Introduction to C language for CS2100

WXY

September 1, 2024

Contents

1 von Neumann Architecture (Prinstone Architecture)

Von Neumann Architecture describes a computer consisting of:

1. Central Proessing Unit (CPU)
    - Register: special, fast but expensive memory within processor. Used to store temproray result of computation. MIPS processor has 32 registers and each register is of 32 bit wide.
    - A control unit containing an instruction register and prgroam counter.

    - An arithmetic/logic unit (ALU)

2. Memory
    - Store both program and data in random-access memory (RAM)
3. I/O Devices

2 Variables in C

A variable is identified by a name (identifier), has a data type and contains a value which could be modified. It also has an physical address in the memory. A variable is declare with a data type. Such as int, float etc. Variables can be initialised at declaration, otherwise would hold an unkonw value (which cannot be assumed to be 0).

Variables in C MUST be declared BEFORE usage. Initialisation may be redundant at times. When initialsation is required but not done so warning will be given by the compiler.

Declaration vs Definition

- Separate compilation:
- *extern* keyword

Separate compilation and definition

if variable is defined in both file1.c and file2.c, which will be separately compiled to file1.o and file2.o if definition duplication cannot be resolved by checking the scopes, the linker would not be able to join the compiled files and no executable can be generated.

3 Data types in C

C is a stringly typed language, which means data must has data type specified. Type casting in C is allowed, for example:

```
float i = 1.5;
int j = i;
printf("\%d", j);
```

j will be converted to integer data type by truncating the decimal part.

4 Program Structure in C

A C program has 4 main parts:

- Preprocessor directives: libraries used, #define etc.

- Input: through stdin (using scanf), or file input.
- Compute: through arithmetic operations and assignment statements.
- Output: through stdout (using printf), or file output.

Compile C program files using command

```
gcc programFileExample.c -o programFileExample
cc programFileExample.c -E -o programFileExample.pp
// to stop at preprocessor directory
cc programFileExample.c -S -o programFileExample.s
// compile into assembly
```

programFileExample will be compiled to binary code in file programFileExample. Without specifying the output file, the code will be compiled by default into a.out.

4.1 Preprocessor Directives

C Preprocessor consists of:

1. Inclusion of header files: libraries used;
2. Macro expansions: defining constants;
3. Conditional compilation

4.1.1 Header Files

To use standard input and output functions such as scanf and printf, <stdio.h> must be included.

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

To use functions in a library, respective header file must be included.

```
#include <math.h>     // Library for mathematical functions
    // must be compiled with -lm to compile
#include <string.h>   // Library for string functions
# include <pthreads.h>
    // must be compiled with -pthreads
```

4.1.2 Marco Expansion

Compiler will do text substitution to replace all marco names in the program with the value declared.

```
#define PI 3.142      // use all CAP for marco
// NOTE: No unintentional semicolon in defining constant, otherwise
```

```
#define PI 3.142;
// PI will be substituted by '3.142;' instead.
```

4.2 Input/Output

scanf(format string, input list); printf(format string, print list);

```
int i;
double j;              // declare before use
printf("Enter a number: ");
scanf("%d", &i);       // &: address of operator
                       // the following input will be stored at the
                       // address of i
printf("Enter a decimal number: ");
scanf("%lf", &j);
printf("You entered %d and %f", i, j);
                       // listing order of variables corresponds to
                       // output order

// Note: scanf uses %lf to take floating point input,
//       while printf uses %f to print floating point output

%5d    // an integer with width of 5, right justified
%8.3f  // an float/double with width of 8, with 3 decimal places, right
   justified
```

Placeholder	Variable Type	Function Use
%c	char	printf/scanf
%d	int	printf/scanf
%f	float/double	printf
%f	float	scanf
%lf	double	scanf
%e	float/double	printf(for scientific notation)

Table 1: Format specifiers

Escape sequences	Meaning	Result
\n	New line	subsequent output will be on the next line
\t	Horizontal tab	Move to the next tab position within the same line
\"	Double quote	literal double quote character
##	Percent	literal charater '%'

Table 2: Escape sequences

5 Pointers

5.1 Pointers

Pointers allow for the direct manipulation of memory contents, as well as bitwise operations by bit manipulation operators. A variable contains a name, data type and an address, which is a number that indicate the physical location of the variable in the memory. The address of a variable can be accessed by '&';

```
int a = 123;
printf("a = %d\n", a);    // a = 123
printf("&a = %p\n", &a); // &a = ffbff7dc
```

5.2 Declare a Pointer

Syntax:

```
//type *pointer_name
int *a_ptr;
```

5.3 Assign Value to a Pointer

Since a pointer contains an address, only an address may be assigned to pointer, which is a hexadecimal number indicating the location of a variable.

```
int a  = 123;      // declare a variable
int *a_ptr;        // declare a pointer points to int
a_ptr = &a;        // assign the address of a to the pointer declared in line
    above
```

5.4 Accessing Variable through Pointer

Using dereference operator '*'

```
int a = 123;
printf("a = %d\n", *a_ptr);  // 123
printf("a = %d\n", a);       // 123
// equivalent
*a_ptr = 456;                // rewrite variable ptr points To
printf("a = %d\n", a);       // 456, value modified through referencing from
    the pointer
```

$$*a_ptr \cong a$$

Example:

```
int i = 10, j = 20;
int *p;
p = &i;
printf("i = %d\n", *p);  // i = 10
*p = *p + 2;
printf("i = %d\n", *p);  // i = 12

p = &j;
printf("j = %d\n", *p);  // j = 20
*p = i;
printf("j = %d\n", *p);  // j = 12
```

Example:

```
#include<stdio.h>

int main() {
    double a, *b;          // a is a double; *b is a pointer to a double
        variable
    b = &a;
    *b = 12.34;
    printf("a = %f\n", a); // a = 12.34
    printf("a = %f\n", *b); // a = 12.34

    printf("a = %f\n", b); // Compilation warning, b is a pointer
    printf("a = %f\n", a); // Compilation error,
                           //cannot dereference from a double (segmentation
                               fault)
}
```

5.5 Tracing Pointers

```
int a = 8, b = 15, c = 23;
int *p1, *p2, *p3;
p1 = &b;      // make p1 points to b, which is 15;
p2 = &c;      // make p2 points to c, which is 23;
p3 = p2;      // make p3 points to the same thing as p2, which is c;
printf("1: %d %d %d\n", *p1, *p2, *p3); // 1: 15 23 23;

*p1 *= a;     // multiply the variable pointed by p1 by a; b => b * a = 90;
while (*p2 > 0) {     // p2 points to c,
                      // *p2 is dereferenced to be the value of c: 23;
    *p2 -= a;         // in each loop iteration, decrement the c by a;
    (*p1)++;          // and increment the value that p1 points to by 1;
}
```

```
printf("2: %d %d %d\n", *p1, *p2, *p3); // 2: 123 -1 -1
printf("3: %d %d %d\n", a, b, c);       // 3: 8 123 -1
```

5.6 Incrementing a Pointer

The value of pointers can also be directly manipulated to locate neigbhouring memory locations.

```
int a; float b; char c; double d;
int *ap; float *bp; char *cp; double *dp;
// int takes 4 bytes
// float takes 4 bytes
// char takes 1 byte;
// double takes 8 bytes;
ap = &a; bp = &b; cp = &c; dp = &d;
printf("%p %p %p %p\n", ap, bp, cp, dp);
// ffbff0a4 ffbff0a0 ffbff09f ffbff090
ap++; bp++; cp++; dp++;
printf("%p %p %p %p\n", ap, bp, cp, dp);
// ffbff0a8 ffbff0a4 ffbff0a0 ffbff098 each pointer increment by size of
   respective type.
ap += 3;
printf("%p\n", ap);
// ffbff0b4   incremented by 12 bytes which is the size of 3 int
```

5.7 Pointer to Pointer

When declaring a pointer to a type, e.g. int *p, the pointer will also be stored in a physical address. Of which another pointer can point to, e.g. int **q = &p;

```
int main(){
int *p; // Pointer to an int
int **q; // Pointer to a pointer to an int.
int x=5, y = 6;
printf("%p, %p, %d, %d\n", p, q, x, y);
printf("%p, %p, %d, %d\n", &x, &y, x, y);

p = &x;
q = &p;
p++;
(*p)--;
(**q)++;

printf("%p, %p, %d, %d\n", p, q, x, y);
return 0;
}
```