Western Washington University
Computer Science Department

CSCI 141 Computer Programming I
Fall 2011

Assignment 1

Submitting Your Work

This assignment is worth 15% of the grade for the course. In this assignment you will create two Ada programs: one to encode ASCII data in Base64 format, and the other to decode from Base64 back to ASCII. Save your program files (the .adb files) in the zipped tar file WnnnnnnnnAssg1.tar.gz (where Wnnnnnnnn is your WWU W-number) and submit the file via the **Assignment 1 Submission** item on the course web site. You must submit your assignment by 10:00am on Monday, October 24.

Your assignments will be evaluated on correct functionality and conformance to the coding standards described at the end of this assignment specification. In this case "correct functionality" means that your programs correctly encode and decode text whose length is a multiple of 3 ASCII characters. As described below, encoding and decoding text which is not a multiple of 3 ASCII characters is a little more complicated. You can gain 2 points of extra credit for handling the general case.

Input to each program is a single line of text from the keyboard.

ASCII code

The American Standard Code for Information Interchange (ASCII) was developed in 1960 to provide a standard numeric representation of text characters in computers and communications. ASCII defines 128 characters, including 33 unprintable control characters, 94 printable characters and the space character.

In the move toward internationalization of computer systems, ASCII has been superseded by the Latin-1 character set for use in countries that use English and European languages. However, the ASCII characters are the first 128 characters of the 256 Latin-1 character set. In order to have distinct representations of 256 characters, Latin-1 needs 8 bits per character.

The Ada character type is the Latin-1 character set and so uses 8 bits per character. The ASCII code for a character is identical to the value of character'pos() attribute in Ada. The character corresponding to an ASCII code is identical to the value of the character'val() attribute in Ada.

Since $2^7 = 128$, ASCII uses 7 bit (binary digits) for each character. The binary representation of the ASCII characters is shown in the table below. For each character, its

binary representation is $b_7b_6b_5b_4b_3b_2b_1$. The decimal value of the ASCII code is $b_7\times2^6 + b_6\times2^5 + b_5\times2^4 + b_4\times2^3 + b_3\times2^2 + b_2\times2^1 + b_1\times2^0$.

For example, the ASCII code for the character 'A' is:

$1000001 = 1\times2^6 + 0\times2^5 + 0\times2^4 + 0\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0 = 64 + 1 = 65$

Since Ada uses 8 bits per character, the 8th bit is always 0 for ASCII characters.

b_7 →					0	0	0	0	1	1	1	1
b_6 →					0	0	1	1	0	0	1	1
b_5 →					0	1	0	1	0	1	0	1
Bits	b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Column → / Row ↓	0	1	2	3	4	5	6	7
	0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
	0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
	0	0	1	0	2	STX	DC2	"	2	B	R	b	r
	0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
	0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
	0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
	0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
	0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
	1	0	0	0	8	BS	CAN	(	8	H	X	h	x
	1	0	0	1	9	HT	EM	)	9	I	Y	i	y
	1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
	1	0	1	1	11	VT	ESC	+	;	K	[	k	{
	1	1	0	0	12	FF	FC	,	<	L	\	l	|
	1	1	0	1	13	CR	GS	-	=	M	]	m	}
	1	1	1	0	14	SO	RS	.	>	N	^	n	~
	1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Illustration 1: The ASCII Binary Codes

Base64 Code

Base64 is an alternative encoding scheme for representation of binary ASCII data that need to be stored in or transferred over media that are designed to deal with textual (printable) data. Base64 encoding of ASCII is used to ensure that the data remains intact without modification during storage or transportation. It is widely used in application software including email via MIME (Multipurpose Internet Mail Extensions), and storing complex data in XML (Extendable Markup language).

Base64 uses the 64 printable characters 'A' through 'Z', 'a' through 'z', '0' through '9', '+' and '/'. Since there are 64 distinct Base64 characters, each requires 6 bits for unique representation. In addition, the '=' character is used for padding at the end of the message if necessary, as described below.

The following table shows the binary values of the 64 Base64 characters.

Value	Char	Value	Char	Value	Char	Value	Char
000000	A	010000	Q	100000	g	110000	w
000001	B	010001	R	100001	h	110001	x
000010	C	010010	S	100010	i	110010	y
000011	D	010011	T	100011	j	110011	z
000100	E	010100	U	100100	k	110100	0
000101	F	010101	V	100101	l	110101	1
000110	G	010110	W	100110	m	110110	2
000111	H	010111	X	100111	n	110111	3
001000	I	011000	Y	101000	o	111000	4
001001	J	011001	Z	101001	p	111001	5
001010	K	011010	a	101010	q	111010	6
001011	L	011011	b	101011	r	111011	7
001100	M	011100	c	101100	s	111100	8
001101	N	011101	d	101101	t	111101	9
001110	O	011110	e	101110	u	111110	+
001111	P	011111	f	101111	v	111111	/

Encoding ASCII in Base64

In a string of ASCII text, each sequence of 3 ASCII codes requires a total of 24 bits. For encoding in Base64, that 24 bit sequence is divided into 4 Base64 codes.

For example, the ASCII codes for “Man” is encoded in Base64 as “TWFu”, as shown below.

ASCII	M								a								n							
Binary	0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Base64	T						W						F				u							

When the number of ASCII characters is not a multiple of 3, the remaining bits of the 3-character sequence are filled with zeros. During encoding in Base64, if there was only one ASCII character in the 24-bit sequence, only the first 2 Base64 characters are used and two '=' characters are used for padding. If there were only two ASCII characters in the 24-bit sequence, only the first 3 Base64 characters are used and one '=' character is used for padding.

For example,

- The Base64 encoding of “tomcat” is “dG9tY2F0”.

- The Base64 encoding of “tomca” is “dG9tY2E=”.
- The Base64 encoding of “tomc” is “dG9tYw==”.

Decoding from Base64 to ASCII

The process for decoding from Base64 to ASCII is the reverse of the encoding process. From a string of Base64 characters, each sequence of four 6-bit Base64 characters is considered as a 24 bit sequence which is then divided into three 8-bit ASCII binary codes.

A sequence of 4 Base64 characters ending in “==” results in only one ASCII character. The 12 bits from the leading two Base64 characters need to be reduced to just 8 bits for the one ASCII character. This is done by discarding the last 4 bits (divide by 16). For example, the first two Base64 characters of “cw==” have binary representation 011100110000. After discarding the trailing 4 zeros, we are left with 01110011, the ASCII character 's'.

A sequence of 4 Base64 characters ending in a single '=' results in only two ASCII characters. The 18 bits from the leading three Base64 characters need to be reduced to just 16 bits for the two ASCII characters. This is done by discarding the last 2 bits (divide by 4). For example, the first three Base64 characters of “c3U=” have binary representation 011100110111100100. After discarding the trailing 2 zeros, we are left with 01110011 and 01110101, the ASCII characters 's' and 'u'.

A base64 sequence not ending in '='results in three ASCII characters. For example, the Base64 “c3Vy” has binary representation 011100110111010101110010. This is divided into the three 8 bit binary codes 01110011, 01110101 and 01110010, the ASCII characters 's', 'u', and 'r'.

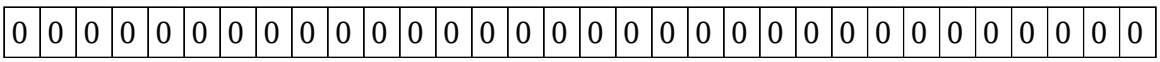
So how do you do all this stuff?

How to we mess around with binary numbers in Ada? Well, fortunately, the answer is : we don't! All we need to do is pack character codes of some length (6 or 8 bits) into an integer and then unpack the character codes of the alternative length (8 or 6 bits) from that integer number.

The representation of an integer on the computer is either 32 or 64 bits. We just need the first (rightmost, least significant) 24 bits to accommodate 3 ASCII codes or 4 Base64 codes.

Packing ASCII Codes into an Integer

Let's say that we want to pack the ASCII characters 'C', 'a' and 't' into an integer. We start the integer (let's give it the name `Number`) with the value 0. For a 32-bit number that is



We place the ASCII code for 'C', which is 01000011 in the rightmost (least significant) 8 bits of `Number`:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**

How do you do that in Ada? Simple:

```
Number := character'pos('C');
```

Now, we need to move those rightmost 8 bits 8 places to the left and insert the ASCII code for 'a', which is 01100001, in the rightmost 8 bits of `Number`.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	0	1	1	0	0	0	0	1

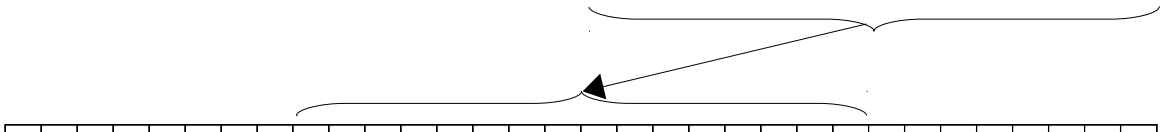
How do you do that in Ada?

```
Number := character'pos('a') + Number * (2 ** 8);
```

Note: to shift the bits n places to the left, simply multiply by 2^n.

Now, we will move those rightmost 16 bits 8 places to the left and insert the ASCII code for 't', which is 01110100, in the rightmost 8 bits of `Number`.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**1**



0	0	0	0	0	0	0	0	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	0	1	1	1	0	1	0	0

How do you do that in Ada?

```
Number := character'pos('t') + Number * (2 ** 8);
```

Note that the first step, in which we inserted the ASCII code for 'C' into `Number`, could have been done exactly the same way as the other two steps:

```
Number := character'pos('C') + Number * (2 ** 8);
```

Do you see a loop here?

Unpacking Base64 Codes from an Integer

Given that we now have the 32-bit integer

0	0	0	0	0	0	0	0	**0**	**1**	**0**	**0**	**0**	**0**	1	1	0	1	1	0	0	0	0	1	0	1	1	1	0	1	0	0

How do we pull out 4 Base64 characters?

To get the first Base64 character, we need the bits shown in **bold** characters above. To get them, we simply shift the whole number 18 bits to the right, leaving:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**1**	**0**	**0**	**0**	**0**

How do we do that in Ada? Let's call the Base64 value `Code`.

```
Code := Number / (2 ** (6 * 3));
```

Note: to shift the number n bits to the right, divide it by 2^n. In this case we are shifting right by 3 groups of 6 bits each, 18 bits altogether.

Now we can discard those leftmost 6 bits from `Number`, leaving

0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**0**	**1**	**1**	**0**	0	0	0	1	0	1	1	1	0	1	0	0

How do you do that in Ada?

```
Number := Number mod (2 ** (6 * 3));
```

Now we want the second Base64 character from the bits shown in **bold** characters above. This time we shift the number 12 bits to the right, leaving:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**0**	**1**	**1**	**0**

How do we do that in Ada?

```
Code := Number / (2 ** (6 * 2));
```

This time, we are shifting right by 2 groups of 6 bits each.

Now we can discard those leftmost 6 bits from `Number`, leaving

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**0**	**1**	1	1	0	1	0	0

How do you do that in Ada?

```
Number := Number mod (2 ** (6 * 2));
```

Now we want the third Base64 character from the bits shown in **bold** characters above. This time we shift the number 6 bits to the right, leaving:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	**0**	**1**

How do we do that in Ada?

```
Code := Number / (2 ** (6 * 1));
```

This time, we are shifting right by just 1 group of 6 bits

Now we can discard those leftmost 6 bits from `Number`, leaving

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**0**	**1**	**0**	**0**

How do you do that in Ada?

```
Number := Number mod (2 ** (6 * 1));
```

Now we want the fourth base64 character from the bits shown in **bold** characters above. This is easy in Ada

```
Code := Number;
```

Which is actually the same as:

```
Code := Number / (2 ** (6 * 0));
```

Do you see a loop here?

Decoding: Packing Base64 Codes into an Integer

This proceeds the same as packing ASCII codes into an integer, except that at each step we shift `Number` to the left by 6 bits (multiply by 2 ** 6), rather than 8 bits.

Decoding: Unpacking ASCII Codes from an Integer.

This proceeds the same as unpacking Base64 codes from an integer, except that our integer divide and mod operations are in multiples of 2^8, instead of multiples of 2^6.

Saving your files in a zipped tar file

1. You only submit the .adb files. First you need to bundle them up into a single tar file. The term "tar" is an abbreviation of "tape archive" and goes back to the days when people would save a back-up copy of their files on magnetic tape. Nowadays, with the price of large disk drives so low, nobody uses tape anymore, but the concept of tar files survives.

   Use the command:

   ```
   tar -cf WnnnnnnnnAssg1.tar *.adb
   ```

   (where Wnnnnnnnn is your W-number).

   The -cf specifies two options for the tar command: 'c' means create and 'f' means that the name of the resulting tar file comes next in the command.

   Following the name of the tar file, we list the files to be included in the tar file. In this case, we want the files `whose names end with ".adb"`.

2. If you now use the command `ls` you should now see the file WnnnnnnnnAssg1.tar in your directory.

3. If you use the command

   ```
   tar -tf WnnnnnnnnAssg1.tar
   ```

   (where Wnnnnnnnn is your W-number), it will list the files within the tar file.

4. Now compress the tar file using the gzip program:

   ```
   gzip WnnnnnnnnAssg1.tar
   ```

   By using the `ls` command again, you should see the file WnnnnnnnnAssg1.tar.gz in your directory. This is the file that you need to submit through the **Assignment 1 Submission** link in the moodle web site.

Don't forget to submit your zipped tar file!

Coding Standards

1. Use meaningful names that give the reader a clue as to the purpose of the thing being named.

2. Use named constants instead of repeated use of the same numeric constant.

3. Use comments at the start of the program to identify the purpose of the program, the author and the date written.

4. Use comments at the start of each procedure to describe the purpose of the procedure and the purpose of each parameter to the procedure.

5. Use comments at the start of each section of the program to explain what that part of the program does.

6. Use consistent indentation:

   - The declarations within a procedure must be indented from the **procedure** and **begin** reserved words. The body of the procedure must be indented from the **begin** and **end** reserved words. Example of procedure indentation:

```
procedure DoStuff is
     Count : integer;
     Valid : boolean;
begin
     Count := 0;
     Valid := false;
end DoStuff;
```

   - The statements within the then-part, each elsif-part and the else-part of an if statement must be indented from the reserved words if, elsif, else and end.

```
if Count > 4 and not Valid then
     Result := 0;
     Valid := true:
elsif Count > 0 then
     result := 4;
else
     Valid := false;
end if;
```