Computer Science 1510

Lecture 29

Lecture Outline

- Strings
- Data structures

The char data type

- In C, characters are stored as one byte integers, but can be treated as characters when output.

- This allows us to do "character arithmetic".

- For example,

```
#include<stdio.h>

int main(int argc, char *argv[])
{
   char x;
   x='A';
   x++;
   printf("%c\n",x);
   return 0;
}
```

prints B.

Character strings

- In C, a character string is simply a one-dimensional array of characters, terminated by the null character. The null character is denoted by `\0`.

- Thus, an n character string requires $n + 1$ bytes of memory.

- Single characters are indicated by single quotes (ex. `'A'`).

- Character strings are denoted by double quotes; ex. "abc" and "Peter Piper" are character strings.

  The null character is not explicitly included in the string, but is added by the compiler.

- As we will see later, C has a number of functions that deal with strings, including one that returns the size (length) of a string, excluding the null character terminator.

  `strlen("abc")` returns 3.

Character strings

- It is worthwhile to note that not all arrays of characters are strings.

  It is possible to have character arrays without the null character terminating the array (this array would then not be a string.)

- Characters are single bytes, so they are often used (or misused) to store small numbers. In fact, because of this, C allows signed and unsigned characters, even though, logically, a character does not have a sign.

The C standard library

- Thus far, we have seen several functions available from the C standard library, as well as several from the math library.

- For example, `fopen`, `fclose`, `fprintf`, `printf`, `fscanf`, and `scanf`, are all functions that are declared in `stdio.h`, which is a part of the C standard library.

- It is beyond the scope of this course to examine all of the functions available in the C standard library, but we will look at a few.

C: fgets

- The `fgets` function reads a string from an open stream, up to a newline character.

- Syntax:

  ```
  fgets(char *s, size, FILE *fp)
  ```

  where `char *s` points to the array where the input string will be stored, `size` is the maximum number of characters to be read, and `FILE *fp` is a pointer to an open stream (ex. a file or `stdin`).

- The newline character is included in the array, and the array is terminated by the null character `\0`.

- The `sscanf` function can then be used to extract the desired elements from the string.

Example: fgets and sscanf

- Example:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int a,b,c,d;
   char buff[256];
   fgets(buff,256,stdin);
   sscanf(buff,"%d %d %d",&a,&b,&c);
   printf("%d, %d, %d\n",a,b,c);
   sscanf(buff,"%d",&d);
   printf("%d\n",d);
   return 0;
}
```

Input:

```
1 2 3 4
```

Output:

```
1, 2, 3
1
```

C: sprintf

- We have seen above how to use the `sscanf` function to read elements from a string.

- We also saw how to assign a string to a statically defined array of characters on the declaration line, and set a pointer to a string.

- How can we set a character array after the declaration line?

- The function required is `sprintf`, which is similar to `printf` except that it prints to a string rather than the screen.

- Syntax:

  ```
  sprintf(string, description, variable-list);
  ```

  where `string` is an array of characters where the output will be written, `description` is the format of the string to be written, and `variable-list` is a list of variables with matching conversion codes in `description`.

Example: sprintf

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int a,b,c;
   /* Declare a static string of length 256 */
   char buff[256];
   /* Set buff to the string "1 2 3 */
   sprintf(buff,"1 2 3");
   /* Read 3 ints from buff */
   sscanf(buff,"%d %d %d",&a,&b,&c);
   printf("%d, %d, %d\n",a,b,c);
   return 0;
}
```

Output:

```
1, 2, 3
```

String manipulation

- A portion of the C standard library is dedicated to string manipulation.
- Standard library functions associated with string manipulation are declared in `string.h`.
- We have seen how to read in strings using `fgets`, as well as how to extract and print to strings using `sscanf` and `sprintf` respectively.
- Now we will see how to copy strings, concatenate strings, compare strings, etc.

C: String copy – strcpy

- Syntax:

  ```
  strcpy(copy_of_string,string);
  ```

  where `string` is copied to `copy_of_string`, including the terminating null character.

- Example:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]){
   char buff1[1024];
   char buff2[1024];
   sprintf(buff1,"Hello");  /* Put Hello in buff1 */
   strcpy(buff2,buff1);  /* Copy buff1 to buff2 */
   printf("%s %s\n",buff1,buff2);
   return 0;
}
```

- Output: `Hello Hello`

C: String concatenation – strcat

- Syntax:

  `strcat(string1,string2);`

  where `string2` is concatenated to the end of `string1`.

- Example:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]){
   char buff1[1024];
   char buff2[1024];
   sprintf(buff1,"Hello");  /* Put Hello in buff1 */
   sprintf(buff2," there");  /* Put there in buff2 */
   strcat(buff1,buff2);
   printf("%s\n",buff1);
   return 0;
}
```

- Output: `Hello there`

C: String comparison – strcmp

- Syntax:

```
strcmp(string1,string2);
```

  returns an integer with value:

  < 0 if `string1 < string2`,
  $= 0$ if `string1 = string2`,
  > 0 if `string1 > string2`.

- Example:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
   char buff1[1024];
   char buff2[1024];
   int cmp;
   fgets(buff1,1024,stdin); /* Read in buff1 */
   fgets(buff2,1024,stdin); /* Read in buff2 */
   cmp=strcmp(buff1,buff2); /* Compare the two strings */
   printf("%d\n",cmp);
   return 0;
}
```

C: String tokenizer – strtok

- Syntax:

  ```
  strtok(string,delimiter);
  ```

  where `string` is the string to be tokenized, and `delimiter` is the character(s) on which to break the tokens. The `strtok` function returns a pointer to the token, or `NULL` when no further token is found.

- On the first call to strtok the string to be tokenized is the argument; on successive calls to retrieve more tokens in the same string, the argument NULL is used.

- A sequence of two or more contiguous delimiters is considered a single delimiter by the function.

C: String tokenizer – strtok

- Example:

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

int main(int argc, char *argv[]) {
    const int buff_size=256;
    char buff[buff_size];
    char *token;
    int i=0;
    printf("Enter a string\n");
    fgets(buff,buff_size,stdin);

    token = strtok(buff," ");
    while (token!=NULL) {
        printf("Token %d: %s\n",i++,token);
        token = strtok(NULL," ");
    }
    return 0;
}
```

- Output:

```
Enter a string
The lively fox chased the tabby cat.
Token 0: The
Token 1: lively
Token 2: fox
Token 3: chased
Token 4: the
Token 5: tabby
Token 6: cat.
```

Data structures

- We saw in Fortran how to group several variables together into a more complex structure.

- For example, creating a data type for a student which contains an ID number, a name, and marks for a particular course.

- The same idea is possible in C using a `struct`.

- Syntax:

```
struct name {
   type field;
   .
   .
   type field;
};
```

  where `name` is a tag used to refer to the structure, and `fields` are the individual elements comprising the structure.

Data structures

- Example:

```
struct student{
   char name[40];
   int id;
   float assign[10];
   float midterm, final, grade;
};
```

- The above definition simply describes what a `student` data type contains.

- To store student data we must first declare a `student`, or an array of students. For example,

```
struct student class[50];
```

  declares an array of 50 students, each having the fields defined above.

Data structures

- It is also possible to declare a structure on the definition line after the end brace. For example,

```
struct student{
   char name[40];
   int id;
   float assign[10];
   float midterm, final, grade;
} class[50];
```

  is a valid declaration for an array of 50 students called `class`.

- It is much more common to define the structure separately from the declaration, however.

  The separation of the structure definition and the declaration of an instance of the structure makes the code more readable.

Data structures

- A structure can be initialized by listing the initial values in order. For example:

```
struct point {
   float x;
   float y;
};
struct point p = {0.0, 1.0};
```

- The value of one structure can be assigned to another structure (the values of all members are copied):

```
struct mystruct {int i; float f; char s[4];};
struct mystruct s1={15,0.5,"hi"}, s2={0,1.0,"bye"};
s1=s2; /* s1 now = {0,1.0,"bye"} */
```

Accessing Members of Data structures

- Given a structure, you can access its individual elements using the "dot" (.) operator.

- For example, to access the first assignment mark of the first student we would have,

```
mark=class[0].assign[0];
```

- To access the second assignment mark of the fourth student:

```
mark=class[3].assign[1];
```

Accessing Members of Data structures

- Structures can also be referenced with pointers. For example,

```
struct student *st1=&class[3];
```

  is a pointer to a student structure, and here points to the fourth student in the structure.

- Given a pointer to a structure, you can access the individual elements of the structure using the "arrow" (->) operator.

- For example, to access the second assignment mark of the fourth student we could have,

```
struct student *st1=&class[3];
mark=st1->assign[1];
```

Data structures

- Consider the following structure:

```
struct point {
   float x;
   float y;
};
```

- We can declare and initialize a point as follows:

```
struct point pt={5.2,3.3};
```

- We can also nest structures. For example, we can define a rectangle by its lower left and upper right corner via

```
struct rectangle {
   struct point sw;
   struct point ne;
};
```

Example 1: struct

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   struct point {
      float x,y;
   };
   struct rectangle {
      struct point sw,ne; /* Lower left and upper right points */
   };
   struct point pt;  /* Point entered by user */
   struct rectangle r1,r2;
   printf("Enter lower left corner of rectangle 1\n");
   scanf("%f %f",&r1.sw.x,&r1.sw.y);
   printf("Enter upper right corner of rectangle 1\n");
   scanf("%f %f",&r1.ne.x,&r1.ne.y);
   printf("Enter lower left corner of rectangle 2\n");
   scanf("%f %f",&r2.sw.x,&r2.sw.y);
   printf("Enter upper right corner of rectangle 2\n");
   scanf("%f %f",&r2.ne.x,&r2.ne.y);

   printf("Enter coordinates of point to locate\n");
   scanf("%f %f",&pt.x,&pt.y);
   if (pt.x>=r1.sw.x && pt.x<=r1.ne.x && pt.y>=r1.sw.y && pt.y<=r1.ne.y)
   {
      if (pt.x>=r2.sw.x && pt.x<=r2.ne.x && pt.y>=r2.sw.y && pt.y<=r2.ne.y)
      {
         printf("Inside both rectangles\n");
      }
      else
      {
         printf("Inside rectangle 1\n");
      }
   }
   else if (pt.x>=r2.sw.x && pt.x<=r2.ne.x && pt.y>=r2.sw.y && pt.y<=r2.ne.y)
   {
```

```
      printf("Inside rectangle 2\n");
   }
   else
   {
      printf("Outside both rectangles\n");
   }
   return 0;
}
```

Data structures

- A structure is really a user defined data type. Simply defining a structure does not allocate memory for the structure; memory is allocated when a variable with the type of the structure is declared.

- The `sizeof` operator can be used with structures, just as it is used with the built-in data types:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   struct point {
      float x,y,z;
   };
   printf("size of structure = %zu \n", sizeof(struct point));
   return 0;
}
```

Output:

```
size of structure = 12
```