Computer Science 1: Starting Computing
CSCI 1300

University of Colorado
Boulder

Computer Science 1: Starting Computing CSCI 1300

Dr. Ioana Fleming
Spring 2019
Lecture 14

University of Colorado
Boulder

Reminders

Submissions:

- Project I: due Sat 2/17 at 6pm
- Homework 5: due 2/24 at 6pm

Disabilities forms

Readings:

- Ch. 4 - this week - loops
- Ch. 2 - strings
- Ch. 6 - next week - arrays

Practicum I: Wed 2/20 at 5:30 pm

Conflict? Contact your TA

Chapter Four: Loops

The Three Loops in C++

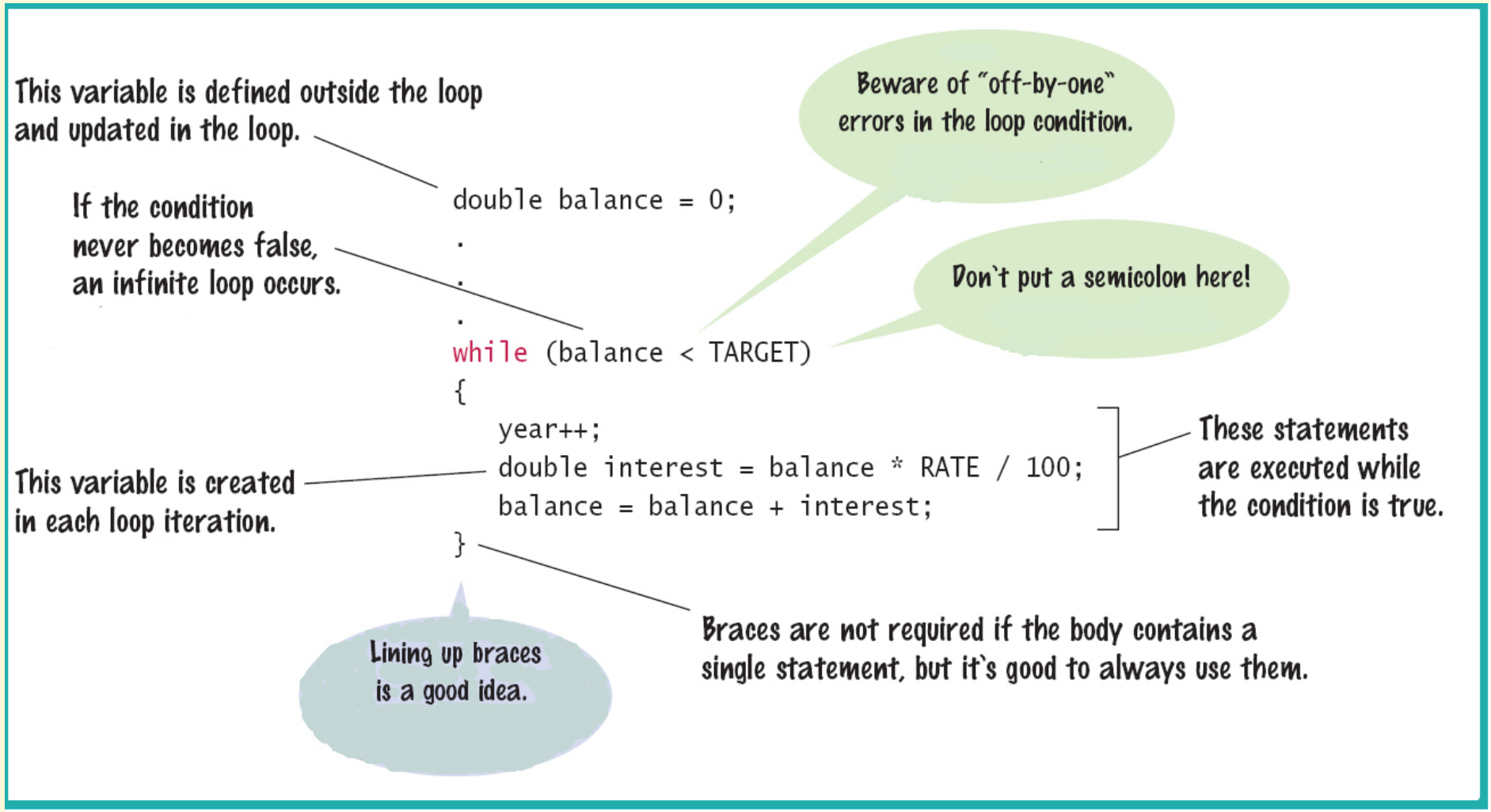
C++ has three looping statements:

while()

for()

do {} while()

The `while` Statement



Topic 4

1. The `while` loop
2. Problem solving: hand-tracing
3. The `for` loop
4. The `do` loop
5. Processing input
6. Problem solving: storyboards
7. Common loop algorithms
8. Nested loops
9. Problem solving: solve a simpler problem first
10. Random numbers and simulations
11. Chapter summary

The `do{ } while()` Loop

The **`while()`** loop's condition test is the first thing that occurs in its execution.

The **`do`** loop (or **`do-while`** loop) has its condition tested only after at least one execution of the statements. The test is at the bottom of the loop:

```
do
{
    statements
}
while (condition);
```

The `do` Loop

This means that the **`do`** loop should be used only when the statements must be executed before there is any knowledge of the condition.

This means that the **`do`** loop will be executed al least once.

This also means that the **`do`** loop is the least used loop.

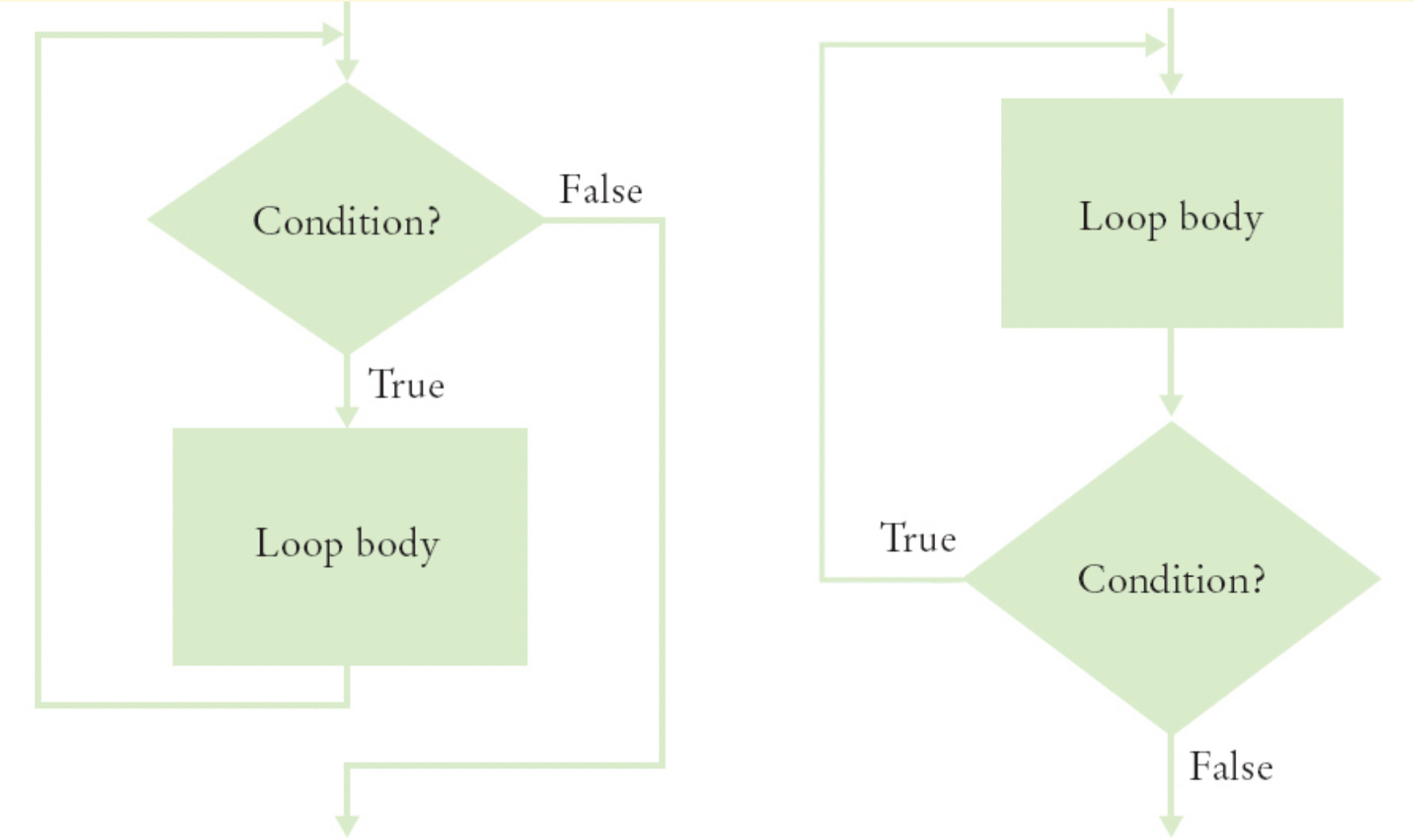
do{ } Loop Code: getting user input Repeatedly

Code to keep asking a user for input until it satisfies a condition, such as non-negative for applying the `sqrt()`:

```
double value;
do
{
 cout << "Enter a number >= 0: ";
 cin >> value;
}
while (value < 0);

cout << "The square root is " << sqrt(value)
<< endl;
```

Flowcharts for the `while` Loop and the `do` Loop



Example of `do...while`

- What output does this loop generate?

```
int j = 1;
do
{
   int value = j * 2;
   j++;
   cout << value << ", ";
} while (j <= 5);
```

Topic 3

1. The `while` loop
2. Problem solving: hand-tracing
3. The `for` loop
4. The `do` loop
5. Processing input
6. Problem solving: storyboards
7. Common loop algorithms
8. Nested loops
9. Problem solving: solve a simpler problem first
10. Random numbers and simulations
11. Chapter summary

The `for` Loop vs. the `while` loop

Often you will need to execute a sequence of statements a given number of times.

You could use a **`while`** loop:

```
counter = 1; // Initialize the counter
while (counter <= 10) // Check the counter
{
   cout << counter << endl;
   counter++; // Update the counter
}
```

The `for` Loop

C++ has a statement custom made ***for*** this sort of processing:

the **`for`** loop.

```
for (counter = 1; counter <= 10; counter++)
{
   cout << counter << endl;
}
```

The `for` Loop Is Better than `while` for Certain Things

Doing something a known number of times or causing a variable to take on a sequence of values is so common, C++ has a statement just for that:

```
for (int count = 1; count <= 10; count++)
{
   cout << count << endl;
}
```

initialization *condition* *statements* *update*

`count = the loop variable`

`for()` loop execution

```
for (initialization; condition; update)
{
   statements;
}
```

- The ***initialization*** is code that happens once, before the check is made, to set up counting how many times the *statements* will happen. The loop variable may be created here, or before the for() statement.
- The ***condition*** is a comparison to test if the loop is done.
  When this test is false, we skip out of the for(), going on to the next statement.
- The ***update*** is code that is executed at the bottom of each iteration of the loop, immediate before re-testing the condition. Usually it is a counter increment or decrement.
- The ***statements*** are repeatedly executed until the condition is false. These also are known as the "loop body".

Scope of the Loop Variable – Define it inside the `for` or earlier?

- The "loop variable" can be defined inside the **`for`** parentheses:

```
int strlen_error(int num)
{
   for(int i=1; num < 10; i++)
   {
         num = num * i;
   }
  return num;
}
```

Scope of the Loop Variable – Define it in the `for` or earlier?

- But then it cannot be used before or after the **`for`** statement – it only exists as part of the **`for`** statement and should not need to be used anywhere else in a program.

```
int strlen_error(int num)
{
   for(int i=1; num < 10; i++)
   {
          num = num * i;
   }
   return i; //syntax error: unknown identifier "i"
}
```

- A **`for`** statement can use variables that were previously defined
  - (In an earlier example, **`counter`** was defined before the loop – so it could be accessed after the loop exited.)

The for Can Count Up or Down

A **for** loop can count down instead of up:

```
for (counter = 10; counter >= 0; counter--)…
```

The increment or decrement need not be in steps of 1:

```
for (cntr = 0; cntr <= 10; cntr +=2)…
```

`for` Loop Examples, Index Values: Table 2

Loop	Values of i	Comment
`for (i = 0; i <= 5; i++)`	0 1 2 3 4 5	Note that the loop is executed 6 times. (See Programming Tip 4.3)
`for (i = 5; i >= 0; i--)`	5 4 3 2 1 0	Use `i--` for decreasing values.
`for (i = 0; i < 9; i = i + 2)`	0 2 4 6 8	Use `i = i + 2` for a step size of 2.
`for (i = 0; i != 9; i += 2)`	0 2 4 6 8 10 ... (infinite loop)	You can use < or <= instead of != to avoid this problem.
`for (i = 1; i <= 20; i = i * 2)`	1 2 4 8 16	You can specify any rule for modifying i, such as doubling it in every step.
`for (i = 0; i < str.length(); i++)`	0 1 2 … until the last valid index of the string str	In the loop body, use the expression `str.substr(i, 1)` to get a `string` containing the i^{th} character.

Solving a Problem with a `for` Statement

- Earlier we determined the number of years it would take to (at least) double our balance.
- Now let's see the interest in action:
  - We want to print the balance of our savings account over a five-year period.

The "…over a five-year period" indicates that a **`for`** loop should be used.

Because we know how many times the statements must be executed, we choose a **`for`** loop.

Solving a Problem with a `for`: Desired Output

The output should be a table with columns for year and balance, something like this:

Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

The Modified Investment Program Using a `for` Loop

ch04/invtable.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const double RATE = 5;
   const double INITIAL_BALANCE = 10000;
   double balance = INITIAL_BALANCE;
   int nyears;
   cout << "Enter number of years: ";
   cin >> nyears;

   cout << fixed << setprecision(2);
   for (int year = 1; year <= nyears; year++)
   {
      balance = balance * (1 + RATE / 100);
      cout << setw(4) << year << setw(10) << balance << endl;
   }

   return 0;
}
```

Infinite Loops Can Occur in `for` Statements

== and **!=** are best avoided
in the check of a **for** statement

for loop example:

```
for (int i = 0; i != 9; i += 2)
   cout << i << " ";
```

The output never ends

0 2 4 6 8 10 12...

Chapter 2: Topic 5

1. Variables
2. Arithmetic
3. Input and output
4. Problem solving: first do it by hand
5. Strings
6. Chapter summary

Strings

- Strings are sequences of characters:

```
"Hello world"
```

- Include the string header, so you can create variables to hold strings:

```
#include <iostream>
#include <string>
using namespace std;
...
string name = "Harry";
         // literal string "Harry" stored
```

String Initializations

- String variables are automatically initialized to the empty string if you don't initialize them:

```
string response;
      // literal string "" stored
            // not garbage!
```

- `""` is called the empty or null string.

Concatenation of Strings

Use the **+** operator to *concatenate* strings;
that is, put them together to yield a longer string.

```
string fname = "Harry";
string lname = "Potter";
string name = fname + lname; //need a space!
cout << name << endl;
name = fname + " " + lname; //got a space
cout << name << endl;
```

The output will be

```
HarryPotter
Harry Potter
```

Common Error – Concatenation of literal strings

```
string greeting = "Hello, " + " World!";
                            // will not compile
```

Literal strings cannot be concatenated. And it's pointless anyway, just do:

```
string greeting = "Hello World!";
```

String Input

You can read a string from the console:

```
cout << "Please enter your name: ";
string name;
cin >> name;
```

When a string is read with the **>>** operator, only one word is placed into the **string** variable.

For example, suppose the user types

```
Harry Potter
```

as the response to the prompt.

Only the string "Harry" is placed into the variable name.

String Input

You can use another input string to read the second word:

```
cout << "Please enter your name: ";
string fname, lname;
cin >> fname >> lname;
 //fname gets Harry, lname gets Potter
```

String Functions

- The **`length`** *member function* yields the number of characters in a string.
- Unlike the **`sqrt`** or **`pow`** function, the **`length`** function is *invoked* with the *dot notation*:

```
string name = "Harry";
int n = name.length();
```

String Data Representation & Character Positions

```
H  e  l  l  o  ,     W  o  r  l  d  !
0  1  2  3  4  5  6  7  8  9  10 11 12
```

- In most computer languages, the starting position 0 means "start at the beginning."

- The first position in a `string` is labeled 0, the second 1, and so on. And don't forget to count the space character after the comma—but the quotation marks are ***not*** stored.

The position number of the last character is always one less than the length of the **`string`**.

`substr` Function

- Once you have a string, you can extract substrings by using the **`substr`** member function.
- **`s.substr(start, length)`**
  returns a **`string`** that is made from the characters in the **`string s`**, starting at character **`start`**, and containing **`length`** characters. (**`start`** and **`length`** are integers)
  - NOTE: the first character has an index of 0, not 1.

```
string greeting = "Hello, World!";
string sub = greeting.substr(0, 2);
   // sub contains "He"
```

Another Example: `substr` Function

```
string greeting = "Hello, World!";
string w = greeting.substr(7, 5);
    // w contains "World" (not the !)
```

- **"World"** is 5 characters long but…
- Why is 7 the position of the **"W"** in **"World"**?
- Why is the **"W"** not @ 8?
- *Because the first character has an index of 0, not 1.*

String Character Positions

```
 H  e  l  l  o  ,     W  o  r  l  d  !
 0  1  2  3  4  5  6  7  8  9 10 11 12
```

```
string greeting = "Hello, World!";
string w = greeting.substr(7);
   // w contains "World!"
```

If you do not specify how many characters to the substr() function, you get all the rest.

String Operations Examples

Statement	Result	Comment
`string str = "C";` `str = str + "++";`	str is set to "C++"	When applied to strings,+ denotes concatenation.
`string str = "C" + "++";`	**Error**	**Error:** You cannot concatenate two string literals.

`cout << "Enter name: ";`
`cin >> name;`
(*User input:* `Harry Morgan`)
	name contains "Harry"	The >> operator places the next word into the string variable.

`cout << "Enter name: ";`
`cin >> name >> last_name;`
(*User input:* `Harry Morgan`)
	name contains "Harry", last_name contains "Morgan"	Use multiple >> operators to read more than one word.

`string greeting = "H & S";` `int n = greeting.length();`	n is set to 5	Each space counts as one character.
`string str = "Sally";` `string str2 = str.substr(1, 3);`	str2 is set to "all"	Extracts the substring of length 3 starting at position 1. (The initial position is 0.)
`string str = "Sally";` `string str2 = str.substr(1);`	str2 is set to "ally"	If you omit the length, all characters from the position until the end are included.
`string a = str.substr(0, 1);`	a is set to the initial letter in str	Extracts the substring of length 1 starting at position 0.
`string b = str.substr(str.length() - 1);`	b is set to the last letter in str	The last letter has position str.length() - 1. We need not specify the length.

String Functions, Complete Program Example

ch02/initials.cpp

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   cout << "Enter your first name: ";
   string first;
   cin >> first;
   cout << "Enter your significant other's first name: ";
   string second;
   cin >> second;
   string initials = first.substr(0, 1)
      + "&" + second.substr(0, 1);
   cout << initials << endl;

   return 0;
}
```

Representing Characters: Unicode

- Printable characters in a `string` are stored as bits in a computer, just like `int` and `double` variables
- The bit patterns are standardized:
  - ASCII (American Standard Code for Information Interchange) is 7 bits long, specifying $2^7 = 128$ codes:
    - 26 uppercase letters A through Z + 26 lowercase letters a through z
    - 10 digits
    - 32 typographical symbols such as +, -, ', \...
    - 34 control characters such as space, newline, and 32 others for controlling printers and other devices.
  - Unicode, which has replaced ASCII in most cases, is 21 bits
    - superset of ASCII; the first 128 codes match
    - The extra bits allow many more characters ($2^{21} > 2x10^6$), required for worldwide languages
    - About 136,000 characters have been assigned so far
    - UTF-8 is the 8-bit subset of Unicode, and UTF-16 is 16-bit, often used by websites and compilers

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

University of Colorado
Boulder