Tutorial on C Language Programming

Teodor Rus

rus@cs.uiowa.edu

The University of Iowa, Department of Computer Science

Tutorial on C programming

C program structure:

- Data structure
- Control structure
- Program structure

Data structures

- Predefined data types:
  - integer (int), small integers (short), large integers (long)
  - real numbers (float), large real numbers (double)
  - character data (char)
- User defined data types using type constructors *array, record, pointer, file*

Declarations

A data object of a defined type T is declared using the construct of the form *T data* where T is a type expression and data is the data object name

Example:

- int x declares x an object of type integer
- short x declares x an object of type small integer
- long x declares x an object of type large integer
- float x declares x an object of type real
- double x declares x an object of type large real
- char x declares x an object of type character

Definitions

- An object of a user defined type T is constructed using one of the type constructors *struct, [], *, FILE* that takes as arguments objects of already defined types.
- A new user defined type T is constructed using the meta-constructor *typedef* and a type or a type constructor

Record type definition

- A record type is defined using the *struct* constructor following the template:

```
struct TypeName
      {
       component1;
       component2;
       component3;
      }
```

- Components are object declarations of the form *T ObjName;*

Note: TypeName is an abstraction

Record object declaration

- An object of type TypeName is obtained by the declaration

  ```
  TypeName MyRecord
  ```

- One can put together the definition and the declaration getting:

  ```
  struct TypeName
        {
         component1;
         component2;
         component3;
        } MyRecord;
  ```

Example record

- Example of a record type definition and declaration is:

```
struct Data
      {
       int Day;
       int Month;
       int Year;
      } MyData, *MyPT, MyArray[Max];
```

Note: type expressions are obtained by combining the type constructors struct, *, [], in a well defined manner

Reference to record components

- MyData.Year, MyData.Month, MyData.Day are references at the components of the data object MyData
- $MyPT->Year$, $MyPT->Month$, $MyPT->Day$ are pointer reference to the same components.
- Note, we need to use MyPT = &MyData before this reference make sense; i.e., $MyPT->Year \equiv (*MyPT).Year$.

Memory representation of records

Consider the following definition and declarations:

```
struct example
      {
       int x;
       int *y;
      } Obj, *PtObj;
```

Memory representation

Memory representation of Obj is in Figure 1

integer
integer address

Figure 1: Record memory representation

Memory representation of PtObj

This is shown in Figure 2

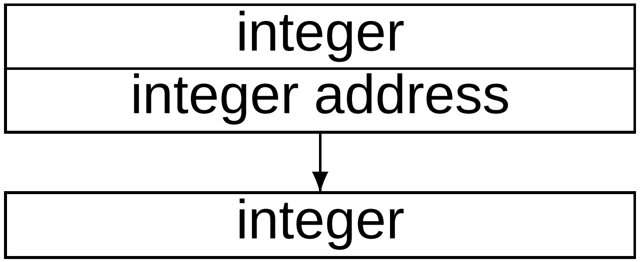

Figure 2: Pointer to record memory representation

Facts about records

To give few important facts about records, assume that PtObj = & Obj has been executed. Then we have:

- Obj.x is the integer x; $PtObj->x$ is the integer x
- Obj.y is the (integer) address y; $Obj->y$ is the address y;
- $++PtObj->x$ increments x not PtObj; $(++Pt)->x$ increments PtObj before accessing x; $(PtObj++)->x$ increments PtObj after accessing x
- $*PtObj->y$ fetches whatever y points to (an integer); $*PtObj->y++$ increments y after accessing whatever it points to (this is an address operation); $(*PtObj->y)++$ increments whatever y points to (this is an integer operation);

Array data type

- A unidimensional array of n objects of type T is defined by

  `T UniName[n]`

  Note, this is both a definition and a declaration
- A bidimensional array of $m \times n$ objects of type T is defined by

  `T  BidimName[m][n]`
- The element i of the array UniName is referenced by ArrayName[i]. Note, $0 <= i < n$

Examples: int x[20], struct example MyArray[100][100]

Array memory representation

- The indices of the elements of an unidimensional array of size n are 0, 1, . . ., n-1
- The elements of a bidimensional array BidimName[m][n] are stored in memory on a row-major, i.e., they are:

```
BidimName[0][0], BidimName[0][1], ... BidimName[0][n-1]
BidimName[1][0], BidimName[1][1], ... BidimName[1][n-1]
BidimName[2][0], BidimName[2][1], ... BidimName[2][n-1]
...
BidimName[m-1][0], BidimName[m-1][1], ... BidimName[m-1][n-1]
```

Union data type

- Unions are records with variable fields like in Pascal
- **Example:**

```
union UniName
     {
      int ival;
      float fval;
      char  *pval;
     } uval, *p;
```

The variable uval may have as value an integer, a real, or a pointer to a character.

- Only one of the components is the value hold by the uval

Reference to union components

- The elements of a union are referenced in the same way as elements of a record (struct) are referenced
- The memory representation of variable uval will be large enough to accommodate any of the values that are used in its definition
- It is the programmer’s task to provide a discriminant that will show what component of a union is in the variable uval at a given time.

Example of a union usage

The symbol table entry of a symbol table used by a compiler:

```
struct SymTabEntry
      {
       char *name;
       int  flags;
       int  stype;
       union
           {
            int   ival;
            float fval;
            char  *pval;
           }sval;
       } SymTab[MaxSymb], *PtSymTab[MaxSymb];
```

Reference to union components

SymTab[i].Object and $PtSymTab[i]->Object$, where $Object \in \{*name, flags, stype, sval\}$

are references to symbol table element components.

Pointer data type

- Every object has an address (name) and a value
- An object of type pointer has as its value the address of an object of a given type
- An object of type pointer is defined by the construct

  `T *PtName;`

  where * show that PtNamed is a pointer and T shows the type of object address it may hold

Example pointers

- int x, z; /* x and z are variables of type integer */
  int *y, *w; /* y and w are variables of type pointer to integer */
  char v, *p; /* p is a variable of type pointer to character */
- Address of an object x of type T is obtained by the operator &, i.e., is &x
- y = &x is a valid assignment while y = x is not

Pointer references

direct by name, indirect by *name

- The name of a variable of type pointer references the address of the object it holds. Hence, w = y is valid but w = p is invalid
- Dereferencing of a variable of type pointer leads us to the value hold in the object whose address is hold by the pointer. Hence, (*y) is the integer whose address is in y
- Operation on a variable of type pointer (such as y) are address type operations
- Operations on the value of the objects whose addresses are hold by pointers (such as (*y)) are data type operations

File data type

- A file is a potentially infinite stream of objects (characters, integers, reals, strings, arrays, etc)
- A file is described by descriptor that shows:
  - type of the objects it contains
  - order relation among its components
  - access method used to file components
- In C-language a file is specified by a name and a file-descriptor
  - File name is user defined
  - File descriptor is obtained from the system using the declaration FILE *fp;

Operations with file

The main operations on a file area: *open, doio, close*

- File open links the file abstraction defined in the program with the physical media where the file objects are stored. In C this is done by

  `fp = fopen(name,mode), where mode is "w", "r" or "rw"`

- File close removes the links established by open.
- I/O operations: printf, fprintf store objects in the file, and scanf and fscanf access objects in a file
- printf, fprintf, scanf, fscanf have a formate that can be learn by inspecting the man page of these functions

User defined types

- Programmers may define their own types using *typedef* construct
- The usage pattern is

  ```
  typedef TypeDefinition TypeName
  ```

  where TypeDefinition is the type expression defining the new type and TypeName is the name of the new type
- Objects of type TypeName are then declared as usual
- TypeName can also be used as component of various type expressions using constructors struct, [], *, and FILE.

Examples

- typedef int LENGTH; /* LENGTH is a new type */
  LENGTH len, maxlen, *L[]; /* variable of type LENGTH */
- typedef char *string; /* string is synonymous to char * */ string p, lineptr[L]; /* These are variable of type string */
- ```
  typedef struct node
               {
                char  *value;
                int    count;
                struct node *Left;
                struct node *Right;
               } TreeRoot, *TreePTR;
  ```
  TreeRoot a; /* a is a variable of type TreeRoot */
  TreePTR b; /* b is a variable of type TreeRoot * */
  ```

Control Flow Structures

C language computation units

- Assignment statements
- Block statements: {statement1; ... ;statement}
- Control statements: branching and looping statements
- Function calls;

Assignment statement

- Syntax: `identifier = expression;`
- Semantics: evaluate `expression` to val and then assign val as the value of `identifier`

Note:

- Type of val should be the same as the type of `identifier`
- Peculiarities: `id++` is equivalent to `id = id + 1` and `id- -` is equivalent to `id = id - 1`
- C expressions are arithmetic or logic; but assignment statements are also expressions.

Branching statements

- if-statements
- if-else statement
- switch-statement
- break-statement
- continue-statement
- unconditional jump statement

If-statement

- Syntax: `if (expr) statement`; where `expr` is boolean
- Semantic: evaluate expression `expr` to val; if val is true execute `statement`, otherwise execute next statement of the program

If-else statement

- Syntax: `if (expr) statement1; else statement2;`
- Semantics: evaluate expression `expr` to val; if val is true execute `statement1` otherwise execute `statement2`; in any case control flows to the next statement of the program

Switch statement

- Syntax:

```
switch (expr) /* expr is a boolean expressio
      {
        case C1: {statement0;break}
        case C2: {statement1;break}
        ...
        default: {DefaultStatement;break}
       }
```

- Semantic: evaluate `expr` to val; if val is equal to one of the case constants C1, C2, . . ., the associated statement is executed; otherwise DefaultStatement is executed. Note, default clause is optional; if not there and val is not equal with any case constant, no action take place

Break statement

- Syntax: `break;`
- Semantic: terminates the execution of a loop or a switch

Continue statement

- Syntax: `continue;`
- Semantic: terminates the current iteration of a loop

Unconditional jump statement

- Syntax: `goto Label;` where `Label:Statement;` belongs to the program
- Semantic: forces control to go to the `Statement;`

Looping statements

- while-statement
- do-while statement
- for-statement

While statement

- Syntax: `while (expr) Statement;` where `expr` is boolean
- Semantic: evaluate `expr` to val; if val is true `Statement` is execute and while statement is repeated; if val is false control flows to the next instruction of the program

Note: true boolean values are any integer different from zero;

false boolean value is the integer zero.

Do-while statement

- Syntax: `do Statement; while (expr);`
- Semantic: equivalent to

```
Statement;
while (Expr) Statement;
```

Note: while statement executes zero or more iterations of the loop; do-while statement executes one or more iterations of the loop.

For statement

- Syntax: `for(expr1; expr2; expr3) Statement;`
- Semantic: equivalent to

```
expr1;
while (expr2)
     {
      Statement;
      expr3;
     }
```

Note: any of the expressions expr1, expr2, expr3 may be omitted; if expr3 is omitted it is interpreted as true, hence various sorts of infinite loops can be performed

Block statement

- Syntax:

```
{
Declaration list;
Statement list;
}
Declaration list:
      Declaration;
      Declaration list Declaration;
Statement list:
      Statement;
      Statement list Statement;
```

- Semantics: statements in Statement list are executed in sequence in the environment provided by Declaration list

Function definition

- Syntax:

```
type name (formal parameter list)
         {
          Declaration list;
          Statement list;
          return result
         }
```

- Semantic: a function definition specifies the computation defined by the Statement list in the environment defined by formal parameter list and Declaration list and return a result of type `type`

Example

```
/* power: raises the value of variable base to /*
/* the power values of variable n, n >= 0 */

int power (int base, int n)
         {
          int i, p;
          p = 1;
          for (i = 1; i <= n; i++)
             p = p * base;
          return p;
         }
```

Note: comments in C are enclosed in /* ... */ Use comments outside of function defi nition; formate function body such that the text indentation allows reader to understand it.

Function declaration

- Syntax: `type name (type1, type2, ...)` where `type` is the function type (i.e., the type of result returned by the function) and `type1, type2, ...` are the types of the formal parameters
- Semantics: declare `name` as the name of a function whose arguments are of types `type1, type2, ...` and whose result if of type `type`

Note: since a function declaration is a declaration it must be provided in the declaration list of the statement that uses it.

Function call

- Syntax: `identifier = name (actual parameters);`
  - `identifier` must have the same type as the type specified in the definition and the declaration of `name`
  - Actual parameters must expressions whose values are of the types that `type1, type2, ...` specified in the definition and the declaration of `name`
- Semantic: execute computation encapsulated in the definition of function `name()` in the environment provided by actual parameters and return the result.

Example: `int x; int power(int, int);...; x = power(2,3); ...`

Parameter passing

Actual parameters are passed by value, except arrays, which are passed by reference.

Remember

- Arrays are transmitted by reference, i.e., the address of the array variable is transmitted.
- To operate on the local elements of a function using them as parameters to another function pointers need to be transmitted at function call
- Initialization of the pointers is required upon function call.
- Note that pointers are typed i.d., int *x, char *x, struct name *x are different pointers.

Function memory representation

A function is represented in memory by two components:

- Execution code, i.e., memory image of executable statements
- Activation record

Activation record

Activation record is a data structure constgructed by the compiler and contains:

- Function return value;
- Static link: a pointer to the activation record of the function that contains the defi nition of the function. In C this is nill.
- Dynamic link: a pointer to the activation record of the function that contains the call of the function
- Stack extension value
- Return address
- List of locations for actual parameters
- Local variables of the function

Structure of a function in memory

- Figure 3 shows the structure of a function in memory:

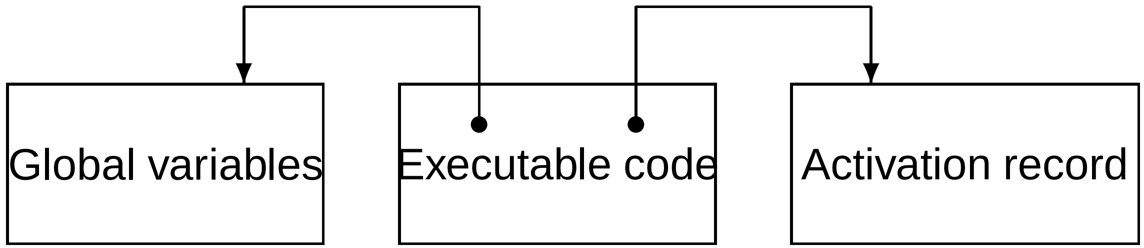

Figure 3: Function memory representation

Structure of a C language program

A C program is composed of four componentgs:

Macro defi nitions (optional)

Global declarations (optional)

Main function (mandatory)

Other functions components of the program (optional)

Note

- A C program has four components: macro definitions, global declarations, main() function, and other functions. Only main() function is mandatory.
- A C language program is executed by the operating system by *calling its functions main()*.
- A C language program is implicitly declared to the system by the presence of the unique names, `main()`

Macro definition component

- Syntax: sequence of macro-operations of the form:

```
#define name value
#include "filename"
#include <filename>
```

- Semantics:
  - `#define name value` allows programmer to use name in the program while compiler replaces it with `value` which can be any string of characters.
  - `#include "filename"` allows the programmer to develop a program on various separate files.
  - `#include <filename>` allows the programmer to make use of files contained in various libraries of the system

Global declarations

- Syntax: declarations of variables that occur outside of the function components of the program.
- Semantic: all global variables are accessible to all function components of the program

Main function of the program

- Syntax:

```
main (int argc, char *argv[])
    {
     Declaration list;
     Statement list;
    }
```

Note: since a function may have no arguments main() { Body } is also valid.

Program execution

- A program is executed by the system calling the function main() as consequence of a command given by the programmer. This command has the form %name arg1 arg2,...
- `argc` is an integer variable where the number of the arguments used in the execution command is stored
- `argv[]` is an array of pointers to strings where the arguments arg1, arg2, ... of the execution command are stored.

Other function components

- Syntax: any function definition
- Semantic: function components of a program may be called by the main() or among themselves. However, in order for main() or any other function to call a function f() the following must be done:
  - f() must have a definition accessible to main() and to other functions that intend to call it
  - f() must be declared in main() and in the functions which intend to call it

Example program

```
#include <stndio.h>
main ()
    {
     int C;
     C = gethchar();
     while (C != EOF)
          {
           putchar(C);
           C = getchar();
          }
     }
```

This program copies the standard input to the standard output

Bonus point assignment

Rewrite th program such that it will copy a file f1 into another file f2; files f1 and f2 should be given in the command line.

Program memory representation

C compiler maps a C program into three segments called *data*, *text* and *stack* as seen in Figure 4

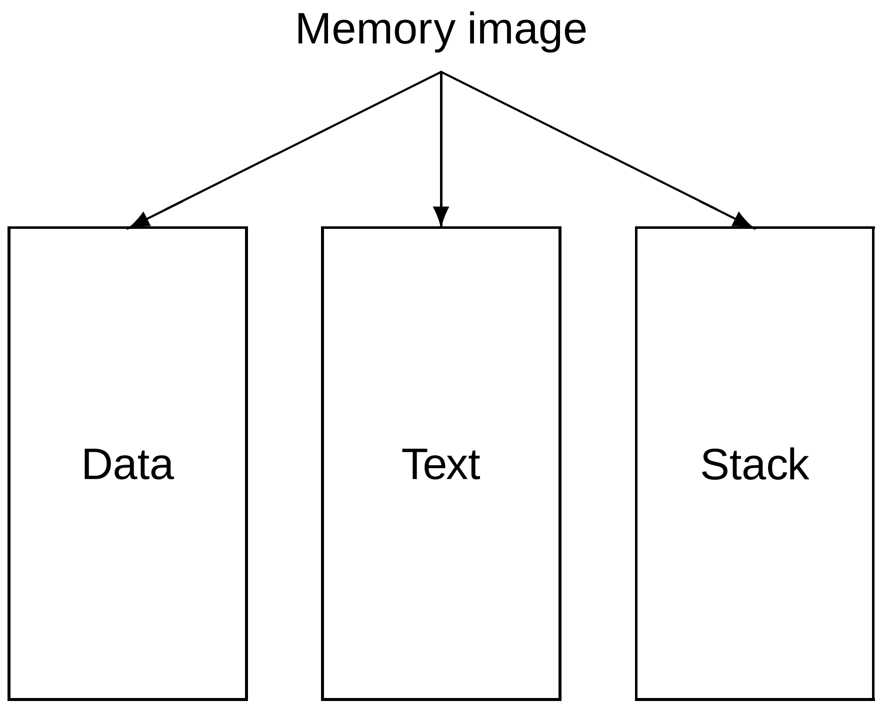

Figure 4: Memory image of a C program

Data segment

- Contains all global data of the program
- Data segment is constructed by the compiler

Text segment

- Contains all executable code of the program
- Each function component of the Text segment has access to the global data in the Data segment and to the local data in the activation record of that function.

Stack segment

- Stack segment is dynamically generated by program execution
- When a function is called its activation record is pushed on the stack segment
- When a function return its activation record is popped out from the stack segment

Development of a C program

- Use an editor to generate the file that contains the program. Example, execute %vi mylms.c
- Compile the C program in the file mylms.c using the command % cc [Options] mylms.c
- If mylms.c contains a C program syntactically correct the result of the compilation is an executable file called a.out.
- If you want to give the name mylms (or any other name) to the executable use option -o mylms in cc command
- Test the program on the test data; use dbx to help this
- Read the documentation for vi, cc, dbx using manual page