Computer Science 1510

Lecture 23/24

Lecture Outline

- Repetition
- Functions

Example: while loop

Calculating mean time to failure:

```
#include <stdio.h>

int main(int argc, char *argv[]){
   int numtimes;
   float ftime, sum, mean;
   const float end=-1.0;

   sum=0.0;
   numtimes=0;

   printf("Enter failure time (%f to stop): ",end);
   scanf("%f",&ftime);

   while (ftime != end) {
      numtimes++;
      sum = sum + ftime;
      printf("Enter failure time (%f to stop): ",end);
      scanf("%f",&ftime);
   }

   if (numtimes!=0){
      mean = sum/numtimes;
      printf("Read %d failure times\n",numtimes);
      printf("Mean time to failure is %f\n",mean);
   }
   else{
      printf("No failure times were entered\n");
   }

   return 0;
}
```

do while loop

- Syntax:

```
do{
    statement-block;
} while (logical-expression);
```

  where `logical-expression` is any valid C expression that evaluates to an integer that can be interpreted as true (non-zero) or false (zero).

- Example:

```
do{
    scanf("%d",&y);
    x++;
} while (x<10 && y!=5);
```

- In a `do while` loop, the first iteration is not dependent on a condition, ie. at least one iteration of the body of the loop is completed.

The break statement

- We have seen the use of the `break` statement in a `switch` block.

- The `break` statement can also be used within a loop to break out of the loop and begin executing from the first statement after the loop.

- For example,

```
while (logical-expression){
   statement-sequence1;
   if (test) break;
   statement-sequence2;
}
statement-sequence3;
```

- In this example, if `test` evaluates to true then `statement-sequence2` is skipped, and execution jumps to `statement-sequence3`.

The continue statement

- The `continue` statement is used within a loop to cause control to immediately return to the beginning of the loop.

- For example,

```
for (i=0;i<n;i++){
   statement-sequence1;
   if (test) continue;
   statement-sequence2;
}
```

- In this example, if `test` evaluates to true then `statement-sequence2` is skipped, and execution jumps to the beginning of the loop, where the increment is executed, and the loop condition is retested.

- If the end condition evaluates to true, then execution continues with `statement-sequence1`.

Functions

- We have used several C functions available in the `stdio` and `math` libraries.

- Now we will see how to write our own functions in C to allow us to divide a program up into smaller parts and/or write general purpose functions that may be reused later.

- Syntax:

```
return_type function_name(argument_list)
{
    variable_declarations;
    body_of_function;
    return expression;
}
```

  where `return_type` is the type of the value returned by the function, `function_name` is the name used to call the function, and `argument_list` defines the values that will be received by the function (from the calling function).

Functions

- Like Fortran functions, C functions return a value. However, the value returned does not have to be stored in a specific variable.

- In C, the `expression` in the `return` statement is the value that will be returned from the function, and can be any valid C expression.

- The `argument_list` takes the form,

  ```
  type1 iden1, type2 iden2,..., typeN idenN
  ```

  where `iden1`, `iden2`,. . . , are the variable names of the arguments to be passed into the function, and `type1`, `type2`,. . . , are the corresponding types of the arguments.

- Like in Fortran, C functions can be called from any location where a value of type `return_type` can be used, including in assignment statements and expressions.

Functions

- Variables defined in a C function are local to that function, and are not visible to the calling program or function.

- In C, a function must be declared before it can be used.

- One way to do this is to define the function before the main function, ie. include the entire function before the main function.

- An alternative method is to write a function *declaration* at the beginning of the file, and define the function at the end.

  This is of the form
  `return_type function_name(argument list);`

Example: Functions - Temperature conversion

```
#include <stdio.h>

float fahr_to_cel(float temp);

int main(int argc, char *argv[]){
   float fahr,cel;
   char next;

   do{
      printf("Enter a temperature in Fahrenheit\n");
      scanf("%f",&fahr);

      /* Convert the temperature to celsius */
      cel = fahr_to_cel(fahr);

      printf("%f in Fahrenheit is %f in celsius\n",fahr,cel);

      /* Remove newline character from buffer */
      scanf("%c",&next);

      printf("Are there more temperatures to convert? [y/n]\n");
      scanf("%c",&next);
    } while (next=='y');

    return 0;
}

float fahr_to_cel(float temp){
    return (temp-32.0)/1.8;
}
```

Type void

- The `void` data type is sometimes useful when writing functions.

- `void` is used when there is to be no value returned, and/or no arguments passed into a function.

- For example, if we declare a function like,

```
int myfunc1(void);
```

  which returns an integer, but takes no arguments, `myfunc1` could be called as follows:

```
a=myfunc1();
```

- We could also declare a function like,

```
void myfunc2(int a, float x);
```

  which takes two arguments, an `int` and a `float`, but returns nothing.

Pass-by-value / Pass-by-reference

- The function `myfunc2` raises the question of what a function that returns nothing would actually accomplish.

- In Fortran we saw that arguments that are passed into subprograms are not copied, but instead, the address of the variables in memory is passed into the function. Thus, depending on the INTENT of a given variable, its value can be modified within the subprogram.

- This behaviour is called pass-by-reference, since in Fortran we are passing the references to the variable locations in memory.

- C however, is pass-by-value. Thus, when a variable is passed into a function in C, the function gets a local copy of the variable, and any changes made to its value would not be seen by the calling function.

Pass-by-value / Pass-by-reference

- So if C functions receive a copy of a variable, and any changes to that variable within the function are not seen by the calling function, then we must ask:
  - Can a C function modify a variable in the calling function?
  - And how can we "return" more than one value from a function?

- To do this we must explicitly pass *references* to the variable locations in memory, that is, we pass the *address* of the variables.

- For this we will need to use pointers (see later).

Example

- Here is a C program with a function:

```
#include <stdio.h>

void location_check(int x){
    int y = 10;
    printf("In location_check() y = %d and &y = %u\n", y, &y);
    printf("In location_check() x = %d and &x = %u\n", x, &x);
}

int main(int argc, char *argv[]){
    int y = 2, x = 5;
    printf("In main(), y = %d and &y = %u\n", y, &y);
    printf("In main(), x = %d and &x = %u\n", x, &x);
    location_check(y);
    return 0;
}
```

- And here is the output:

```
In main(), y = 2 and &y = 3221223212
In main(), x = 5 and &x = 3221223208
In location_check() y = 10 and &y = 3221223164
In location_check() x = 2 and &x = 3221223184
```