Kush Shah '23
kshah2@wpi.edu

CS2303 In Class Exercises

Jan 15, 2020

January 9, 2020

Abstract

Today's goal is that you know the primitive data types in C, and how to create your own non-primitive datatypes.

Name and WPI email:

1 Turing Machine Model

The Turing machine model of computation has a control, and a memory. We will discuss the memory, and the idea that the control has a cursor positioned at a specific point in the memory.

2 Memory in the Turing Machine Model

For each cell in the memory, there is an index number, called its address. Think of memory as a roll of paper towels. Each sheet in the roll of paper towels is indexed. The first sheet has index 0. The next sheet has index 1. Each successive address is obtained by adding 1 to the current address.

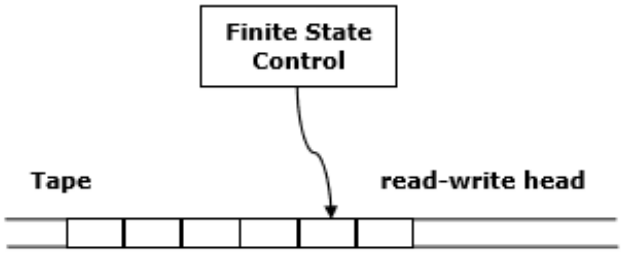

Figure 1: The Turing machine models computation. There is a semi-infinite memory. Think of it as a roll of paper towels that has a start, but no end. Each sheet in the roll of paper towels has a number, starting with 0, and counting up by 1.

3 A Memory Address

3.1 in Turing Machine

In a Turing machine, an address may be any of the natural numbers, starting at 0. This is a countably infinite set.

3.2 in a Finite Machine

In machines we encounter, there will be an upper bound to the memory address.

3.3 in a Memory Chip

Memory chips have some number of address pins. The maximum address for any given memory chip is given by all 1's for the number of address pins it has.

4 A Pointer is an Address

In C, this address concept is very useful. It is renamed pointer. Think of the control's cursor in the Turing machine model diagram. The pointer "points to" one sheet in the roll of paper towels. We associate the sheet in the roll of paper towels with an addressable memory cell. Machines are typically byte-addressable, so we associate the sheet in the roll of paper towels with a memory cell containing 8 bits (called a byte). This means, the pointer corresponds to the index, likewise the address, of that byte.

5 Bytes

5.1 Binary Numbers, and Hexadecimal

Using the binary number system, it takes 4 bits to represent the numbers 0 through 15 inclusive. We like single digits, so after 0 through 9, we make use of A through F to complete the idea of 0 through 15, but using single characters. That way, we can also use 0 through 9, and A-F in the next "column"; this is no longer the 10's column, it is the 16's column.

Fill in the remaining spots in the table. If you have any question about how to do it, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.

Examples:

Word	Decimal	Binary	Hexadecimal
Zero	0	0b0000	0x00
One	1	0b0001	0x01
Two	2	0b0010	0x02
Three	3	0b0011	0x03
Four	4	0b0100	0x04
Five	5	0b0101	0x05
Six	6	0b0110	0x06
Seven	7	0b0111	0x07
Eight	8	0b1000	0x08
Nine	9	0b1001	0x09
Ten	10	0b1010	0xA
Eleven	11	0b1011	0xB
Twelve	12	0b1100	0xC
Thirteen	13	0b1101	0xD
Fourteen	14	0b1110	0xE
Fifteen	15	0b1111	0xF

It took 4 bits to represent 0 through 15 in the unit's column, and 4 additional bits to represent 0 through 15 in the 16's column.

These 8 bits are called a byte, and they are the addressable unit in the hardware.

5.2 Numbers and Bytes

We shall defer numbers with fractional parts for now. Thinking on integers, in mathematics they are classified as negative, zero or positive. In C, we have the idea "unsigned"; in the set of "unsigned" bytes includes 0 through 255. If you reflect on binary notation and the definition of byte as 8 bits, you will be able to determine why 255 is the highest value of an unsigned byte.

5.3 Characters and Bytes

By keeping track elsewhere of the type of interpretation of a given byte, we can use the eight bits within any given byte to mean one of several things. One example is characters. In some smaller alphabets, such as the English alphabet, the range 0 through 255 representable by one byte is sufficient to represent the characters, and we have several codes, commonly we use the ASCII code. Because each digit 0 through 9, and the letters A through F are all characters, we can represent numbers as groups of characters. That implies there are at least two ways to represent the hexadecimal digits: As binary numbers as seen in the table above, but also as characters. When binary number representation is used, 4 bits are enough. When using a character, 8 bits are used.

You can find the ASCII representation in many places, including `https://www.ascii-code.com/` Fill in the remaining spots in the table. If you have any question about how to do it, please ask. In subsequent lectures this material

will be considered background, so please take this opportunity to get it clarified.

Character	ASCII
a	97
A	65
b	98
B	66
f	102
F	70
0	48
1	49
2	50

5.4 Signed Numbers, Two's Complement

By contrast with unsigned numbers, we often want to be able to represent negative numbers. An initial surmise is that one bit is reserved for the sign (good idea). What might not be immediately imagined is that there is a good reason for using the two's complement form for representing negative numbers. To understand two's complement, consider that -1 is the number to which, when we add 1, we get zero.

$$x + 1 = 0$$

So, how might we best represent -1 with our byte? Consider, all of the eight bits in the byte are 1. When we add 1 to that, each bit, starting on the right, but carrying through all 8 bits, the 1 changes to a 0.

```
  11111111
+00000001
----------
  00000000
```

and there is a "carry bit" that is one, which we can discard.

Thus, we added 1 to something, we got zero, so that something must be -1.

5.5 Using a byte to represent one of several possible ideas

Suppose there is some other memory keeping track of which interpretation is being used for a given byte. Then we can look at a byte, augment it with an interpretation, and see the idea that is being represented.

Fill in the remaining spots in the table. If you have any question about how to do it, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.

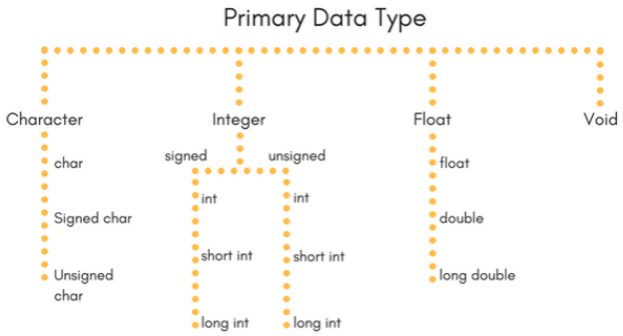

Figure 2: C datatypes, arranged.

Binary	Unsigned Byte	Signed Byte	ASCII
00000000	zero	zero	Null character
00000001	one	one	Start of heading
00000010	two	two	Start of text
...			
01111110	126	126	~
01111111	127	127	don't care
10000000	128	-128	don't care
10000001	129	-127	don't care
10000010	130	-126	don't care
...			
11111100	252	-4	don't care
11111101	253	-3	don't care
11111110	254	-2	don't care
11111111	255	-1	don't care

6 Using Multiple Bytes Together

C uses multiple bytes, grouped, to represent data. C uses the name datatype to identify which group of bytes. Data could be from the set of integers that goes higher than positive 255, and goes lower than negative 128. Multiple bytes are needed to represent character sets with more than 256 members. Multiple bytes are used to represent numbers with fractional parts.

To find out how many bytes C is using on your platform, use the operator sizeof(). C supports multiple datatypes. You might like this diagram from `https://www.studytonight.com/c/datatype-in-c.php`.

Though the diagram omits pointers, C also has pointers. For each of the datatypes in C, there is a C datatype that is a pointer (address of) that datatype. Are you wondering about this definition, that perhaps it does not have a termination condition? For example, suppose the datatype integer. The existence

of the datatype integer implies the existence of an address for any instance of integer. Now that we have the datatype address-of-an-integer, (denoted int*), we also must have the idea of an address for that address. It is denoted int**. There is also the idea of address of one of those, and unsurprisingly it is denoted int***. After a while, it becomes uninteresting to keep track of all these levels of addresses. For those comfortable with the idea of RAM and other memory chips, and the idea of a memory address register (MAR), consider the integer to be located in RAM at some address, and consider that that address is saved in RAM, with an address of its own. Then, we can access the address in RAM, put the contents of the databus into the MAR, then use that value as the address to get at the integer. We can do this process of using the data as an address, as many times as we like.

7 Variables

No matter what datatype our byte, or group of bytes, has, it is convenient to attach a name to it. You can write code that is obscure to read by naming your data as var1, var2, var3 etc. Preferably, you will choose names that convey something of the meaning of your data.

int numClasses = 5;

MovieTitle could be the name of a group of bytes that are being interpreted as a list of characters. Of course there are movie titles of a single character. We could write:

char movieTitle = 'Z';

When we have a collection of data, each datum of the same type, we can use an array. We might have a collection of movie titles that we like, and another collection of those we don't.

char goodMovies[100][50];
char badMovies[100][50];

Each collection allows 100 movies, the names of which may each be as long as 49 characters. (The last character is reserved to be '\0'.)

Then, to select at an element in the collection, we can use an index.

char favoriteMovie[50];
strcpy(favoriteMovie, goodMovie[3]);

One idea about variables is that some memory has to be reserved for them (so, they will have an address as a result of that reservation). We can also keep track of their address if we wish.

float my2pi = 3.141 *2;

float* whereMy2piIs = &my2pi;

The amount of memory reserved depends upon the datatype. Variables that are **double** take more bytes that variables that are **float**.

8 Pointers to Variables

Suppose we have two variables:
float my2pi = 3.141 *2;

and
double myOther2pi = 3.14159 *2;

We have pointers to these.
float* whereMy2piIs = &my2pi;

and
double* whereMyOther2piIs = &myOther2pi;

To access the memory just after that reserved for my2pi, the address of my2pi is incremented:

***(my2pi+1)**

refers to the next float after the float my2pi.

To access the memory just after that reserved for myOther2pi, the address of myOther2pi is incremented:

***(myOther2pi+1)**

refers to the next double after the double myOther2pi.

Recall that pointers are addresses, and addresses identify bytes. Recall that double uses more bytes than float. It makes sense that pointers, which participate in pointer arithmetic, have a notion of the datatype to which they point.

9 Practice Declaring Variables of Primitive Datatypes

Variable names may not begin with digits. Variable names do not include spaces or punctuation. Later (soon) we will see the use of the period.

It is a significantly good idea to initialize variables. The reason is, many programmer errors arise from failure to initialize.

To initialize the value of a pointer, declare the variable to which it points, first. If this is not suitable in the application, use 0; Either the pointer will be set to some other value during execution, prior to being used to read, OR, if the programmer makes a mistake, the code using the variable will try to read from or write to address 0, which will cause a segmentation fault. This is readily

found and fixed. The alternative, which is using the variable uninitialized, will cause random behavior, not necessarily recognized, which is a hassle to find (This is bad.).

Fill in the remaining spots in the table. If you have any question about how to do it, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.

Task:	Datatype	Variable Name	Initialization
Declare and initialize a variable of type integer	int	x	=0;
Declare and initialize a variable of type character	char	aLetter	='w';
Declare and initialize a variable of type float	float	xF	=0.0;
Declare and initialize a variable of type double	double	xD	=4.5;
Declare and initialize a variable of type pointer to integer	int*	xP	=&x;
Declare and initialize a variable of type pointer to character	char*	LetterP	=&aLetter;
Declare and initialize a variable of type pointer to float	float*	xFP	=&xF;

10 Do It Yourself Datatypes

10.1 struct

Developers frequently have ideas that are expressible using aggregations of the primitive datatypes in C. For example, contact information can include a person's name, address, birthday, shirt size, etc. Imagine have a datatype for contact information, then a pointer to that datatype, and the ability to move through a list of contacts by adding 1 to the pointer to the datatype.

Let's make a simpler example to begin. Parts will be the same every time, and the rest will be the decision of the developer.

```
typedef struct
{
  int justACountingNumber;
  double canHaveAFractionalPart;
  char aLetterOrDigit;
```

```
  long anIntegerThatCanBeBig;
} nameOfCustomType;
```

The first line, **typedef struct**, and the { after it, will be the same every time.
The last line will always start with } will have some datatype name, and will always end with a semicolon.

Here is an exercise. If you have any question about how to do it, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.

Recall that complex numbers consist of two real numbers. For a real number, use the C datatype double. Write a datatype for a complex number.

Imagine a datatype that combines an integer followed by a single letter. For example 25B. Write a datatype for this.

```
typedef struct{
int theNum;
char theChar;}myDataType;
```

To declare a variable of any datatype that is a struct, use the datatype name and give a variable name.

For example, suppose we have:

```
typedef struct
{
  int justACountingNumber;
  double canHaveAFractionalPart;
  char aLetterOrDigit;
  long anIntegerThatCanBeBig;
} Motley4;
```

We can declare an instance of type Motley4; we need a variable name;

```
Motley4 m4;
```

To set values of m4, set the values of its component fields individually. For example,

```
  m4.justACountingNumber = 7;
m4.aLetterOrDigit = 'q';
```

Practice: (If you have any question about how to do it, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.)

Set values into variables of the type Complex you defined earlier.

We also might want to read values from the fields of a compound datatype. Variable names to the left of the equal sign are getting assigned, just as when we initialize. To read the value from a field, we identify the field we want on

```
typedef struct{
double x;
double y;
} complex;
```

the right hand side of the equal sign.

double someDouble = m4.canHaveAFractionalPart;

Practice: (If you have any question about how to do it, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.)

Read a value from a field of a variable of the type Complex you defined earlier, into a newly declared variable (which should be of the appropriate type).

10.2 enumerated types

Mathematics includes the idea of finite sets. Being finite, for a set, implies that it is possible to enumerate the members of the set. This idea is useful in programming, and C provides a means of specifying the members of such a set. The members will be attached to distinguishable integers. The compiler will perform this, and if the developer desires, the developer may specify the integer to which a member is attached; the developer's specification must allow the members to be distinguishable using the integers.

Example:

```
typedef enum
{
     Monday,
     Tuesday,
     Wednesday,
     Thursday,
     Friday = 100,
     Saturday,
     Sunday

}dayOfWeek;
```

11 Changing the Datatype of Data

It can happen that a variable has some content that makes sense in a datatype different from the datatype of that variable. We can change the datatype that is applied to the data. For example, we might have a character variable, and wish to do some arithmetic with it. The Caesar cipher starts with a letter of the (let us say, English) alphabet, and maps it to the letter offset by some integer. Suppose we map a letter to the letter next after it. For 'z', it becomes 'a'. So we have a character, we want to think of it as a number, so that we can add 1 to it, so long as it is not a z. We can use the process of casting.

```
char aChar = 'a';
int aCharI = (int) aChar;
```

```
int anotherInt = aCharI+1;
char anotherChar = (char)anotherInt;
printf("The resulting character is %c.\n", anotherChar);
```

Practice: (If you have any question about how to do it, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.)

1. Using the code above, change it so that the letter is initially some alphabetic character other than a, but not w,x,y or z also not W,X,Y or Z. Also, add 3 to it.

```
aChar = 'k';
int added = (int) aChar +3;
cNewChar = (char) added;
```

   Note that, in ASCII, capital letters start at A=65, and small letters start at a=97. Write code that will change a capital letter into the corresponding lower case letter.

```
char aLetter = 'A';
char aLetter = (char)
((int) aLetter + 32);
```

2. Create a datatype that is an enumerated type. In the datatype, list all WPI buildings in which you have a class this term.

```
typedef enum
{
Stratton Hall,
Fuller,
Olin
}Buildings;
```