18. Characters and C-strings

Objectives

- Declare character variables with or without initialization
- Output and input character values
- Use operators for character type
- Understand the relationship between characters and ASCII decimal values
- Declare C-string variables with or without initialization
- Use C-string functions

Containers

We're already talked about arrays (arrays of ints, arrays of doubles, etc.) Usually we use all the slots in the arrays. However an array can be used to model **containers**. What do I mean by that? For instance suppose you want to write a program to contain student IDs of current students who are at least 7 feet tall. (Not a big list I suppose …) In fall 2009 the list might contain:

3145673, 4135778, 2356317

in spring 2009 it might be

3145673, 4135778, 2356317, 5678246

and in fall 2010 it might be

2356317, 5678246

As you can see the number of things in the list changes.

An array can be used to model a container. It can be done in two ways (at least!) You can keep an extra variable to indicate the number of things modeled by the array. For instance try this program that prompts the user for ids to be added to the id array and terminates when the user enters -1.

```
int length = 0;
int id[10];
while (1)
{
    if (length == 10)
    {
        std::cout << "oops ... I'm full!\n";
        break;
    }
    int x;
    std::cin >> x;
    if (x == -1) break;
    id[length] = x;
    length++;

    for (int i = 0; i < length; i++)
    {
        std::cout << id[i] << ' ';
    }
    std::cout << '\n';
}
```

In this case **`length`** is used to indicate that the actual student ids are

```
id[0], id[1], …, id[length-1]
```

and the rest

```
id[length], id[length+1], …, id[999]
```

are unused.

A second way to model a container is to use a special **sentinel** value to denote **"end-of-data"**. Try this program that does the same thing as above except that I'm using -9999 to mark the end-of-data.

```
int id[10] = {-9999};
while (1)
{
    // find length
    int length = 0;
    for (int i = 0; i < 10; i++)
    {
        if (id[i] == -9999)
        {
            length = i;
            break;
        }
    }
    if (length == 9)
    {
        std::cout << "oops ... I'm full!\n";
        break;
    }

    int x;
    std::cin >> x;
    if (x == -1) break;
    id[length] = x;
    id[length + 1] = -9999;

    // print values in container
    int i = 0;
    while (id[i] != -9999)
    {
        std::cout << id[i] << ' ';
        i++;
    }
    std::cout << '\n';
}
```

Note that the array begins with

-9999, …

(the … after the -9999) If I entered 42, the array becomes

    42, -9999, …

And if I entered 135 as the next number, the array becomes

    42, 135, -9999, …

Note that when using a sentinel to implement a container

- When adding the end of the container you always need to search for the sentinel value (more or less to compute the length of the container)
- The number of values you can put into the container is always **one less than the actual size** of the container. Why? Because the sentinel value takes up on of the spots in the array!

We'll be talking about C-strings in this set of notes. You'll see that the C-string uses the second method to describe characters in a string.

Characters

Recall that this is a character:

```
'A'
```

That's nothing new. To declare a character variable you do this:

```
    char c;
```

You can of course initialize it too:

```
    char c = 'A';
```

And of course you already know that you can output and input characters:

```
std::cout << 'A' << std::endl;

char c = 'A';
std::cout << c << std::endl;

std::cin >> c;
std::cout << c << std::endl;
```

Another thing you already know is that you can compare characters:

```
char c = 'A';
std::cout << "gimme an A: ";
std::cin >> c;
if (c == 'A')
{
    std::cout << "thanks!";
}
else
{
    std::cout << "r u the troublemaker-type?";
}
```

ASCII: Characters and integers

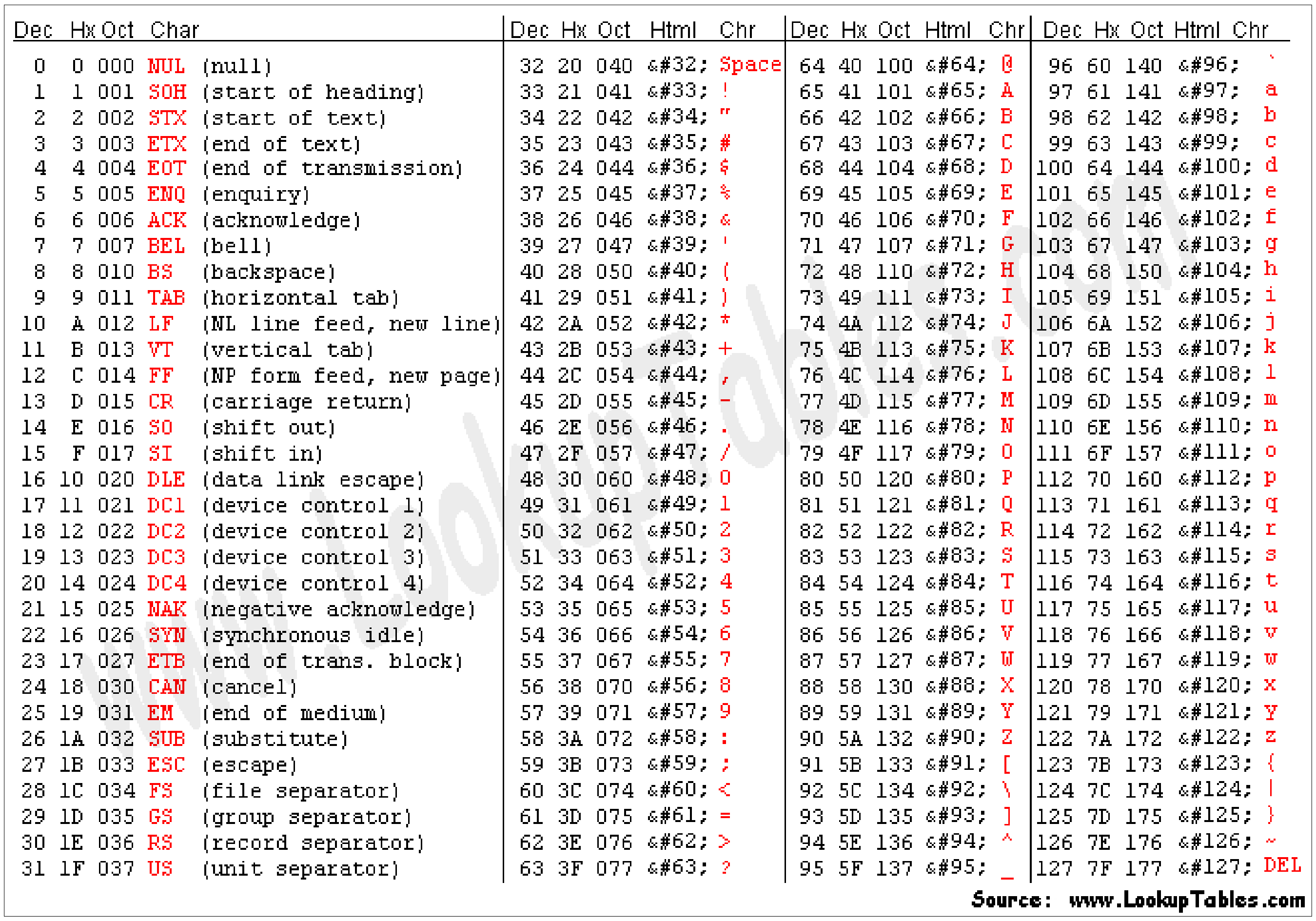
One important thing you should know is that the character type is actually an integer type. You can typecast between the two. Try this:

```
std::cout << (int) 'A' << std::endl;
std::cout << int('A') << std::endl;
char c = 'A';
std::cout << (int) c << std::endl;
std::cout << int(c) << std::endl;
```

As you can see the integer value corresponding to character `'A'` is 65. Now try this:

```
std::cout << (char) 65 << std::endl;
std::cout << char(65) << std::endl;
```

The translation between characters and integer value is provided by the so-called ASCII table. You can google the web to read more about the ASCII standard and its history. You can also go straight to wikipedia.com and search for “ascii”. Here's one such table. Look for 'A' and you see that the integer value corresponding to 'A' is 65 (look under the “Dec” column - “Dec” means the integer value is in base 10 **dec**imal notation.)



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Exercise. What is the ASCII value of 'Z'? (You can of course use the web ... but it's easier just to write a program).

Now with the ASCII value of 'Z', fill in the blank and run this program ... WAIT!!! ... can you figure out what it does BEFORE running the program?

```
for (int i = 65; i <= _____; ++i)
{
    std::cout << (char) i << std::endl;
}
```

In fact since characters can be automatically typecasted to integers you can do this too:

```
for (int i = 'A'; i <= 'Z'; ++i)
{
    std::cout << (char) i << std::endl;
}
```

Exercise. What is the ASCII value of 'Z' and '5'? (Use the above ASCII table.) Is the following true or false?

```
      'Z' < '5'
```

Now verify your guess by running this:

```
std::cout << ('Z' < '5') << std::endl;
```

YET another way to do this is to realize that there are arithmetic operators on the character type. First try this:

```
std::cout << 'A' + 1 << std::endl;
```

This tells you that 'A' is automatically typecasted to it's ASCII value. Now you can do this:

```
std::cout << (char)('A' + 1) << std::endl;
```

Finally ... run this:

```
for (char c = 'A'; c <= 'Z'; ++c)
{
    std::cout << c << std::endl;
}
```

This tells us that besides == and !=, the character type has the boolean operators <, <=, >, >=. Basically these comparison operators compare the ASCII value of the character. And of course you have arithmetic operators since the character type is an integer type.

So you see ... characters are actually numbers ... at least in the computer.

Digit Character to Integer Value

Here's a neat trick.

Suppose you want to "translate" the character `'0'` to integer `0`, character `'1'` to integer `1`, ..., character `'9'` to `9"`. One way is to do this: Suppose `c` is a character with character `'0'` or `'1'` or … or `'9'`, and you want `i` which is an `int` to be `0` if `c` is `'0'`, `1` if `c` is `'1'`, etc. You can do this:

```
char c = ' ';
std::cin >> c;
int i = 0;
switch (c)
{
    case '0': i = 0;
              break;
    case '1': i = 1;
              break;
    case '2': i = 2;
              break;
    case '3': i = 3;
              break;
    case '4': i = 4;
              break;
    case '5': i = 5;
              break;
    case '6': i = 6;
              break;
    case '7': i = 7;
              break;
    case '8': i = 8;
              break;
    case '9': i = 9;
              break;
}
std::cout << i << std::endl;
```

Or you can also use a stack of if-else. Make sure you run this program.

That's great. It works. But it shows you're a newbie!!! Here's the smart way to do it. Notice that the ASCII table for '0', '1', ..., '9' is this:

character	ASCII integer value	value that you want
'0'	48	0
'1'	49	1
'2'	50	2
'3'	51	3
'4'	52	4
'5'	53	5
'6'	54	6
'7'	55	7
'8'	56	8
'9'	57	9

Do you see a pattern? (SPOILERS AHEAD)

The point is that the character '0', ..., '9' appear in order in the ASCII table. Look at the table again:

character	ASCII integer value	translated value
'0'	48	0 = 48 – 48
'1'	49	1 = 49 – 48
'2'	50	2 = 50 – 48
'3'	51	3 = 51 – 48
'4'	52	4 = 52 – 48
'5'	53	5 = 53 – 48
'6'	54	6 = 54 – 48
'7'	55	7 = 55 – 48
'8'	56	8 = 56 – 48
'9'	57	9 = 57 – 48

which is the same as

character	ASCII integer value	translated value
'0'	48	0 = int('0' – '0')
'1'	49	1 = int('1' – '0')
'2'	50	2 = int('2' – '0')
'3'	51	3 = int('3' – '0')
'4'	52	4 = int('4' – '0')
'5'	53	5 = int('5' – '0')
'6'	54	6 = int('6' – '0')
'7'	55	7 = int('7' – '0')
'8'	56	8 = int('8' – '0')
'9'	57	9 = int('9' – '0')

In all cases, if c is a character '0',...,'9' the integer value you want is ...

c – '0'

That's all!!! In other words, this program:

```
char c = ' ';
std::cin >> c;
int i = 0;
switch (c)
{
    case '0': i = 0;
              break;
    case '1': i = 1;
              break;
    case '2': i = 2;
              break;
    case '3': i = 3;
              break;
    case '4': i = 4;
              break;
    case '5': i = 5;
              break;
    case '6': i = 6;
              break;
    case '7': i = 7;
              break;
    case '8': i = 8;
              break;
    case '9': i = 9;
              break;
}
std::cout << i << std::endl;
```

can be **rewritten** as:

```
char c = ' ';
std::cin >> c;
int i = c - '0';
std::cout << i << std::endl;
```

I don't know about you. But I prefer the second version!!! Make sure you run this program.

Exercise. Write a program that prompts the user for a, b, c, …, z (i.e. lowercase letters) and prints 0, 1, 2, …, 25 respectively.

Exercise. Write a program that prompts the user for a, b, c, …, z, A, B, C, …, Z (i.e. lower and uppercase letters) and prints 0 for a or A, 1 for b or B, 2 for c or C, …, 25 for z or Z respectively.

C-strings

A **C-string** is nothing more than an array of characters. The only thing different between a C-string and an array of integers (say) is that the **special escape character '\0'** (the 0 is the **number** 0, not the letter O) is used to mark the **end-of-string**. This is a special character just like '\n' or '\t' because '\0', '\n', '\t' are not "visible" characters.

Try to print '\0' and you won't see a thing:

```
std::cout << '\0' << std::endl;
```

(There are in fact lots of invisible characters.)

The point of the end-of-string marker '\0', is to end some string processing. This is what I mean. Try this:

```
char spam[] = {'a', 'b', 'c', '\0', 'd', 'e'};
std::cout << spam << std::endl;
```

In this case the "processing" is the print statement:

```
std::cout << spam << std::endl;
```

The `'\0'` basically tells the print statement that printing should stop after printing the third character `'c'`. The output looks like this:

```
abc
```

Note another thing ... **You can actually print a character array!!!** Remember that you cannot do that for an array of integers, doubles, or booleans. You have to write a for-loop in those case. As a reminder try this:

```
int x[] = {1, 2, 3, 4, 5};
std::cout << x << std::endl;
```

Let's go back to the above example:

```
char spam[] = {'a', 'b', 'c', '\0', 'd', 'e'};
std::cout << spam << std::endl;
```

Note that it does NOT mean that the size of the array is 3; it is true that three characters are printed. There are actually 6 characters in the array:

```
'a', 'b', 'c', '\0', 'd', 'e'
```

So you need to distinguish the difference between the ***size*** of the string `spam` and its ***length*** which is defined to be the number of characters up to, but not including, the first `'\0'`.

Informally when we think of

```
char spam[] = {'a', 'b', 'c', '\0', 'd', 'e'};
```

as a string, we say that it is made up of `'a', 'b', 'c'`. But **as a character array** is made up of six characters. Make sure you remember that.

You can think of the character '\0' in the array

```
char spam[] = {'a', 'b', 'c', '\0', 'd', 'e'};
```

as a sentinel value to mark the end of data for the string represented as an array. The character '\0' is also called the **null character**. This is the reason why C-strings are also called **null-terminated strings**.

Exercise. What are the lengths and sizes of the following variables?

```
char ham[] = {'3', '.', '\0', '1', '4'};
char eggs[] = {'\0', 'b', '\0', 'a'};
```

You can verify your answers by counting the number of characters printed:

```
std::cout << ham << std::endl;
std::cout << eggs << std::endl;
```

There is a shorthand for `{'a', 'b', 'c', '\0'}`. Try this:

```
char spam[] = "abc"
std::cout << spam << std::endl;
```

Note in this case that `spam` has size 4 (there are 4 characters in the array) and length 3 (there are 3 characters before the first '\0'); `"abc"` is

`{'a', 'b', 'c', '\0'}`.

Exercise. What is the length and size of the following variable?

```
char spam[100] = "ab\0cd";
```

Exercise. What is the length and size of the following variable?

```
char spam[] = "ab\0cd";
```

Exercise. What are the length and size of the following variable?

```
char spam[100] = "";
```

Of course you can do this:

```
std::cout << "abc\0de" << std::endl;
```

You have already seen this ...

```
std::cout << "hello world" << std::endl;
```

during the first week of class. Now you know that `"hello world"` is an array of characters. It's just the character array made up of the following characters:

```
      'h','e','l','l','o',' ','w','o','r','l','d','\0'
```

(Don't forget the '\0'!!!)

Exercise. Can you do the following?

```
std::cout << {'a', 'b', 'c'} << std::endl;
```

If you have a long string you can break it up like this:

```
char foo[] = "abc"
             "def";
std::cout << foo << std::endl;
```

Basically C++ will join it up for you so that the above code is the same as this:

```
char foo[] = "abcdef";
std::cout << foo << std::endl;
```

Exercise. Can you do this:

```
std::cout << "abc" "def" << std::endl;
```

Can you do the same for integers?

```
std::cout << 123 456 << std::endl;
```

Exercise. You now know that although you cannot print an array of integers with `std::cout` (you need to write a for-loop), you can print an array of characters (it will only print up to `'\0'`). What about input? Try this:

```
char s[100];
std::cin >> s; // type in abcde
std::cout << s << std::endl;
```

Not too shocking right?

Exercise. Run the above program again and enter this as input:

```
      hello world
```

What do you see?

The problem with `std::cin` is that is that `std::cin` cuts up inputs at spaces, tabs, or newline (i.e. whitespaces). So for the above exercise, C++ will only give the string `"hello"` to variable s. If you do want spaces to go into your string variable, then you need to do something else for input. Check out a later section on the `getline()` function.

The `strlen()` function

C-strings are so useful that several functions are provided with your compiler so that you can work effectively with them. Depending on your compiler, you might need to do this when you want to use these C-string functions:

```
#include <iostream>
#include <cstring>

... YOUR PROGRAM ...
```

If your compiler is really old then you might need to do this instead:

```
#include <iostream>
#include "string.h"

... YOUR PROGRAM ...
```

Here's the first function ...

The function `strlen()` returns the **length** of the string parameter. This is called the **string length function**. Try this:

```
std::cout << strlen("abc") << std::endl;
std::cout << strlen("a\0c") << std::endl;
std::cout << strlen("") << std::endl;

char s[] = "abc";
std::cout << strlen(s) << std::endl;

char t[] = "ab\0c";
std::cout << strlen(t) << std::endl;

char u[100] = "";
std::cout << strlen(u) << std::endl;
```

By the way, what if your compiler does not have the `strlen()` function? Well the code to perform the same thing as the `strlen()` function is very simple. You just loop through the character array until you see `'\0'`. You have a counter that counts the number of characters scanned until `'\0'`, not including the `'\0'` character. Here you go:

```
char x[100] = "abc";

int len = 0;
while (x[len] != '\0')
{
    len++;
}
std::cout << len << '\n';
```

Run the program. Change it to this and run it again:

```
char x[100] = "hello world";
```

```
int len = 0;
while (x[len] != '\0')
{
    len++;
}
std::cout << len << '\n';
```

There's actually a "problem" with the `strlen()` function. You see ... if a person accidentally calls the `strlen()` function with a character array that does ***not*** contain a `'\0'`, the function can actually go beyond the array bound. This might crash the program. Here's an example:

```
char x[] = {'a', 'b', 'c'};
std::cout << strlen(x) << std::endl;
```

This is what we call a **buffer overflow**: Your string length function will access a value in the array `x` that is actually outside `x`. We'll come back to this issue later.

Other C-string functions

In fact there are MANY standard string functions that come with all C compilers. Here are some of the most basic ones. Experiment with this code and figure out what the string functions do:

```
char s[100] = "aaa";
char t[100] = "bbb";
char u[100];

// string copy function
// overwrite the string in u with the string in
// s
strcpy(u, s);
std::cout << u << std::endl;

// string concatenation function
// put the string in t to the end of the string
// in u
strcat(u, t);
std::cout << u << std::endl;

// WARNING: You cannot do strcat(u, u), i.e., the
// two arguments of the function must be different.

// string comparison function
// 0 is returned when the strings are
// the same. Otherwise a nonzero value is
// returned.
int i = strcmp(u, s);
std::cout << i << std::endl;
```

You might wonder what nonzero value is returned by `strcmp()` when the strings are different.

```
std::cout << strcmp("a", "b") << std::endl;
std::cout << strcmp("a", "c") << std::endl;
std::cout << strcmp("a", "d") << std::endl;
std::cout << strcmp("a", "z") << std::endl;
std::cout << strcmp("a", "!") << std::endl;
std::cout << strcmp("a", "ba") << std::endl;

std::cout << strcmp("b", "a") << std::endl;
std::cout << strcmp("c", "a") << std::endl;
std::cout << strcmp("d", "a") << std::endl;
std::cout << strcmp("ba", "a") << std::endl;
```

See it yet? (Hint: Check an ASCII table.)

There are other C-string functions but you will need to know a little more in order to understand how to use them.

The `strnlen()`, `strncmp()`, `strncat()` functions

All the string functions we talked about run into problems if the character arrays passed into these function do not contain '\0' resulting in buffer overflow.

In fact this is one scenario of the so-called **buffer overflow attack**, a technique used in many virus software (malware).

Accompanying these functions are similar functions that allow you to specify a stopping point. For instance in the case of `strlen()` (the string length function) there is a function called

```
strnlen()
```

Besides passing in a string, you also pass in a maximum possible length for the string.

```
char s[100] = ""; // s[0] = '\0';
std::cin >> s;
std::cout << strnlen(s, 99) << std::endl;
```

Note that I pass in 99 and not 100. If `s` has a **size of 100** the **maximum possible length is 99** since in this case the **last character is used by ~~by~~ `'\0'`**. Therefore `s` (of size 100) has a maximum length of 99.

Nsync Bye Bye Bye
(Cole)

Exercise. What is the output of this code:

The accompanying safer functions for `strcmp()` and `strcat()` are `strncmp()` and `strncat()`. So we have the following ***pairs*** of functions:

```
strlen()          strnlen()
strcmp()          strncmp()
strcat()          strncat()
```

Try this program:

```
char s[100] = "aaa";
char t[100] = "bbb";
char u[100];

// string copy function
// In this case at most 100 chars are copied
// from s to u
strncpy(u, s, 100);
```

```
std::cout << u << std::endl;

// string concatenation function
// In this case at most 100 chars are
// concatenated from t to u
strncat(u, t, 100);
std::cout << u << std::endl;

// string comparison function
// In this case at most 100 characters are
// compared.
int i = strncmp(u, s, 100);
std::cout << i << std::endl;
```

The only problem in the above example is the call to `strncat()` function. You see if `u` already has a string of length 10, then only 90 characters are left for concatenation. If `t` has length greater than that, the program might crash. Therefore it's safe to write this:

```
// string concatenation function
// In this case at most 100 chars are
// concatenated from t to u
strncat(u, t, 100 - strnlen(u, 100));
std::cout << u << std::endl;
```

But in this case some character might not be copied from `t` to `u`. Depending on what your program is supposed to do, you might want to print an error message and halt the program.

The `std::cin.getline()` function

Recall that there is a problem with input of strings ...

`std::cin` cuts up inputs at spaces, tabs, or newline (i.e. whitespaces). So if you run this code:

```
char s[100];
std::cin >> s;
std::cout << s << std::endl;
```

with this input

```
to be or not to be
```

you will find that the output is

```
to
```

In other words C++ chops up the input at spaces, tabs and newlines. So `s` is only assigned the string `"to"`.

Now … what ever happened to the rest of the input:

```
be or not to be
```

?!? Try this:

```
char s[100];
char t[100];
char u[100];

std::cin >> s >> t >> u;
std::cout << s << '\n' << t << '\n' << u << '\n';
```

with input

```
i'm zaphad beeblebrox
```

Get it?

Exercise. What if you entered 5 spaces at the ***beginning*** of the input like this

```
     i'm zaphad beeblebrox
```

What is the value of `s` in this case?

So how do you handle the problem of spaces and tabs? What if you really want spaces and tabs to go into a string? Then you have to use another function. Try running this:

```
char s[100];
std::cin.getline(s, 100);
std::cout << s << std::endl;
```

with this input:

```
to be or not to be
```

Note that the `std::cin.getline()` function also allows you to specify the size of `s`. In the above example, C++ will only read in at most 99 characters, reserving the last for '\0'. This prevents buffer overflow.

The `std::cin.getline()` function will assign all the characters up to but not including `'\n'` (when you press the enter key you are including a `'\n'` in your input. In other words `getline()` cuts up input at the `'\n'` character.

The `std::cin.getline()` function can actually do a lot more. You can get it to cut up input at any character you specify. For instance

```
      std::cin.getline(s, 100, '\t');
```

will cut up input at `'\t'` (tab) characters. (The `'\t'` in this case is sometimes called the stop character.)

Try this

```
char s[10];
char t[10];
char u[10];

std::cin.getline(s, 10, ',');
std::cin.getline(t, 10, ',');
std::cin.getline(u, 10, ',');

std::cout << '[' << s << "]\n"
          << '[' << t << "]\n"
          << '[' << u << "]\n" ;
```

with input

```
aaa,  bbb  ,c c c,
```

Exercise. What happens when you run the above program with this input

```
0123456789,b,c,
```

`std::cin` and `std::cin.getline()`

The **extremely important thing** to remember is that `std::cin` does not work well with `std::cin.getline()`.

Exercise. Try to mix `std::cin` and `std::cin.getline()` like this:

```
char s[100];
int i;

std::cin >> i;
std::cout << i << std::endl;

std::cin.getline(s, 100);
std::cout << s << std::endl;
```

Run the program and see what happens.

Type conversion: C-strings to numerics

We've been using the `std::cin` for inputs and it works for input of integer values, double values, C-string values, etc. We saw above that the problem is that `std::cin` cuts up input at whitespaces. The `std::cin.getline()` does work by really taking in everything as input. But the problem is that it works for strings.

```
#include <iostream>
#include <cstdlib>

int main()
{
  char s[100] = "123";
  char t[100] = "12.3";

  int i = strtol(s, NULL, 10);
  double d = strtod(t, NULL);

  std::cout << i << '\n' << d << '\n';
}
```

The functions above are

`strtol()`	string-to-long-integer
`strtod()`	string-to-double

The `strtol()` converts the string to a **`long int`** whose largest possible value might be larger than your usual **`int`**.

I'll explain the `NULL` business later. Right now I just want to get the usage of the functions to you now so that you can use it.

Type conversion: numerics to C-strings

Of course you also need to know how to convert an int or a double to a C-string. There are several ways to do this.

Here's the first method in C-style (i.e. using a C function)

```
#include <cstdio> // or #include <stdio.h>

...

    char c[100];
    int i = 42;
    double d = 3.14;

    sprintf(c, "%d", i);
    std::cout << c << '\n';

    sprintf(c, "%f", d);
    std::cout << c << '\n';
```

The second method uses C++ features:

```
#include <sstream>

...

    char c[100];
    int i = 123;
    double d = 3.14;

    std::stringstream out;

    out << i;
    strcpy(c, out.str().c_str());
    std::cout << c << '\n';

    out << d;
    strcpy(c, out.str().c_str());
    std::cout << c << '\n';
```

Exercise

Exercise. Write a program that prompts the user for a string and prints the character that occurs most frequently in the string.

Exercise. Write a program that prompts the user for a string and checks if the string is a palindrome. The check should ignore non-letters and the case of letters. For instance “madam, I'm Adam” is a palindrome.

Exercise. Write a program that prompts the user for a string, that allows user to perform a substitution of characters. For instance, if the user enters “hello world” and then request to have l replaced by r.

```
input: hello world
source: l
target: r

hello world
l->r
--rr- ---r-

source: d
target: e
hello world
l->r, d->e
--rr- ---re

source: l
target: o
l->o, d->e
hello world
--oo- ---oe

source: w
target: o
*** invalid target: o is already a target
target: i
l->o, d->e, w->i
hello world
--oo- i-—oe

source: +
```

The program quits when the user enters + for source. This simple program can be used to help you solve a type of encrypted text called substitution cipher..

Exercise. Write a program that prompts the user for two strings and checks if the second string occurs as a substring of the first. If so the index where the second occurs in the first is printed. Otherwise -1 is printed.

Exercise. Write a program that accepts a string from the user and a window column size and prints the string in the console window with the given size. For instance if the user entered “it was a dark and stormy night …” and a size of 10, then the output is

```
it was a
dark and
stormy
night ...
```

Note that at most 10 characters from the string is printed per row. If the user entered 15 for the size then the output is

```
it was a dark
and stormy
night ...
```

Exercise. Write a program accepts a string that is an integer expression, interprets the expression, perform the appropriate operations and prints the results. For instance is the user enters the string “1 + 2”, the program prints 3. If the user enters “1 – 2 + 3 * 4” the program prints 11. The program should handle +, -, *, /, % and integer values.