Overview of the C Programming Language

CS-272 Fall 2002
Structure of a C Program

General Structure

- C is very flexible in terms of structure of the program
- it is recommended to maintain a nice and structured organization of the program; this will make the program easier to read and to maintain
- Here following is a sample C program

// This is a simple program

// GLOBAL SECTION --------------------------------------------------------------------------

// libraries
#include <stdio.h>
#include "genlib.h"

// Constants
#define MAX 10

// Variables
int useless;

// prototypes
int square (int input);

// FUNCTIONS SECTION --------------------------------------------------------------------------

// main function
main()
{
    int count;
    printf(" N | N\n\n");
    for (count = 0; count < MAX; count++)
    {
        printf(" %d | %d \n", count, square(count) );
    }
}
int square (int input)
{
    int result;

    result = input * input;
    return (result);
}

Comments about the program:

• program contains comments; these are just used for documentation and are ignored by the compiler (removed by the preprocessor); in C comments can be either started with the symbols // or by enclosing the comment between /* and */

• observe that we have decided to structure the program into two parts. The first part, called Global section, does not contain actual statements (there is no ‘code’ to be executed appearing here), but it contains only general declarations to guide the compiler in the generation of the code. The second part, the Functions Section, contains a collection of functions, implementing the operations that are required by the program to solve the problem at hand. Some comments about this:

  o each function is simply a collection of instructions with a name attached to it. You can request execution of such collection of instructions by simply indicating the name of the function. You can think about writing functions as a process of ‘extending’ the programming language – you create new instructions that are needed to solve the problem at hand.

  o each C program has to contain at least one function. In addition each program has to contain a function called main which represents the point where the program execution is expected to start.

• most programs make use of functions that have been written by other people; for example, the above program makes use of the function called printf, used to display messages on the screen. This function has been developed by the same people that implemented the C language. All these ‘predefined’ functions are stored in libraries. A library, in simple words, is just a collection of functions that have been written by other people. Whenever you use functions that are stored in a library, you need to notify this fact to the compiler. The compiler needs to know which libraries should be searched to find the functions used by the program. This notification is represented by the include statements present in the Global section of the program. In the example, we are making use of two libraries.

  o observe that the include where the library name is enclosed between angular brackets represents access to a standard C library

  o the include statements where the name of the library is enclosed between double quotes represent access to a user-defined library
• the Global section contains declaration of constants, using the \#define statements

• the Global section contains prototypes: these are short descriptions of the functions that compose the program

• each function is composed by two parts:
  o the header, that contains the name of the function and the description of the inputs and output
  o the body: enclosed between braces, it contains description of the variables used by the function and a collection of instructions, implementing the behavior of the function.

### Variables and Basic Types

• each C program is used to manipulate data; data manipulated by a program have to be stored in memory; thus the language has to provide a mechanism to store and retrieve data from memory. In the most popular programming languages the method used to accomplish this task is called variables

• each variable is a piece of memory used to store one data item (container)

• each variable is described by a number of attributes:

  1. Name: used to identify the entity in the program (naming conventions)
  2. Type: used to define what kind of data will be stored in this variable and what sort of operations can be applied to such variable
  3. Value: current content of the area of memory associated to the variable
  4. Memory: a collection of memory locations reserved for the variable
  5. Lifetime: the period of time from the creation of the variable (moment in which memory is reserved for the variable) to its destruction (moment in which the memory associated to the variable is released and returned to the system)
      o static variables
      o dynamic variables
  6. Scope: the parts of the program where the variable can be used:
     o global variables
     o local variables

Some more information about variables:
• each variable has to be declared; the declaration defines most of the above mentioned attributes; it is used to specify name and type of the functions (explicitly); you may provide an initial value;

```plaintext
int x;
float y = 3.14;
```

The position where the declaration is provided is used to determine the scope of the variable; if the declaration is inside a function, then the variable is local to the function; if the declaration is outside (in the global section) then the scope is the whole program. In addition this will affect whether the lifetime is static or dynamic. You can also affect the lifetime using the `static` attribute in the variable declaration:

```plaintext
static int x;
```

In the previous example, we have one variable, called `useless` which is global and static. The variable `result` is dynamic and it is local to the function `square`. The variable `count` is also dynamic and it is local to the function `main`.

### Data Types

#### General concepts

• a data type is composed of two components
  1. a Domain: the set of values that can be legally assigned to variables of this type
  2. a set of operations that indicate how values can be manipulated

• we commonly classify the data types
  o atomic data types vs. complex data types
  o primitive data types vs. user-defined data types

#### Integer Types

• primitive and atomic data types

• the domain contains a subset of the integer numbers (i.e., all the integer numbers within a given range)

• C provides multiple integer data types: short, int, long, long long

• they differ in the amount of memory assigned and in the range of integer that belong to their domain

• implementation dependent
**Floating Point Types**

- used to store numbers that are not integer (e.g., 3.14159)
- primitive and atomic types
- different types: float, double, long double
- different memory (4, 8, 12); this implies greater precision (store a larger number of digits)
- precision = number of digits of the number that can be stored
- two notations: 2.1242 and 1.3E+8
- operations: +, -, *, / (real division)

**Characters**

- variables that can store a single character
- standard set of characters (ASCII); basic 128 characters, extended 256 (1 byte)
- one type: char
- internally stored as numbers (ASCII encoding); e.g., A is 65, a is 97, 0 is 48
- characters can be written either symbolically between single quotes ('a' '$' 'z') or using their ASCII code (97)
- internally only numbers are stored; char is a special type of integer type; same operations as int can be used. E.g.,

```c
char c1,c2;

c1 = 'a';
c2 = c1 - 32;
```
• special characters: they have no special symbols to represent them (non printable) but they have a special meaning. They are described by a symbol preceded by '\'. E.g., \n, \t, \a

**Boolean Data Type**

• not a standard C type; defined in the genlib library
• used to describe variables that can store only the logical values true and false
• name of type: bool
• constants: TRUE and FALSE
• operations: operations on logical values: and (&&), or (||), not (!)

**Input and Output**

• goal of input: read information from the keyboard and store the data read in variables
• need different functions to read data of different types (need to interpret the syntax of the input, need to create the appropriate internal representation)
• for the different types
  o char: getchar
    ```
    char c;
    c = getchar();
    ```
  o integer types (int and long): GetInteger (simpio library, user defined)
  o floating point types (float and double): GetReal
• to produce output: printf function
  o simple case: just a message: printf("Hello world");
  o new line characters: printf("Hello world \n");
  o create message by inserting values in specified point of the message (placeholders): printf("The value of x is %d and y is %d\n",x,y);
    o placeholder depends on the type of the value to print:: %d for int; %hd for short; %ld for long; %f for float and double; %c for character

Simple program:

```
// compute the average of three numbers acquired from the user
```
// -- GLOBAL Section --------------------------
#include <stdio.h>
#include "simpio.h"

// -- Functions -------------------------------
main()
{
    float n1,n2,n3;
    float average;

    printf("This program computes average of three numbers\n");
    printf("Insert first number: ");
    n1 = GetReal();
    printf("Insert second number: ");
    n2 = GetReal();
    printf("Insert third number: ");
    n3 = GetReal();
    average = (n1+n2+n3)/3;
    printf("The average of %f, %f, and %f is %f\n",n1,n2,n3,average);
}

Expressions

- sequence of operations applied to some operands
- problem of converting a two-dimensional expression into a linear form
- \((-b + sqrt(b*b-4*a*c)) / (2*a)\)
- problem: make sure that the operations are applied in the correct order
- parenthesis
- precedence: () , ! and - , */% , + - , ....
- associativity (mostly left associative)
- mixing types;
  - coercion order (safe and no loss of information)
    \(9 / 2.0 = 4.5\) (9 is converted from int to float as 9.0)
  - explicit conversion (type cast) (e.g., if I want \(a/b\) to be a real result)
    \(int a, b;
    float c;
    c = a / (float)b;\)
Assignment

- key operation; allows to change the value of a variable
- overwrites old value (only one value can be inside a variable)
- standard semantics: evaluate expression and store value
- simpler statement in a language
- implicit type coercion based on the type of the variable
- an assignment is also an expression: \( z = (x=6) + (y=7); \)
- = associate to the right; it is legal to write \( x = y = z = 6; \)
- dangerous: \( z = (x=6) + (x=7); \)
- simplified assignment statements: \( x++, ++x, x--, \) and \( -x \)
- no differences when used as statements; differences when used as expressions

Boolean Expressions

- ability to perform comparisons between values;
- \( == \) for equality and \( != \) for inequality
- numeric types can be compared for \( >, <, >=, \) and \( <= \)
- need to be careful: better to compare values of the same type (e.g., 5 may be different from 5.0)
- each produces a Boolean value; Boolean values can then be combined with the operations seen before (\( !, &&, || \)).
- short-circuit evaluation

Statements

- simple statements: assignments and function calls; terminated by semicolon
- basic control structures (sequencing, conditional, iteration)
- \{ .... \} block; example:
  ```
  main()
  {
      int one,two,three,result;
      printf("Insert 3 numbers:");
  ```
one = GetInteger();
two = GetInteger();
three = GetInteger();
result = (one+two+three)/3;
printf("The average of %d, %d, and %d is %d
",one,two,three,result);
}

Note that we have only one statement here (complex).

- if and if else
- example of program to determine if an input is even or odd

main()
{
    int input;

    printf("Insert number: ");

    input = GetInteger();
    if (input % 2 == 0)
        printf("%d is even\n", input);
    else
        printf("%d is odd\n", input);
}

- repetition: repeat a statement as long as a condition is satisfied

while (condition)
    statement

For example: simple program that reads a number and computes the sum of the digits of the number:
// Program that reads an integer and computes the sum of the digits of the integer

#include <stdio.h>
#include "simpio.h"

// constants
#define MINIMUM 0

// --- functions -------------------------------

main()
{
    int number, sum, digit;

    printf("Please insert a non-negative integer: ");
    number = GetInteger();
    sum = 0; // initially the sum of the digits is empty

    while (number != 0)
    {
        digit = number % 10; // extract the last digit
        number = number / 10; // remove the last digit
        sum = sum + digit;
    }

    printf("The sum of the digits gives %d\n", sum);
}
Another example: let us consider the problem of adding together a list of numbers that the user is typing on the keyboard. We do not know in advance how many numbers will be inserted, but the only thing that we know is that the user will type a special number (e.g., -1) to denote the end of the input.

// addition of a sequence of numbers; use of sentinel-driven loop

#include <stdio.h>
#include "simpio.h"

// ------ functions ------

main()
{
    int number, sum;
    sum = 0;

    while (TRUE)  // observe: this is an infinite loop; we don't have a stop condition
    {
        printf("Insert next number (-1 to stop): ");
        number = GetInteger();
        if (number != -1)
            sum = sum + number;
        else
            break;
    }

    printf(" The sum is %d\n",sum);
}

• if the number of iterations to be performed is known in advance (definite loop), then it may be appropriate to use a different type of loop – the for loop. The for loop is a counting loop and has the advantage of collecting in a single line all the information that pertain the execution of the loop (initial value of the counter, increment of the counter, exit condition).

    for ( initialization ; condition ; increment )
    statement

corresponds to

    initialization;
    while (condition)
    {
        statement
        increment;
    }

• simple to use: e.g., I want to add all the numbers from 1 to 100

    for (count=1; count <= 100; count=count+1)
sum = sum + count;

• or I want to add all the even numbers between 1 and 100:

    for (count =2; count<=100; count=count+2)
        sum = sum + count;

**Functions**

• collection of statements with a given name

• whenever the name is referred ("called"), the corresponding sequence of
  statements is executed

• function definition:

    header
    Body

• the Body is always a Block (compound statement)

• the header includes

    output-type function-name ( formal parameters )

    o output type: each function produces a result (a single value) at the end of
      its execution; the output type describes what type of value the function
      will produce as result

    o formal-parameters: description of the inputs that the function needs in
      order to perform its job. In particular, we need to know

        ▪ how many inputs are expected

        ▪ what is the type of each individual input

        ▪ what name are going to use to refer to a specific input within the
          function

    bool is_even (int input)
    {
        int result;

        result = input % 2;

        if (result == 0)
            return TRUE;
        else    return FALSE;
• note that the header of this function identifies a function called is_even; the function will eventually produce a value that is of type Boolean (either TRUE or FALSE); the function is expecting one input, of type integer. Each time I need to access the input value within the function, I will simply need to refer to the name of the formal parameter (input).

• Thus, the formal parameters allow you to get a hold of the values provided as inputs to the function.

• when the function terminates, it is necessary to explicitly specify what value has to be returned as result of the execution of the function. This step is accomplished through the use of the return statement; this immediately stops the execution of the function and copies its single argument as result.

• we can have functions that do not return any result; in this case the output-type should be set to void and the return statement should have no argument. For example, a function that prints a line of "*" of a given length:

```c
void print_starts (int length)
{
    int count;
    for (count = 0 ; count < length; count++)
        printf("*");
    return;
}
```

• a function is executed only when it is explicitly called. The function call is a simple statement, that contains the name of the function as well as its actual parameters. The actual parameters are the values that we intend to give to the function as inputs.

```c
result = is_even(25);
```

in this case I request execution of is_even, I provide one actual parameter (25) and I will collect in the variable result the value that the function will return at the end of its execution.

• actual parameters can be any expressions; their value will be computed before calling the function and the function will ONLY receive these values.

• Example:

```c
main()
{
    int x,y;
```
x = 0;

y = 6;

function(x,y);

printf("At the end x=%d and y=%d\n",x,y);

}     

void function (int input1, int input2)
{
    input1 = input1+1;
    input2 = input2 * 2;
    return;
}

The key point here is that input1, input2 are two new variables that are created when the function is called; these variables are initialized with the values obtained from the actual parameters (the values of the actual parameters are copied into these new variables). All the modifications to input1 and input2 are local to the function and do not affect anything outside of the function. This means that when the function completes, we will not see any change in the values of x and y. This process is what we call CALL BY VALUE because the function receives as input exclusively a copy of the values of the actual argument (NOT the actual arguments themselves --- e.g., the function in the example gets a copy of the value of x, it does not get x itself).
**USER DEFINED DATA TYPES**

### Simple data type

C allows the user to define new atomic data types. The only mechanism to accomplish this is through the notion of **enumeration**. In the case of enumeration a type is defined by simply describing its domain --- i.e., by simply listing what are the values that are legal for this data type.

- **typedef enum { north, east, west, south } Direction;**
- defines a new data type, called Direction;
- variables of type Direction can be assigned any of the four values indicated (north, east, south, west).
- This declaration is placed in the global part of the program

```c
#include <stdio.h>

// a type declaration
typedef enum {north, east, west, south} Direction;

// -- functions ----
main()
{
  Direction mydir;
  
  ....
  mydir = north;

  ....
```

- note that this is just a syntactic sugar; C replaces the type with int and defines constants (starting from zero) for the listed values
- you can affect the way the translation is performed:
  ```c
typedef enum { north=5, south, east, west } Direction;
  ```
  - in this case north will be translated to 5, south to 6, east to 7 and west to 8.
  - you can perform operations as for integers (increment, decrement, comparisons)

### Typedef

typedef can be used independently from the enumerations. It is a general mechanism that allows one to give a name to a type. The general format is

```c
typedef .... here you describe a type .... NameOfDataType;
```

For example, if we want to represent the age of a person, we could use unsigned short as a good type; but if we want to stress that this is to be used only to represent an age, I could add the declaration
typedef unsigned short Age;

Makes the code more readable (and easier to update).

Allows to distinguish “different” uses of the same type (e.g., I could also use unsigned short to represent a part number, but part numbers are different than ages…).

Arrays

• simple example: normalize the 5 grades given in a class

// read 5 grades and curve them w.r.t. the highest grade

#include <stdio.h>
#include “simpio.h”

// -- functions –

main()
{
  int gr1, gr2, gr3, gr4, gr5;
  int max = 0;
  printf("Insert grade: ");
  gr1 = GetInteger();
  if (max < gr1) max = gr1;
  printf("Insert grade: ");
  gr2 = GetInteger();
  if (max < gr2) max = gr2;
  printf("Insert grade: ");
  gr3 = GetInteger();
  if (max < gr3) max = gr3;
  printf("Insert grade: ");
  gr4 = GetInteger();
  if (max < gr4) max = gr4;
  printf("Insert grade: ");
  gr5 = GetInteger();
  if (max < gr5) max = gr5;
  gr1 = gr1 / max * 100;
  gr2 = gr2 / max * 100;
  gr3 = gr3 / max * 100;
  gr4 = gr4 / max * 100;
  gr5 = gr5 / max * 100;
}

• problem when the number of variables increases
• use of arrays
  o uniform collection of variables (all same type)
  o elements organized linearly in a sequence
  o collection has a single name
  o elements are individually accessible through their position in the sequence (start count from 0)
  o static
• array declarations:
  int array[10];
• at the declaration the size of the array has to be known
• initialize the content of an array at declaration
  int array[6] = {10, 20, 30, 40, 50, 60};
• if too many values are indicated, those in excess are ignored
• if too few values are indicated, the rest of the array is initialized to zero;
  Thus:
  int array[100] = {0};
  will set the whole array to zero in a single step.
• elements of the array are standard variables and can be used wherever standard variables are admissible
• each element or the array is recognized through its index;
  array[i] = array[j] * 2;
• typically you are interested in accessing many elements of the array; the ability to use variables as indices allows the use of loop to compact the processing; e.g., increment by one all the elements of the array
  for (i=0; i < size; i++)
    array[i]++;
• effective versus allocated size
• arrays as parameters (call by reference)