Computer Programming & Problem Solving

CS100

Mrs Sanga G. Chaki
Department of Computer Science and Engineering
National Institute of Technology, Goa
January, 2023

Last Class - Topics

1. Typecasting
2. Typedef
3. Binary Numbering system
4. Hexadecimal numbering system
5. Relation between binary and hex
6. Operation on bits – Bitwise Operators
7. Bit fields
8. Build process
9. File processing – argc, argv
10. Enum
11. Volatile qualifier
12. Variadic functions

TypeCasting

Typecasting

1. Used when we need to force the compiler to explicitly convert the value of an expression to a certain datatype.
2. You have already used this – where?
   a) Dynamic memory allocation

Typecasting

main.c

```c
#include <stdio.h>
int main(){
    float num = 9.85;
    printf("All weird things possible with programming");
    printf("\nValue of num = %f",num);
    printf("\nValue of num = %0.3f",num);
    printf("\nValue of num = %0.2f",num);
    printf("\nValue of num = %0.1f",num);
    printf("\nValue of num = %20f",num);
    printf("\nValue of num = %20.2f",num);
    printf("\nNow lets see typecasting");
    printf("\nValue of num = %d",num);
    printf("\nValue of num = %d",(int) num);
    return 0;
}
```

Output

```
/tmp/H8FZpukQ1f.o
All weird things possible with programming
Value of num = 9.850000
Value of num = 9.850
Value of num = 9.85
Value of num = 9.9
Value of num =             9.850000
Value of num =                 9.85
Now lets see typecasting
Value of num = 1923940557
Value of num = 9
```

Typedef

1. The typedef is a keyword that is used in C programming to provide existing data types with a new name.
2. You have already used this – where?
   a) Structures, unions

Binary Number System

Binary Numbering Systems

1. Numbering system - system of writing to express numbers
2. Base of a numbering system - The base of a number system refers to the total number of digits that are actually used in the given number system.
3. The number system that has the base 'b' consists of its digits in the [0, b-1] range.
4. For decimal NS, base = 10
5. For binary, base = 2
6. For Hex, base = 16
7. For octal, base = 8

Binary Numbering Systems

1. Binary numbers are represented by only two symbols or digits, i.e. 0 (zero) and 1(one).
2. A single binary digit is called a “Bit”. A binary number consists of several bits.
3. MSB, LSB
4. Conversion from Decimal to Binary
5. Conversion from Binary to Decimal
6. Eg: 9 = 1001

Hexadecimal Number System

Hexadecimal Numbering Systems

1. Numbering system
2. Base of a numbering system
3. For decimal NS, base = 10
4. For binary, base = 2
5. For Hex, base = 16
6. In hex notation, the ten digits 0 through 9 are used to represent the values zero through nine, and the remaining six values, ten through fifteen, are represented by symbols A to F.

Hexadecimal Numbering Systems

1. Example:
   a) 327 (decimal) = 3*100 + 2*10 + 7*1
   b) 327 (hex) = 3*256 + 2*16 + 7*1
2. Machines use binary – easier to represent using Hex

Hexadecimal Numbering Systems

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

1. So, binary 1010 0110 = hex A6
2. 1100 0101 0011 1010 binary is C53A

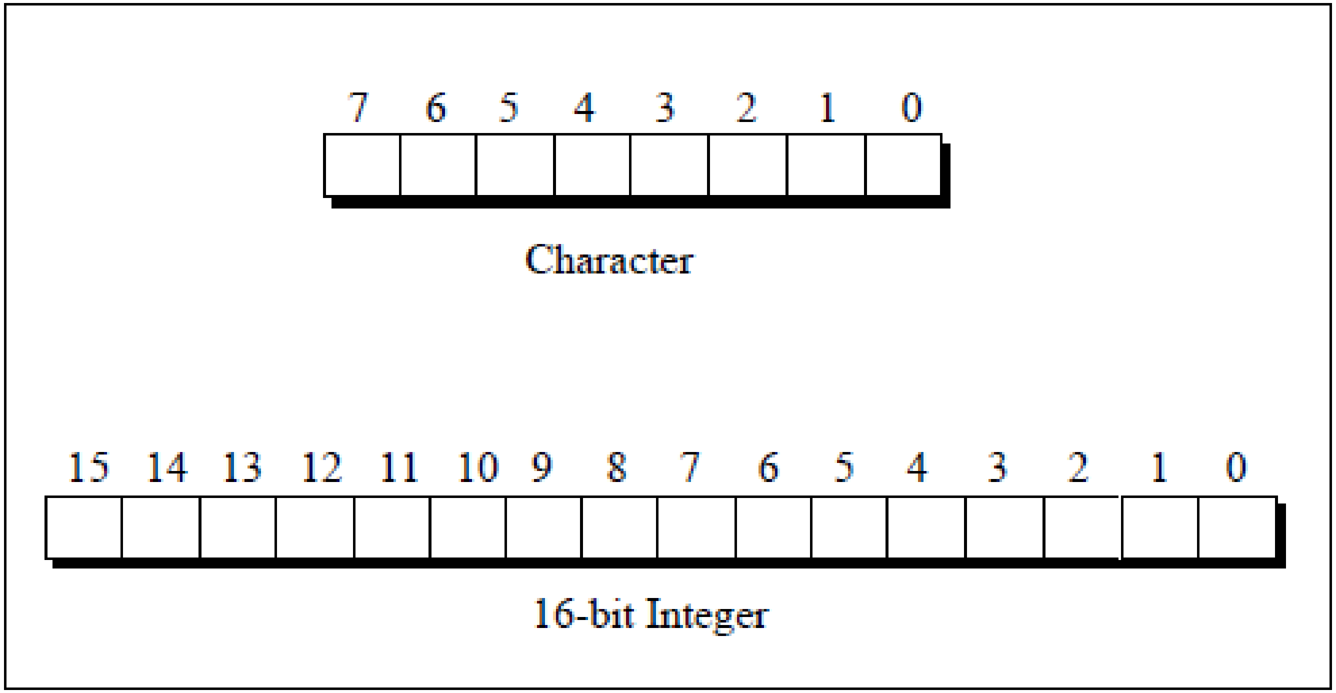
About Bits

Bit Operations – Bit Operators

1. **Byte manipulation – what we have done till now**
2. **Bit manipulation – interact directly with the hardware**
3. **Works on ints and chars – not on float/doubles.**

Operator	Meaning
~	One's complement
>>	Right shift
<<	Left shift
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR(Exclusive OR)

Bit Numbering



Bit Operation – One’s Complement

1. On taking one’s complement of a number,
   a) all 1’s present in the number are changed to 0’s and
   b) all 0’s are changed to 1’s.
2. Number = Binary equivalent of a number.
3. Example:
   1. if input = 1010
   2. One’s complement is 0101

```
k = ~j ;
```

Bit Operation – Right Shift Operator

1. It needs two operands.
2. It shifts each bit in its left operand to the right
3. Example:
   a) Let variable *var* contain the bit pattern 11010110
   b) var >> 2 would give 00110101
4. As the bits are shifted to the right, blanks are created on the left. These blanks must be filled somehow.
5. They are always filled with zeros

Bit Operation – Right Shift Operator

1. **Example:**

   64 >> 1 gives 32
   64 >> 2 gives 16
   128 >> 2 gives 32

   27 >> 1 is 13
   49 >> 1 is 24 .

2. **In a >> b if b is negative the result is unpredictable**

3. **If a is negative than its left most bit (sign bit) would be 1.**

Bit Operation – Left Shift Operator

1. Just like right shift.
2. Bits are shifted to the left
3. For each bit shifted, a 0 is added to the right

Bit Operation – AND

1. This operator is represented as &.
2. Works on bits – like AND Gate
3. Bits of both operands compared to get output
4. Both the operands must be of the same type (either char or int)

First bit	Second bit	First bit & Second bit
0	0	0
0	1	0
1	0	0
1	1	1

Bit Operation – AND - Example

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	0

This operand when ANDed bitwise

7	6	5	4	3	2	1	0
1	1	0	0	0	0	1	1

With this operand yields

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0

this result

Bit Operation – OR

1. Represented as |
2. Functions similar to AND
3. Truth table:

<table>
<tr><th>|</th><th>0</th><th>1</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

Bit Operation – XOR

1. Represented as ^
2. Truth Table:

^	0	1
0	0	1
1	1	0

Bit Operation – Example

```c
#include <stdio.h>
int main() {
	int a = 3, b = 2, c = 64;
	printf("One's complement of a and b = %d and %d\n",~a, ~b);
	printf("Right shift of a and c by 2 = %d and %d\n",a>>2, c>>2);
	printf("Left shift of a and c by 2 = %d and %d\n",a<<2, c<<2);
	printf("a AND b = %d\n",a&b);
	printf("a AND c = %d\n",a&c);
	printf("a OR b = %d\n",a|b);
	printf("a OR c = %d\n",a|c);
	printf("a XOR b = %d\n",a^b);
	printf("b XOR c = %d\n",b^c);
	return 0;
}
```

Bit Operation – Example

```
/tmp/RdI6De9GXL.o
One's complement of a and b = -4 and -3
Right shift of a and c by 2 = 0 and 16
Left shift of a and c by 2 = 12 and 256
a AND b = 2
a AND c = 0
a OR b = 3
a OR c = 67
a XOR b = 1
b XOR c = 66
```

Bit Fields

Bit Fields

1. In C, you can state the size of your structure (struct) or union members in the form of bits.
2. This concept is to because of efficiently utilizing the memory when you know that the amount of a field or collection of fields is not going to exceed a specific limit or is in-between a range.
3. Eg: a number of variables which can only be TRUE/FALSE are clustered together in structure form. Each needs only 1 bit to be represented.
4. We can implicitly characterize the width of a variable that tells the compiler to use only that specific number of bits.

Bit Fields

main.c

Run

```c
#include <stdio.h>
struct {    /* define simple structure */
    unsigned int a1;
    unsigned int b1;
    unsigned int a2;
    unsigned int b2;
    unsigned int a3;
    unsigned int b3;
} status1;
struct {    /* define a structure with bit fields */
    unsigned int a1 : 1;
    unsigned int b1 : 1;
    unsigned int a2 : 1;
    unsigned int b2 : 1;
    unsigned int a3 : 1;
    unsigned int b3 : 1;
} status2;
int main( ) {
    printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
    printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
    return 0;}
```

Bit Fields

Output

```
/tmp/HpNek1udJE.o
Memory size occupied by status1 : 24
Memory size occupied by status2 : 4
```

Bit Fields

1. Eg: Used when we need to work with several variables whose maximum values are very small.
2. So small that each variable can be stored using a few bits only – they do not require entire bytes/4bytes.
3. Example: We want to store data about students
   a) Gender: M/F/Non Binary/Don’t want to specify: 2bits
   b) Club: Can be member of 1 of 8 clubs: 3 bits
   c) Sports: Can participate in 1 of 6 sports: 3 bits
   d) Department: Can belong to one of 12 depts: 4 bits
   e) Total = 12 bits = can be stored in a single integer (if we consider 16 bit compiler)

Bit Fields

1. **But there are no bit level datatypes in C – so we use bit-fields.**

main.c

```c
#include <stdio.h>
#define MALE 0;
#define FEMALE 1;
#define NB 2;
#define DWS 3;
int main(){
    struct stud{
        unsigned gender:2;
        unsigned club: 3;
        unsigned sports: 3;
        unsigned dept: 4
    };
    struct stud s1,s2;
    s1.gender = FEMALE;
    printf("Gender of student s1 = %d",s1.gender);
    printf("\nBytes occupied by stud = %d",sizeof(s1));
    return 0;
}
```

Output

```
/tmp/H8FZpukQ1f.o
Gender of student s1 = 1
Bytes occupied by stud = 4
```

Build Process

Build Process – what happens when the code *runs*?

1. When the C code runs there is <u>four stages of C code building process</u>
2. cisoexe
3. Each stage utilizes different 'tools' such as
   a) a preprocessor,
   b) compiler,
   c) assembler, and
   d) linker.

Build Process – what happens when the code *runs*?

1. **Preprocessor:**
   a) **All the preprocessor directives are evaluated and replaced.**
   b) **The input file for this stage is *filename.c* file.**
   c) **The output file is *filename.i* or preprocessed file.**
   d) **Strips out comments from the input c file**

Build Process – what happens when the code *runs*?

2. Compiler:

a) C code gets converted into architecture specific assembly code

b) Decomposition of C operations into numerous assembly operations. Each operation itself is a very basic task. Lexical, syntactical, semantic analysis.

c) The input file for this stage is filename.i file.

d) The output file is filename.s or filename.asm file.

Build Process – what happens when the code *runs*?

3. Assembler:

a) Assembly code that is generated by the compiler gets converted into object code by the assembler.

b) The input file for this stage is filename.asm file.

c) The output file is filename.o or filename.obj file

d) Object Code: machine code, with information that allows a linker to see what symbols are in it and symbols it requires in order to work.

Build Process – what happens when the code *runs*?

4. Linker:

   a) It takes one or more object files as input and combines them to produce a single (usually executable) file.

   b) Executable file = binary form.

   c) In this process filename.exe gets made from filename.obj.

Arguments to main()

argc and argv

1. main() is a function we use in all our C programs.
2. We can pass arguments to main( ) at the command prompt
   a) Called command line arguments
   b) argc and argv

argc and argv

1. argv: Argument Vector: an array of pointers to strings
2. argc: Argument count: an int (number of strings to which argv points)
3. When program is executed:
   a) strings at the command line are stored in memory
   b) address of the first string is stored in argv[0],
   c) address of the second string is stored in argv[1]
4. Why needed? - When we execute a program on a terminal, we might need to pass some arguments that are expected by the program, which can be used during the execution of the program.

argc and argv

```c
#include <stdio.h>
int main(int argc, char** argv) {
    printf("You have entered %d arguments\n", argc);
    int i;
    for(i=0;i<argc;i++){
        printf("Argv[%d] = %s\n", argv[i]);
    }
    return 0;
}
```

```
labuser@labuser-OptiPlex-3010:~/Desktop/try_folder$ gcc argcv.c
labuser@labuser-OptiPlex-3010:~/Desktop/try_folder$ ./a.out this is an example code
You have entered 6 arguments
Argv[0] = ./a.out
Argv[1] = this
Argv[2] = is
Argv[3] = an
Argv[4] = example
Argv[5] = code
labuser@labuser-OptiPlex-3010:~/Desktop/try_folder$
```

Enum

Enumeration

1. Enumeration or Enum in C is a user defined datatype
2. It consists of integer constants that are given names by a user.
3. Why used?
   a) Ease of understanding/maintenance of code
   b) For constants, i.e., when a variable can have only a specific set of values.

Enumeration

1. **Defining the datatype Eg: enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};**
2. **When we declare this, internally, I am assigning the int constant 0 to**

```c
#include<stdio.h>
int main()
{
    enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
    enum week day;
    day = Wed;
    printf("%d",day);
    return 0;
}
```

Enum - Example

```c
#include<stdio.h>
int main()
{
    enum stud_dept{cse, mce, eee, ece};
    enum stud_dept student1, student2, student3;
    student1 = eee; //Internal value = 2
    student2 = ece; //Internal value = 3
    // student3 = bios; //error
    printf("%d  %d  ",student1,student2);
    return 0;
}
```

Output

```
/tmp/uuGZnJDsgI.o
2  3
```

1. **Notes:**
   a) **The possible values an enum can take = enumerators**
   b) **Internally, the enumerators are treasted as ints starting from 0.**
2. **What enums can achieve, can also be done using macros.**

Enum - Example

main.c

Run

```c
#include <stdio.h>

enum days{Sunday=1, Monday, Tuesday, Wednesday, Thursday, Friday,
    Saturday};
int main(){
    for(int i=Sunday;i<=Saturday;i++){
        printf("%d\n",i);
    }
    return 0;
}
```

Output

```
/tmp/H8FZ
1
2
3
4
5
6
7
```

Volatile Qualifier

Volatile Qualifier

1. When we run code, compiler optimizes it.
2. It can use CPU registers to speed things up
3. If we do not want this – use declare such variables as ***volatile*** (eg: volatile int i;)
4. So every time we use this variable, its value is loaded from memory into registers, operated on, and written back to memory.
5. Generally used with variables which are modified by factors external to the program-
   a) Temperature
   b) Speed of a vehicle etc.

Variadic Functions

Variadic Functions

1. In mathematics and in computer programming, a variadic function is a function which accepts a variable number of arguments.
2. Example: printf(), scanf()
3. It takes one fixed argument and then any number of arguments can be passed.
4. The variadic function consists of at least one fixed variable and then an ellipsis(…) as the last parameter.

Variadic Functions

1. General Syntax:
    a) int function_name(data_type variable_name, ...);
2. Values of the passed arguments can be accessed using a header file: stdarg.h

Variadic Functions

1. Methods:
    a) va_start: enables access to variadic function arguments
    b) va_arg: accesses the next variadic function argument
    c) va_copy: makes a copy of the variadic function arguments
    d) va_end: ends traversal of the variadic function arguments
2. va_list:
    a) holds the information needed by va_start, va_arg, va_end, and va_copy
    b) will be the pointer to the last fixed argument in the variadic function

Variadic Functions: Example

```c
#include <stdarg.h>
#include <stdio.h>
int add(int n, ...){ // Variadic function to add numbers
    int sum = 0;
    va_list ptr; // Declaring pointer to the argument list
    va_start(ptr, n);
    for (int i = 0; i < n; i++){
        // Accessing current variable and pointing to next one
        sum += va_arg(ptr, int);}
    va_end(ptr); // Ending argument list traversal
    return sum;}
int main(){
    printf("\n 1 + 2 = %d",add(2, 1, 2)); // Calling variadic function
    printf("\n30 + 40 + 50 = %d",add(3, 30, 40, 50));
    printf("\n6 + 70 + 800 + 9000 = %d",add(4, 6, 70, 800, 9000));
    return 0;}
```

Variadic Functions: Example Output

```
1 + 2 = 3
30 + 40 + 50 = 120
6 + 70 + 800 + 9000 = 9876
```

Thank You!