16.070 Introduction to Computers and Programming

March 14 **Recitation 6** **Spring 2002**

Topics:

- Quick review of PS4 issues
- Fundamental Data types
- ASCII / Arithmetic Conversion
- Number Systems / Logical Operation
- Data Representation
- Sampling

Common Programming Mistakes

At office hours and while grading a number of mistakes have surfaced repeatedly.
Here are a few:

1. Pointer variables should be initialized to the constant NULL.
2. The address a pointer points to is incremented by pointer arithmetic according to the size of the variable type that the pointer is declared as.
3. Constants (#define and const) should be all uppercase.
3. Function prototypes specified in homework should be used without modification.

Fundamental Data Types

The fact is that data types are platform specific. IBM, SGI, Sun and other manufacturers all adhere to their own standards regarding the amount of memory assigned for different data types. ANSI-C does provide a minimum standard, indicating **at least** how many bytes should be assigned to each standard data type.

Variable Type	Keyword	Bytes Required	Range
Character	`char`	1	-2^7 to 2^7-1
Integer	`int`	2	-2^{15} to $2^{15}-1$
Short Integer	`short`	2	-2^{15} to $2^{15}-1$
Long Integer	`long`	4	-2^{31} to $2^{31}-1$
Unsigned Character	`unsigned char`	1	0 to 2^8-1
Unsigned Integer	`unsigned int`	2	0 to $2^{16}-1$
Unsigned Short Integer	`unsigned short`	2	0 to $2^{16}-1$
Unsigned Long Integer	`unsigned long`	4	0 to $2^{32}-1$
Single-Precision Floating-Point*	`float`	4	-10^{38} to 10^{38}
Double-Precision Floating-Point**	`double`	8	-10^{308} to 10^{308}

* Approximate precision to 7 digits
** Approximate precision to 19 digits

Table 1 ANSI-C minimum memory requirements for standard data types

How do we know how much memory is allotted for each data type on your specific platform? Programming languages have a *sizeof(),* or equivalent, statement. It is always good style to use *sizeof()* in C when you rely on the amount of memory used by your program. This statement returns the size of any data type e.g.

```
int a = 0;
a=sizeof(int);
printf("integers have size: %d bytes",a);
```

Output:

```
integers have size: 4 bytes
Press any key to continue
```

You will notice that integers on the PC platform, running MS Windows, consume more memory than the minimum ANSI specification of 2 bytes.

ASCII Character Table

You may have noticed that the above table specifies a variable of type *char* to have a range from 0 to 255. How can a character correspond to a number?
The **ASCII** ("American Standard Code for Information Interchange") codes are used to represent characters as one byte integers. The first 128 of them (0 → 127) are the standard ASCII characters, while the next 128 (128 → 255) are the extended ASCII characters (symbols, accented letters, Greek letters, etc...).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Table 2 Hex-indexed ASCII table

Some examples…

Character	Decimal	Hex	Octal	Binary
space	32	20	40	0010 0000
!	33	21	41	0010 0001
"	34	22	42	0010 0010
#	35	23	43	0010 0011
$	36	24	44	0010 0100
A	65	41	101	0100 0001
B	66	42	102	0100 0010

Control Codes

NUL (null) SOH (start of heading) STX (start of text) ETX (end of text) EOT (end of transmission) - Not the same as ETB ENQ (enquiry) ACK (acknowledge) BEL (bell) - Caused teletype machines to ring a bell. Causes a beep in many common terminals and terminal emulation programs. BS (backspace) - Moves the cursor (or print head) move backwards (left) one space. TAB (horizontal tab) - Moves the cursor (or print head) right to the next tab stop. The spacing of tab stops is dependent on the output device, but is often either 8 or 10. LF (NL line feed, new line) - Moves the cursor (or print head) to a new line. On Unix systems, moves to a new line AND all the way to the left. VT (vertical tab) FF (form feed) - Advances paper to the top of the next page (if the output device is a printer).	CR (carriage return) - Moves the cursor all the way to the left, but does not advance to the next line. SO (shift out) - Switches output device to alternate character set. SI (shift in) - Switches output device back to default character set. DLE (data link escape) DC1 (device control 1) DC2 (device control 2) DC3 (device control 3) DC4 (device control 4) NAK (negative acknowledge) SYN (synchronous idle) ETB (end of transmission block) - Not the same as EOT CAN (cancel) EM (end of medium) SUB (substitute) ESC (escape) FS (file separator) GS (group separator) RS (record separator) US (unit separator)

Type Casting

Sometimes we don't like the standard rules of arithmetic conversion to be applied to us. In such cases we may decide to *cast* variables or results of arithmetic into specific types. Lets look at an example:

```
double Velocity, Time_Elapsed;
Distance = Velocity*Time_Elapsed;
```

We would usually need to define *Distance* as being of data type *double*. A compiler like Visual C would even compile the code if *Distance* was defined as *int*, but it would issue a warning message. This is not necessarily true for all compilers! Other C compilers, such as Interactive C, which we will use when programming the Handy Boards, are more strict about types. Many compilers don't even hold with the arithmetic conversion rules that you have learned. The safest bet is to make use of a process called *type casting*. The correct way to cast a variable of one data type into another would be:

```
/* variable declaration */
int Distance = 0;
double Velocity = 10.0;
double Time_Elapsed = 100.0;

/* Type casting */
Distance = (int)(Velocity*Time_Elapsed);
```

Arithmetic Conversions

Sometimes equations in your code will contain variables of different types, and all in the same equation. What should the type of the answer to such an equation be? Standard C compiler rules exist that deal with such cases:

1. C temporarily converts the value of the “lower” type to that of the “higher” type, and then performs the operation, producing a value of the “higher” type (see hierarchy table on page 4).
2. Characters and short ints are converted to int before operations are performed.
3. You can use casts to force the conversion of one data type to another. The use of a cast overrides the standard rules for type conversions. The cast operator does not change the value stored n the variable.
4. Casting is done by explicit type conversions immediately before an expression. Place the type in parentheses immediately in front of the variable you want to change.
5. Unsigned property does not necessarily propagate from lower to higher. Do not mix unsigned and signed data types in operations; use casts. The reason is that the storage of unsigned types varies from computer to computer and hence if you do not use casts, you may end up with unwanted results.
6. All decimal numbers without a declared type default to double.

Conversion Hierarchy Table

Hierarchy of Signed Data Types	**Hierarchy of Unsigned Data Types**
long double	unsigned long
double	
float	unsigned
long int	unsigned short
int	unsigned char
short int	
char	

The above table indicates that the *long double* data type is at the top of the hierarchy, with *char* being at the bottom.

Examples:

1.
```
int a = 0;
  double b = 0.0;
  c = a + b; /* c would need to be of type double (Rule 1) */
```

2.
```
int a = 0;
  c = a + 5; /* c would need to be of type integer (Rule 2) */
```

Number Systems

In everyday situations, we are used to using a decimal (base 10) number system. It has digits 0 through 9 and each digit's position in a number is ten raised to the power of the position multiplied by the number:
thus, $123 = (1 * 10^2) + (2 * 10^3) + (3 * 10^0)$

Three different number systems are commonly used in software engineering: binary (base 2), octal (base 8), and hexadecimal (base 16). In binary, the only digits available are 0 and 1. In octal the only digits available are 0 through 7. In hexadecimal, 0 through 9 are used, plus 5 additional digits (A,B,C,D,E,F) are used to represent (10,11,12,13,14,15).

Example: The same numbers are written below is several different number systems.

Decimal	**Binary**	**Octal**	**Hexadecimal**
129	10000001	201	71

Often times (especially when you're learning) writing the powers of two above a binary number will make it easier to read:

$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
1	0	0	0	0	0	0	1

Converting between binary, octal, and hexadecimal number systems only involves regrouping digits. A hex number can be created from a binary number by putting the binary digits into groups of four starting from the least significant bit (rightmost bit) and then allowing each group of four digits represent a hex digit. Likewise, an octal number is created from a binary number using the same approach, but putting bits into groups of 3.

Example: The same numbers are converted from binary to octal and from binary to hexadecimal.

Binary	**Octal**	**Binary**	**Hexadecimal**
10 000 001	201	1000 0001	81
1 010 100	124	101 0100	54

Converting binary, octal, or hexadecimal numbers to decimal numbers is accomplished by multiplying each digit by the base raised to the position of the digit in the manner described at the beginning of this section. ($129 = \underset{\text{binary}}{1*128 + 1*1} = \underset{\text{octal}}{2*64 + 1*1} = \underset{\text{hex}}{7*16 + 1*1}$)

Converting decimal numbers to binary, octal, or hexadecimal is not as elegant a process. Start with the highest power of the base (2, 8, or 16) that can be subtracted out of the decimal number and subtract the highest multiple of that power possible. The multiple of that power is the digit corresponding to that power in the new base. Repeat this operation on each result until the number being converted is 0.

Example: The number 129_{10} will be converted to octal.
$129 - 64*2 = 1$ ➔ digit corresponding to 8^2 is 2
$1 - 1*1 = 0$ ➔ digit corresponding to 8^0 is 1
Thus $129_{10} == 201_8$

Number System Operations

Just as it is possible to perform a variety operations such as addition, subtraction, multiplication, and division for decimal numbers, it is also possible to perform these operations on binary, octal and decimal numbers.

For instance: $FEED_{16} + FACE_{16} = ?$

$$\begin{array}{rr} & FEED_{16} \\ + & FACE_{16} \\ \hline & 1F9BB_{16} \end{array}$$

An interesting similarity between number systems occurs when you multiply or divide a number by its base. Multiplying a base 10 number by 10 will add a zero to the right end of the number ($34_{10}*10_{10}=340_{10}$). Likewise, multiplying a base 2 number by 2 will add a zero to the right end of the number ($1001_2 * 2_{10} = 10010_2$). This property holds when an octal number is multiplied by 8 and when a hexadecimal number is multiplied by 16.

A similar property exists for dividing by a number's base. Just as when 340 is divided by 10 the result is 34, when 1010_2 is divided by 2_{10} the result is 101_2. Likewise, when a hexadecimal number 140_{16} is divided by 16_{10}, the result is 14_{16}.

The process of multiplying or dividing a binary number by two shifts the bits in that number to the left (multiplying by 2) or to the right (dividing by 2). Another way to shift bytes left and right is using the shift operators in C: `>>` and `<<`

For instance the C expression:

```
int x = 8 << 2;
```

will shift the binary representation of 8 left two bits and assign the result to x.
Thus, $8_{10} == 1000_2$, $1000_2 << 2 == 100000_2 == 32_{10} == 8*2*2$

Consequently $8 >> 2 == 2$, because $1000_2 >> 2 === 10_2$

Shift operates by manipulating the binary bit representation of a number.
There is also a set of logical bitwise operations:

Operation name	C operator	Function
bitwise and	&	Bitwise and will AND each bit of the binary representation of a number individually and produce an output number composed of the results of the individual ands
bitwise or	|	Like bitwise and, but instead will OR each bit of the binary representation of a number
bitwise xor	^	Like bitwise and, but instead will XOR each bit of the binary representation of a number
bitwise not	~	Like bitwise and, but instead will NOT each bit of the binary representation of a number

An example of bitwise operation would be: `6 ^ 4 == 2`

$$\begin{array}{rr} & 110_2 \\ \text{xor} & 100_2 \\ \hline & 010_2 \end{array}$$

An excellent combined use of shift and bitwise operations would be to ascertain the configuration of a particular set of bits. This could be accomplished by first ANDing all other bits with 0 and then shifting the number right until the bits in question are the list significant and the only bits remaining. Then, their values can
For instance, to find out what value is indicated by the two leftmost bits in a 7 bit number, bitwise AND that number with 96 (1100000_2) and then rightshift 5. The result will be 3 indicating both bits bits in question are 1.

Data Representation

Negative Numbers

There are a number of ways to represent the sign of a number, even when that number is represented by a series of bits in the computer. One of the simplest ways is to reserve one bit to hold a number's sign. Using the most significant bit for the sign (0 = positive, 1 = negative) and the remaining 7 bits for the number, the range of an 8 bit number would go from –127 to +127. This method is call *signed magnitude*.
If our convention were that a 1 in the sign bit indicated a negative number,
-2 would be: 10000010_2

Signed magnitude is generally considered inefficient and confusing, because two possible representations exist for 0.

Another approach to representing negative numbers is called *two's complement*.
To use the two's complement approach, creating a negative number is accomplished by inverting each bit of the corresponding positive number (like the bitwise "not" operation described on the previous page) and then adding 1. It turns out that inverting 0 with this method yields 0, thereby eliminating the problem of dual representation. The range of possible values using an 8 bit integer with the two's complement system is –128 to +127.

For instance, negative 2 in an 8 bit two's complement system would be found by taking the binary representation of 2: $0000\ 0010_2$, inverting the bits to get $1111\ 1101_2$ and then adding 1 to get: $1111\ 1110_2$

Verifying Authenticity – Parity Checking

Often times, when transferring data from one location to another, that data will become altered in some way. Many different methods for ensuring data integrity (and even restoring corrupted data) currently exist. One simple method that can identify single bit changes in a number is called *parity checking*. With *parity checking*, one bit in a number is reserved as the parity bit. The value of this bit is determined by summing the number of bits set to "1". If the convention being used is even parity, the sum of the bits in the number should add up to an even number. Thus, if they don't, the parity bit will be set to 1 to ensure that the bits **do** add up to an even number.

Example: using an even parity convention with the parity bit being the most significant bit, the number 76 is to be transferred

$76_{10} == 01001100_2$

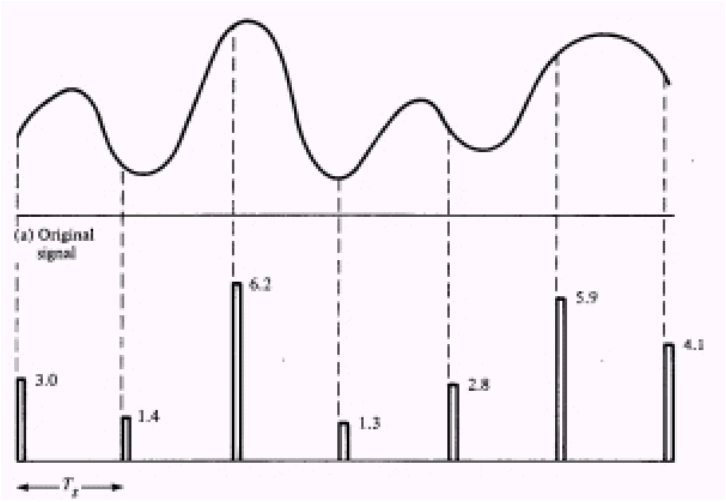
the sum of the digits = 1+0+0+1+1+0+0 = 3

because 3 is an odd number, the parity bit must be set to 1 to make the data transferred have even parity… thus, the resulting number is:
$11001100_2 == 204_{10}$

Sampling

The process of converting naturally occurring measurements in the real world into values in a computer that may be analyzed or acted upon is the process of converting continuous analog data into digital data. Analog data has many advantages in that it is easy for humans to process and it occurs naturally with virtually infinite detail. Conversely, digital data has the advantage of being easy for computers to process, is perfectly reproducible, and is quantitative. Examples of analog data are photographs, cassettes, and human memory. Examples of digital data are compact discs, floppy disks, and DSL lines.

In the picture below, data are converted from a continuous curve to a series of discrete point measurements. This is done through a process known as *sampling*, whereby data is measured at a regular interval.



One of the pitfalls of sampling is encountered when the frequency of the continuous data being measured is higher than that of the sampling itself. In the picture below, two separate frequencies are present, but if sampling only occurred every 0.1 seconds, a person trying to analyze the data would not know which was the frequency present. This problem is known as aliasing and can be prevented by sampling at more than twice the highest frequency of interest (known as the Nyquist Criterion) and filtering out all frequencies higher than that.