ALiCE:

ARTIFICIAL LIFE, CULTURE & EVOLUTION

ISIS 170 / CompSci 107 / VMS 172

Fall 2012

C++ & Windows API

Language Guide

C++ Language Basics

C++ Statements		
=	Assigns what is on the right side of the = to what is on the left. If **a** was 12, then **a** now becomes 25.	`a = a + 13;`
==	Asks whether what is on the right of the = is the same as what is on the left. If **a** is 25, then the statement is evaluated as **false**.	`(a == 56)`
( )	Groups code to clear up any ambiguities in the order in which they are evaluated. Also encloses the parameters of a function call.	`(3+6) * (-2-a) / 5` `random(6);`
{}	Groups larger blocks of code.	`for (i = 0; i < 10; i++) { sum = sum + i; }`
[]	An array subscript, the elements within a list or table.	`x[23] = 6;`
;	Terminates every statement.	`x = x * x;`

Shorthand Operators				
++	Increment	`i++;`	is the same as	`i = i + 1;`
--	Decrement	`i--;`	is the same as	`i = i - 1;`
+=		`b += 100;`	is the same as	`b = b + 100;`

Operators in order of Precedence	
( )	Function call
[ ]	Array subscript
->	Indirect component selector
.	Direct component selector
!	Logical negation
+	Plus
-	Minus
( )	Expression parentheses
*	Multiply
/	Divide
%	Remainder (modulus divide)
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to
&&	Logical and
||	Logical or
=	Assignment

-=		`c -= 10;`	is the same as	`c = c - 10;`

switch statements enable multiple decision branches:

```
switch (year) {
   case 1: {
      Edit1->Text = "You are a Freshman.";
      break;
   }
   case 2: {
      Edit1->Text = "You are a Sophomore.";
      break;
   }
   case 3: {
      Edit1->Text = "You are a Junior.";
      break;
   }
   case 4: {
      Edit1->Text = "You are a Senior.";
      break;
   }
   default: {
      Edit1->Text = "Are you a Grad?";
   }
}
```

Note: The "break;" statements transfer control below and outside the scope of the "switch."

if / else commands execute blocks of code only if the value in parentheses is true:

A single block of code may be executed:

```
if (gpa > 3.25) {
   Edit1->Text = "That's better than good.";
}
```

Any or all blocks of code may be executed:

```
if (zipCode == 90290) {
   Edit1->Text = "You live in Topanga.";
}
if (zipCode == 27708) {
   Edit1->Text = "You live near Duke.";
}
if (zipCode == 90077) {
   Edit1->Text = "You live in Beverly Glen.";
}
if (zipCode == 27278) {
   Edit1->Text = "You live in Hillsborough.";
}
if (sex == 0) {
   Edit2->Text = "You are female.";
}
```

```
if (age > 30) {
   Edit3->Text = "You can't be trusted.";
}
```

Only one of the two blocks of code will be executed:

```
if (hunger > 50) {
   searchForFood();
}
else {
   searchForMate();
}
```

Only one of the blocks of code will be executed:

```
if (hungerForChocolate > 10) {
   goForChocolate();
}
else if (thirstForWater > 50) {
   goForWater();
}
else if (wakefulness < 70) {
   goForCoffee();
}
else if (thirstForWater < 30) {
   goForWater();
}
else {
   goHome();
}
```

`while` loops test a condition before entering the code block.
The code is never executed unless the condition is met.

```
while (runWayClear == true) {
   allowAircraftToLand();
}
```

You will never allow an aircraft to land as long as the runway is not clear.

`do-while` loops test a condition after leaving the code block.
The code is always executed once.

```
do {
   dance();
}
while (musicStopped == false);
```

You will always have one dance, even if the music has stopped.

`for` loops take three parameters enabling you to initialize, terminate and increment the loop counter:
Assuming you wished to do something with each agent from id 23 to 79,
setting `first` to 23 and `last` to 79 would do the job.
If you wanted every odd numbered agent from 23 to 79, then the last parameter should be `id = id`

+ 2:

```
for (int id = first; id <= last; id = id + 1) {
   // these agents increase in age
   agent[id].age ++;
}
```

nested **for** loops
allow you to cycle through all the cells in a 2d grid.

```
for (row = 0; row < 500; row ++) {
   for (column = 0; column < 500; column ++) {
      // make resource grow
      resource[row][column] ++;
   }
}
// grow food in every cell in the world
```

The **break** statement causes an immediate exit from a switch or a loop:

```
for (student = 0; student < 500; student++) {
   if (score[student] == 99) break;
}
// the first student with a "99" score
// or student 500 will be shown

Edit1->Text = student;
```

The **goto** statement may be used to break out of a double or more deeply nested loop:

```
for (row = 0; row < 500; row ++) {
   for (column = 0; column < 500; column ++) {
      // look for row and column containing 77
      if (77 == cell[row][column]) {
         goto exit;
      }
   }
}
exit:
// we have now found the row and column
// of the first cell containing "77"
```

The **continue** statement returns you to the loop's beginning,
skipping statements that follow it and incrementing the loop counter:

```
for (student = 0; student < 500; student++) {
   if (score[student] < 50) continue;
   score[student] = score[student] * 1.2;
   bonus[student] = true;
   bonusesAwarded++;
   notify(student);
   notify(professor);
}
// students with scores of "50" or
// higher receive bonus points
```

The **return** statement causes an immediate exit from a function,
returning the value of "sum" as an integer :

```
int addTwoNumbers (int a, int b) {
   int sum = a + b;
   return sum;
}
```

C++ Functions

Think of functions as agents: they sense, think and act.

By convention, any functions that you declare and define should be named with a lower-case initial letter.

Functions are the jobs you need to coordinate in order to get a larger project done. As a programmer, think of yourself as the manager of an enterprise, the general of an army or the director of a movie. We'll pursue the cinematic metaphor. Imagine yourself as the director of a movie. You are in charge of everyone. You coordinate the work of the actors, cameramen, focus pullers, extras, carpenters, electricians, set decorators, prop handlers and sound recorders. Each has a job to do, and each may delegate smaller jobs to those beneath them. But you are in command of the big picture. It is your job to decide what is to be done, how to do it and who to assign to each task.

When you write an application, you are the director of a complex set of functions, your agents to whom you have assigned these specific tasks. Think of C++ functions as your agents. It is up to you to specify what each wil do and how. As as the creator and programmer of an artificial world, it is your responsibility to identify the jobs that need to be done, to describe each job or process as a C++ function, and to organize these functions and their relationships to one another in order to accomplish the task at hand. Think of your application as an organized group of functions; think of it as the social organization of your crew of employees:

- Each one has a name (**functionName**).
- Each one has one or more jobs to do **(the body of the function)**.
- Each one may, or may not, when it is called, receive further data to help it do its job (**its parameters**).
- Each may, or may not, reply directly to whoever called it (**its returnValue**).

For further flexibility, any function, may call any other function, just as any crew member may call upon another for assistance. You must organize all of this. You must describe the "chain of command:"

The Director calls, "Lights 3." (A function call with one parameter and one return value.)

- The lighting person throws the switch on one flood and two spots, and reports back when this is done.

The Director calls, "Camera 2, 5." (A function call with two parameters and one return value.)

- Cameraman 2 starts his camera rolling...
- Cameraman 5 starts his camer rolling...
- They respond that they have done so...

The Director calls, "Action." (A function call with no parameters and no return value.)

- The actors begin to follow their scripts, the key grip pushes the camerman in his dolly along the track, the sound person moves her microphones overhead...
- And each person cues others to begin their jobs: extras swarm around the lead actors, assistants keep the cables from tangling, the continuity person takes notes...
- No one responds directly to the Director...

These calls to action are called **function calls** in programming.

An Abstract Overview

A function may accept **parameters**, do something in its **body**, and directly **return only one value**.
However, it may also change the values of any number of global variables and the properties of any number of components.

```
// An abstract description of a function:
returnType functionName (parameters) {
   // do this (the function body)
   return returnValue;
}
```

It might be helpful to think of a **function** as an **agent** with a name, who can sense, think, and act:

```
// Thinking of a function as an agent:
reply functionName (instructions) {
   // do something (thought and action)
   return reply;
}
```

Or you may wish to think of a **function** as **black box** with a name, which has **inputs** and one **output**:

```
// Thinking of a function as a black box:
(output) functionName (inputs) {
   // do something (action and output)
   return returnValue;
}
```

You **call** a function by calling its **name** followed by the parameters, if there are any, in **parentheses**.

In summary:

- Each function has a name you call it by (i.e. its **functionName**).
- You can give it additional data when you call if you wish to do so (i.e. its **parameters**).
- It will do its job.
- You can wait for it to report on what it's done (i.e. its **returnValue**) or simply assume that it has done its job.

An concrete example of a function with no returnValue and no parameters:

To indicate that the function expects no **parameters** and returns no **returnValue**, we insert the word **void** where the parameters would otherwise be.

```
// Ready to shoot!

void shooTheScene (void) {
   quietOnTheSet();
   lights();
   camera();
   action();
}
```

To call it we simply yell:

```
shootTheScene();
```

An concrete example of a function with one returnValue but no parameters:

To indicate that the function returns a **returnValue** but expects no **parameters**, we insert the data type of the **returnValue** before the function name and the word **void** where the **parameters** would otherwise be.

```
// Is there enough film in the camera?
int howMuchFilmIsLeft (void) {
   int feetLeft = 1000 - feetUsed;
   return feetLeft;
}
```

Since the function has an integer return value, we can use the function call itself as a variable in another statement:
The function call will be replaced by its return value, the number of feet that are left.

```
if(howMuchFilmIsLeft() < 50) reloadTheCamera();
```

In other words, if there are less than 50 feet of film left, tell the cameraman to reload.

An concrete example of a function with no returnValue but several parameters:

To indicate that the function expects no **returnValue** but does take **parameters**, we insert the word **void**
where the **returnValue** would otherwise be and data types and variable names of the **parameters** .

```
// On to the next scene!
void prepareForNextScene (int act, int scene) {
   intScriptPage = act * 10 + scene;
   placesEveryone();
}
```

To call it we may simply say:

```
prepareForNextScene(3, 4);
```

An concrete example of a function with one returnValue and several parameters:

To indicate that the function expects one boolian (true or false) **returnValue** and two **parameters**, we insert the data type
of the **returnValue** `bool` and data types and variables `int` of the two **parameters** .

```
// Get the actors on stage!
bool actorsToYourMarks (int femaleLead, int maleLead) {
   while (femaleLead == notReady) {rehearse()};
   while (maleLead == notReady) {rehearse()};
   bool allReady = true;
   return allReady;
}
```

Since the function has a boolean **returnValue**, we can use the function call itself as a variable in a larger statement:
The function call will be replaced by its return value, and we can call **action()**.

```
if(actorsToYourMarks(11, 27)) action();
```

In other words, as long as the male and female leads are not ready, they should rehearse.
When they are ready, we will call for action.

Color Graphics Language

175

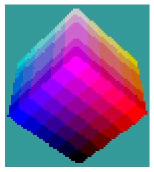
Color Theory
Understanding Color Spaces

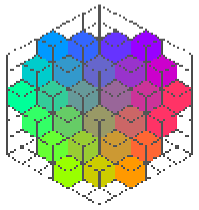
Color is represented by three variables, with values from 0 to 255:
red, green, blue.

The gamut of color is often displayed as a color cube with three dimensions:
red, green, blue.

The 8 corners of the color cube are shown in the rightmost column ->

RED value	*GREEN value*	*BLUE value*	*Result*	*Notation*	*BGR hex code*
255	255	255	WHITE	W	0xFFFFFF
255	0	0	RED	Additive Primaries: RGB	0x0000FF
0	255	0	GREEN		0x00FF00
0	0	255	BLUE		0xFF0000
0	255	255	CYAN	Subtractive Primaries: CMY	0xFFFF00
255	0	255	MAGENTA		0xFF00FF
255	255	0	YELLOW		0x00FFFF
0	0	0	BLACK	K	0x000000





Color Constants

Windows recognizes these color constants, representing the corners of the color cube (top row), plus darker variants (bottom row).
Windows also recognizes other colors (see the color combo-box in the properties tab of most visual components).

`clRed` *additive primary RED*	`clLime` *additive primary GREEN*	`clBlue` *additive primary BLUE*	`clAqua` *subtractive primary CYAN*	`clFuchsia` *subtractive primary MAGENTA*	`clYellow` *subtractive primary YELLOW*	`clBlack` *BLACK*	`clWhite` *WHITE*
clMaroon	`clGreen`	`clNavy`	`clTeal`	`clPurple`	`clOlive`	`clGray`	`clSilver`

All coordinates on your display are measured in pixels from the top left corner of the component.

- *the top left pixel has the coordinates 0, 0*
- *x is the distance to the right or east*
- *y is the distance down or south*

Form Component

The `Form` *component is the entire Window of your application. You can draw anywhere on a Window using* **`Form1->Canvas`**.

PaintBox Component

The `PaintBox` *component defines a smaller region within the Window on which your application can draw.*
All measurements are relative to the top left corner of the PaintBox.

A `PaintBox` *is useful for maintaining multiple visualizations on the screen at one time. Use it as you would use* **`Form1->Canvas`**.

Canvas Shape Methods

- **`Canvas->Ellipse(x1, y1, x2, y2);`**
  where x1 and y1 represent the top left corner of the object, and x2 and y2 the bottom right
- **`Canvas->MoveTo(x, y);`**
  moves the cursor to x and y in order to begin a `LineTo()` *command*
- **`Canvas->LineTo(x, y);`**
  draws a line to x and y
- **`Canvas->Rectangle(x1, y1, x2, y2);`**
  where x1 and y1 represent the top left corner of the shape and x2 and y2 the bottom right
- **`Canvas->RoundRect(x1, y1, x2, y2, x3, y3);`**
  where x3 and y3 represent the diameters of curvature of the rounded corners
- **`Canvas->TextOut(x, y, "string");`**
  writes the specified string beginning at position x and y
- **`Refresh( );`**
  clears the canvas without resetting the Brush and Pen values

Canvas Polygons and Polylines

- **`TPoint points[6];`**
  creates an array of six `TPoint` *objects with elements numbered from 0 to 5*
- **`points[0].x = 40; points[0].y = 10;`**
- **`points[1].x = 20; points[1].y = 60;`**
- **`points[2].x = 70; points[2].y = 30;`**
- **`points[3].x = 10; points[3].y = 30;`**
- **`points[4].x = 60; points[4].y = 60;`**
- **`points[5].x = 40; points[5].y = 10;`**
  fills the array with `x` *and* `y` *values*
- **`Canvas->Polyline(points,5);`**
  draws a line with the current `Pen` *color beginning with* `points` *element* `[0]` *and ending with* `points` *element* `[5]`
- **`Canvas->Polygon(points, 5);`**
  draws a line with the current `Pen` *color and fills it with the current* `Brush` *color beginning with* `points` *element* `[0]` *and ending with* `points` *element* `[5]`

Canvas Puting Pixels

- **`Canvas->Pixels[x][y] = clBlue;`**
  puts a blue pixel at coordinate x and y.
  any color constant designated by the prefix letters (not numerals) "cl" may be used.
- **`Canvas->Pixels[x][y] = 0xFF0000;`**
  puts a pixel at coordinate x and y given by the hexadecimal byte values of its Blue, Green, and Red coordinates in the color cube.
- **`Canvas->Pixels[x][y] = static_cast<TColor>(RGB(red, green, blue));`**
  puts a pixel at coordinate x and y given by the decimal (0-255) byte values of its Blue, Green, and Red coordinates in the color cube.
- **`Canvas->Pixels[x][y] = static_cast<TColor>(colorRamp(range, value));`**
  puts a pixel at coordinate x and y given by the a numerical value along a range of numerical values (see colorRamp()).

Getting Pixels

- **`int color, red, green, blue;`**
  declares variables to receive colors
- **`color = Form1->Canvas->Pixels[X][Y];`**
  retrieves the RGB color triplet
- **`red = GetRValue(color);`**

retrieves the Red color component

- **`green = GetGValue(color);`**
  retrieves the Green color component
- **`blue = GetBValue(color);`**
  retrieves the Blue color component

Properties

- **`Canvas->Pen->Color = clBlack;`**
  where `Pen` *is the outline color. Black is the default*
- **`Canvas->Pen->Style = psSolid;`**
  where `Style` *can be* `psSolid`, `psClear`, `psDash`, `psDot`, `psDashDot` *or* `psDashDotDot`. *Solid is the default*
- **`Canvas->Pen->Width = 1;`**
  if `Width` *is greater than one, then* `Style` *may revert to* `psSolid`. *One is the default*
- **`Canvas->Brush->Color = clWhite;`**
  where `Brush` *is the fill color. White is the default*
- **`Canvas->Brush->Style = bsSolid;`**
  where `Style` *can be* `bsSolid`, `bsClear`, `bsHorizontal`, `bsVertical`, `bsFDiagonal`, `bsBDiagonal`, `bsCross` *or* `bsDiagonalCross`. *Solid is the default*

Variables

77

Variable Type	Description	Size in Bytes	Range from	Range to
`char`	Character	1	-128	127
`unsigned char`	Unsigned Character	1	0	255
`short`	Short Integer	2	-32, 768	32,767
`unsigned short`	Unsigned Short Integer	2	0	65,535
`int` **`long`**	Integer Long Integer	4	-2,147,483,648	2,147,483,647

`unsigned int`
`unsigned long`
	Unsigned Integer
	Unsigned Long Integer
		4	0	4,294,967,296

`float`	Floating Point (Real)	4	-3.4E308 -3.4 x 10^308	3.4E38 -3.4 x 10^308
`double`	Double Floating Point (Real)	8	-1.7E308 -1.7 x 10^308	1.8E308 1.8 x 10^308
`bool`	Boolean (true or false)	1	false	true

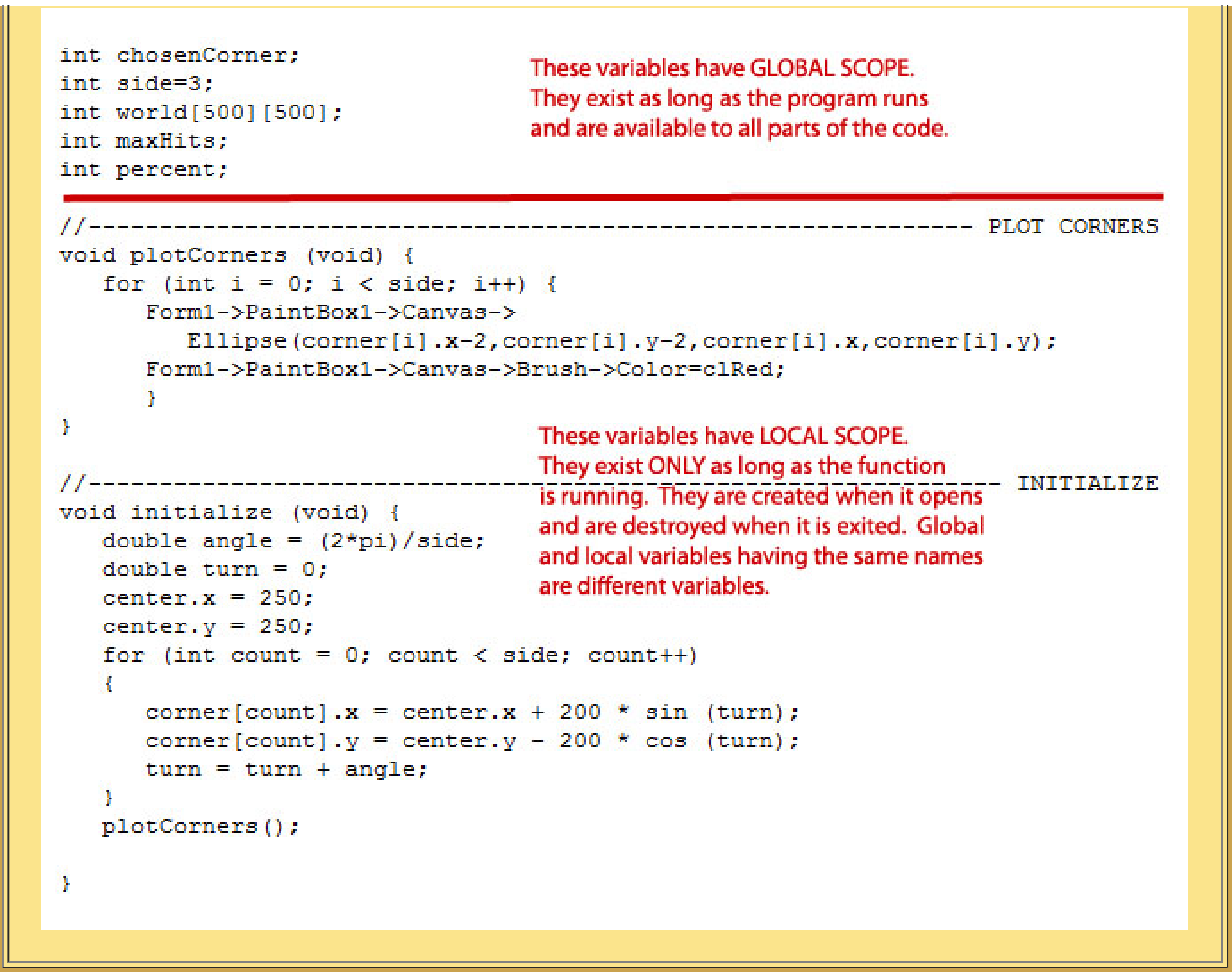
Upper and Lower-Case Variable Names

By convention, if you declare and define a variable, you should give it a name beginning with a lower-case letter.

Variable Arrays

A variable may hold a single value like: `myAge`
A one-dimensional list of the ages of 25 students in this class: `studentAge[25]` with indices ranging from 0 to 24.
A two-dimensional table of the position of pieces on a chess board: `chessGame[8][8]` with indices ranging from 0 to 7.
Indices are always set off by square brackets `[]` and we always begin counting index values at zero.

Variable Scope



```
int chosenCorner;
int side=3;
int world[500][500];
int maxHits;
int percent;
```

These variables have GLOBAL SCOPE.
They exist as long as the program runs
and are available to all parts of the code.

```
//----------------------------------------------------------------- PLOT CORNERS
void plotCorners (void) {
   for (int i = 0; i < side; i++) {
      Form1->PaintBox1->Canvas->
         Ellipse(corner[i].x-2,corner[i].y-2,corner[i].x,corner[i].y);
      Form1->PaintBox1->Canvas->Brush->Color=clRed;
      }
}

//------------------------------------------------------------------- INITIALIZE
void initialize (void) {
   double angle = (2*pi)/side;
   double turn = 0;
   center.x = 250;
   center.y = 250;
   for (int count = 0; count < side; count++)
   {
      corner[count].x = center.x + 200 * sin (turn);
      corner[count].y = center.y - 200 * cos (turn);
      turn = turn + angle;
   }
   plotCorners();

}
```

These variables have LOCAL SCOPE.
They exist ONLY as long as the function
is running. They are created when it opens
and are destroyed when it is exited. Global
and local variables having the same names
are different variables.

B7 B6 B5				Ø Ø Ø	Ø Ø 1	Ø 1 Ø	Ø 1 1	1 Ø Ø	1 Ø 1	1 1 Ø	1 1 1
BITS				CONTROL		NUMBERS & SYMBOLS		UPPERCASE		LOWERCASE	
B4	B3	B2	B1								
Ø	Ø	Ø	Ø	NUL 0	DLE 16	SP 32	0 48	@ 64	P 80	\ 96	p 112
Ø	Ø	Ø	1	SOH 1	DC1 17	! 33	1 49	A 65	Q 81	a 97	q 113
Ø	Ø	1	Ø	STX 2	DC2 18	" 34	2 50	B 66	R 82	b 98	r 114
Ø	Ø	1	1	ETX 3	DC3 19	# 35	3 51	C 67	S 83	c 99	s 115
Ø	1	Ø	Ø	EOT 4	DC4 20	$ 36	4 52	D 68	T 84	d 100	t 116
Ø	1	Ø	1	ENQ 5	NAK 21	% 37	5 53	E 69	U 85	e 101	u 117
Ø	1	1	Ø	ACK 6	SYN 22	& 38	6 54	F 70	V 86	f 102	v 118
Ø	1	1	1	BEL 7	ETB 23	' 39	7 55	G 71	W 87	g 103	w 119
1	Ø	Ø	Ø	BS 8	CAN 24	(40	8 56	H 72	X 88	h 104	x 120
1	Ø	Ø	1	HT 9	EM 25	) 41	9 57	I 73	Y 89	i 105	y 121
1	Ø	1	Ø	LF 10	SUB 26	* 42	: 58	J 74	Z 90	j 106	z 122
1	Ø	1	1	VT 11	ESC 27	+ 43	; 59	K 75	[91	k 107	{ 123
1	1	Ø	Ø	FF 12	FS 28	, 44	< 60	L 76	\ 92	l 108	| 124
1	1	Ø	1	CR 13	GS 29	– 45	= 61	M 77	] 93	m 109	} 125
1	1	1	Ø	SO 14	RS 30	. 46	> 62	N 78	↑ 94	n 110	~ 126
1	1	1	1	SI 15	US 31	/ 47	? 63	O 79	_ 95	o 111	↓ 127