name*, *gende*r and *age*. *person* is an optional tag that names the structure.

A *struct* declaration defines a new type. We can name this type with a *typedef*.

```
typedef struct person {
  char name[25];
  char gender;
  int age;
} Person;
```

Here, we have named the structure *Person*.

We can declare variables of this type,

```
Person student;
```

and, in doing so, space in memory is reserved for the structure and its member variables.

We can give *student* its values at the point of declaration

```
Person student = { "Jo King", 'f', 21 };
```

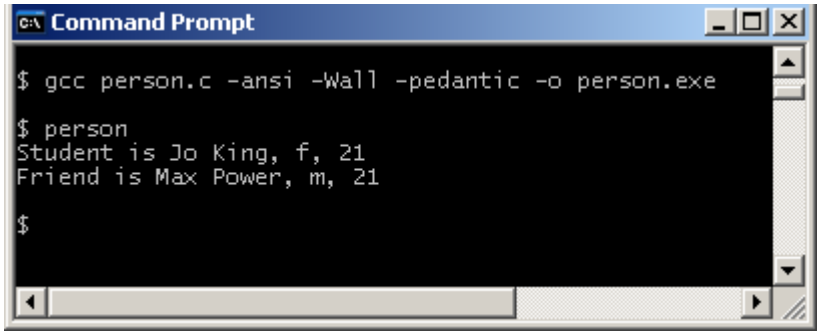
Or we can give *student* its *name*, *gender* and *age* line-by-line.

```
strcpy(student.name, "Jo King");
student.gender = 'f';
student.age = 21;
```

The structure member operator . (dot) selects a member of the structure.

We need an example program. Here it is.

```
/* person.c: creates a Person structure */

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

int main()
{
  typedef struct person {
    char name[25];
    char gender;
    int age;
  } Person;

  Person student = { "Jo King", 'f', 21 };
  Person friend;

  strcpy(friend.name, "Max Power");
  friend.gender = 'm';
  friend.age = 21;

  printf("Student is %s, %c, %d\n",
            student.name, student.gender, student.age);
  printf("Friend is %s, %c, %d\n",
            friend.name, friend.gender, friend.age);
  return 0;
}
```



17.3 Pointers and Structures

We introduce pointers to structures. First, we need a structure:

```
typedef struct tagPerson {
  char name[25];
  char gender;
  int age;
} Person;
```

Then we declare a pointer to that structure.

```
Person *student;
```

The variable *student* is of type pointer to *Person*.

As usual with pointers, we need to physically allocate memory to the variable.

```
student = malloc(sizeof(*student));
```

Notice we have used **pointer* to refer to what student points to as the argument to *sizeof()*. Another way of doing this is to use the tag as the argument value.

```
student = malloc(sizeof(struct tagPerson));
```

We refer to the members of structure with the `->` operator.

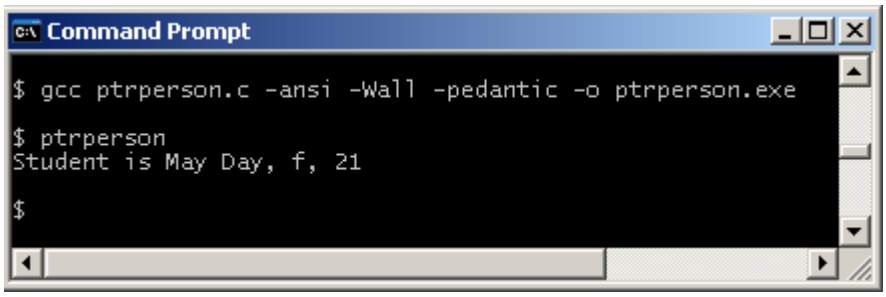
```
strcpy(student->name, "May Day");
student->gender = 'f';
student->age = 21;
```

and

```
  printf("Student is %s, %c, %d\n",
  student->name, student->gender, student->age);
```

Shown below is the entire program and its run.

```
/* ptrperson.c: illustrates pointers and structures */

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

int main()
{
  typedef struct tagPerson {
    char name[25];
    char gender;
    int age;
  } Person;

  Person *student;

  student = malloc(sizeof(*student));
  strcpy(student->name, "May Day");
  student->gender = 'f';
  student->age = 21;

  printf("Student is %s, %c, %d\n",
        student->name, student->gender, student->age);

  return 0;
}
```

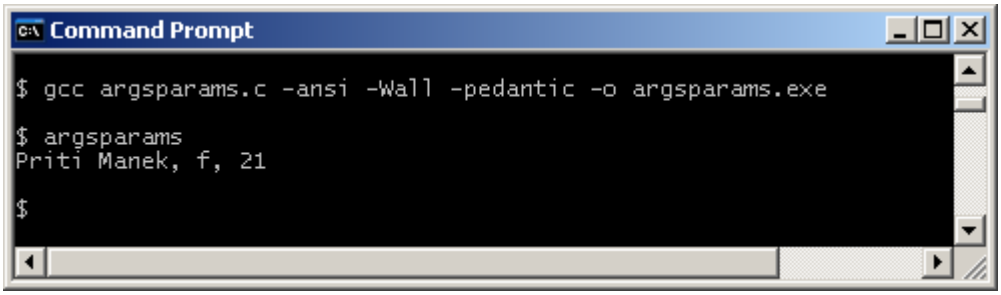


17.4 Structures as Arguments, Parameters and Return Values

Passing structure variables as arguments to function parameters, and returning them as function values, presents no surprises.

```
/* argsparams.c: uses structures as arguments and parameters */

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

typedef struct tagPerson {
  char name[25];
  char gender;
  int age;
} Person;
```

```
/* getPerson: initialises and returns a Person structure */
Person getPerson()
{
  Person person;
  strcpy(person.name, "Priti Manek");
  person.gender = 'f';
  person.age = 21;
  return person;
}

/* printPerson: displays person's attributes */
int printPerson(Person person)
{
  return printf("%s, %c, %d\n",
              person.name, person.gender, person.age);
}

int main()
{
  Person friend;
  friend = getPerson();
  printPerson(friend);
  return 0;
}
```



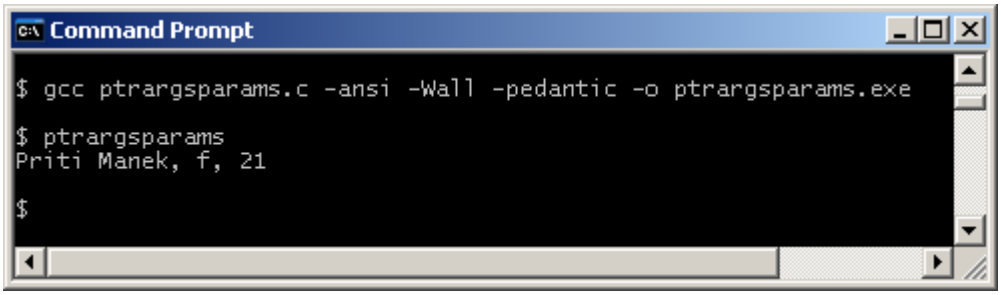
You can assign one structure variable to another no problem.

Notice that the structure *Person* is defined outside *getPerson()*, *printPerson()* and *main()*. This means that each of the three functions have access to the structure. We say that the structure has *global scope* - it can be accessed from anywhere within the program.

In the next program we return a pointer to a structure, and pass a pointer to a structure.

```
/* ptrargsparams.c: uses pointers to structures as arguments */

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

typedef struct tagPerson {
  char name[25];
  char gender;
  int age;
} Person;
```

```
/* getPerson: initialises a pointer to Person structure */
Person *getPerson()
{
  Person *person = malloc(sizeof(*person));
  strcpy(person->name, "Priti Manek");
  person->gender = 'f';
  person->age = 21;
  return person;
}

/* printPerson: displays person's attributes */
int printPerson(const Person *person)
{
  return printf("%s, %c, %d\n",
              person->name, person->gender, person->age);
}

int main()
{
  Person *friend;
  friend = getPerson();
  printPerson(friend);
  return 0;
}
```



In *main()* we declare a pointer variable to the *Person* structure, and assign to it the value returned by *getPerson()*.

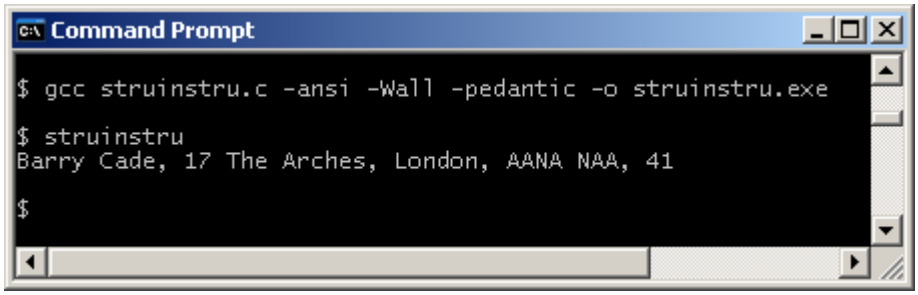
getPerson() reserves memory for the structure, initialises each member and then returns the pointer variable.

printPerson() receives a pointer to a *Person* structure in its parameter, and displays the contents of each member. Incidentally, *printf()* returns the number of characters written.

17.5 Structures Within Structures

A person *has* a name, an address and an age. An address *has* a house number with street, a city and a postcode.

```
/* struinstru.c: features a structure within a structure */

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

typedef struct tagAddress {
  char street[20];
  char city[15];
  char postCode[12];
} Address;

typedef struct tagPerson {
  char name[15];
  Address address;
  int age;
} Person;

/* getPerson: initialises and returns a person */
Person getPerson()
{
  Person person;
  strcpy(person.name, "Barry Cade");
  strcpy(person.address.street, "17 The Arches");
  strcpy(person.address.city, "London");
  strcpy(person.address.postCode, "AANA NAA");
  person.age = 41;
  return person;
}

/* printPerson: displays a person's attributes */
int printPerson(Person person)
{
  return printf("%s, %s, %s, %s, %d\n",
         person.name, person.address.street, person.address.city,
       person.address.postCode, person.age);
}

int main()
{
  Person friend;
  friend = getPerson();
  printPerson(friend);
  return 0;
}
```



We refer to a structure member within a structure member by, for example,

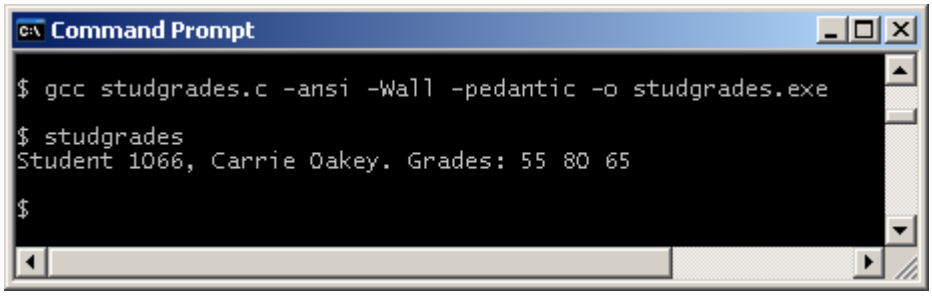
```
strcpy(person.address.city, "London");
```

This says that *city* is a member of *address* and *address* is a member of *person*.

17.6 Arrays as Structure Members

A student *has* a number, a name and a course, and a collection of up to three grades for assignments completed. We hold the three grades in an array.

```
/* studgrades.c: features an array within a structure */

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

typedef struct tagStudent {
  char number[10];
  char name[15];
  int grades[3];
} Student;

/* getStudent: initialises and returns a student */
Student getStudent()
{
  Student student;
  strcpy(student.number, "1066");
  strcpy(student.name, "Carrie Oakey");
  student.grades[0] = 55;
  student.grades[1] = 80;
  student.grades[2] = 65;
  return student;
}

/* printStudent: displays student attributes */
int printStudent(Student student)
{
  int i;

  printf("Student %s, %s. Grades: ", student.number, student.name);
  for (i = 0;
       i < (sizeof student.grades / sizeof student.grades[0]); i++)
    printf("%d ", student.grades[i]);
  printf("\n");
  return 0;
}

int main()
{
  Student student = getStudent();
  printStudent(student);
  return 0;
}
```



Lines worthy of note include

```
for (i = 0;
     i < (sizeof student.grades / sizeof student.grades[0]); i++)
  printf("%d ", student.grades[i]);
```

(sizeof student.grades / sizeof student.grades[0]) returns the size of the array, i.e. the number of elements it contains. We assume here that the array is completely filled with values.

17.7 Structure Arrays

A class attendance list is an array of students where a student has a name and a present mark: 0 if absent, 1 if present.

First, we define just one student.

```
typedef struct tagStudent {
  char name[15];
  int present;
} Student;
```

Then we define the list of students.

```
typedef struct tagList {
  Student student[3];
} List;
```

Here, we have specified that the list comprises just three students. Defined like this we have a structure with a member that is an array.

markList() gives each student in the list a name and whether they are present, or not.

```
/* markList: initialises the list of students present */
List markList()
{
  List list;
  strcpy(list.student[0].name, "Pearl Button");
  list.student[0].present = 1;

  strcpy(list.student[1].name, "Carrie Oakey");
  list.student[1].present = 0;

  strcpy(list.student[2].name, "Ann D'Pandy");
  list.student[2].present = 1;
  return list;
}
```

Then we go on to display the attributes of just one student.

```
/* printStudent: prints a single student's attributes */
int printStudent(Student student)
{
  return printf("%-12s %d\n", student.name, student.present);
}
```

%-12s in the format specification says print the string value left justified in a field width of 12 characters.

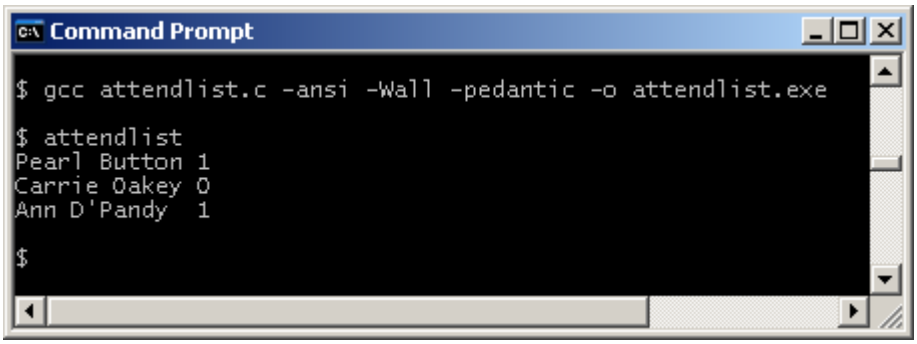
Now we can display the entire list of students.

```
/* printList: displays the list of students */
int printList(List list)
{
  int size = sizeof list.student / sizeof list.student[0];
  int i;

  for (i = 0; i < size; i++)
    printStudent(list.student[i]);
  return 0;
}
```

Here is the entire program and its run.

```
/* attendlist.c: features an array of structures */
#include <stdio.h>
#include <string.h>

typedef struct tagStudent {
  char name[15];
  int present;
} Student;

typedef struct tagList {
  Student student[3];
} List;

/* markList: initialises the list of students present */
List markList()
{
  List list;
  strcpy(list.student[0].name, "Pearl Button");
  list.student[0].present = 1;
  strcpy(list.student[1].name, "Carrie Oakey");
  list.student[1].present = 0;
  strcpy(list.student[2].name, "Ann D'Pandy");
  list.student[2].present = 1;
  return list;
}

/* printStudent: prints a single student's attributes */
int printStudent(Student student)
{
  return printf("%-12s %d\n", student.name, student.present);
}

/* printList: displays the list of students */
int printList(List list)
{
  int size = sizeof list.student / sizeof list.student[0];
  int i;

  for (i = 0; i < size; i++)
    printStudent(list.student[i]);
  return 0;
}
```

```
int main()
{
  List list = markList();
  printList(list);
  return 0;
}
```



As a general principle, when dealing with a collection of objects, we first provide functions for dealing with just one object, test them, then go onto provide functions that manage the collection.

17.8 Precedence

We see how the member selector operators fit in to the table of precedence: they have the highest priority.

Operator					*Description*	*Precedence*
( )	[ ]	->	.		brackets, array, member selector	highest
++	--	*	&	sizeof	increment, decrement, indirection, address	
*	/	%			times, divide, mod	
+	-				add, subtract	
<	<=	>	>=		relational operators	
==	!=				equality operators	
&&					logical and	
||					logical or	
?:					conditional operator	
=					assignment operator	lowest

Exercise 17.1

1. A library loan *has* a borrower number, an accession number and a date the copy borrowed is due for return. A book title may have several copies, the accession number identifies a particular copy. Write and test a program that creates a loan type, creates a variable of this type, assigns appropriate values to it, and then prints it out.

2. A borrower *has* a name, a number, and a collection of up to five loans. Write and test a program that creates a borrower and prints their attributes.

3. A book title *has* an author, a title, and a number of copies. A catalogue *has* a description and a collection of titles. Write and test a program that could be the beginnings of a library cataloguing system.

We have looked at structures in C. **Next** we look at text files.

Bibliography

Kernighan B and Ritchie D *The C Programming Language* Prentice Hall 1988 pp 135, 142, 145

Mark Williams Company *ANSI C A Lexical Guide* Prentice Hall 1988