楽しいC言語

2020/04/16

長谷川 喬恒

内容

1・・・　はじめに

楽しい C 言語は変数や配列など、ある程度コードについて学んでいる前提で書いています。環境は VisualStudio2019（記載現在）、C 言語を使います。

VisualStudio ダウンロード

C++によるデスクトップ開発を選択する。

2 ・・・　2 進数、16 進数

2 進数、16 進数とは？

私たちにとってわかりやすい数字は 0~9 までの数字で 10 進数と言います。それに対して

コンピュータがわかりやすいのが 0 と 1 までの 2 進数と 0~F までの 16 進数です。

10 進数	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16 進数	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2 進数	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

なぜ、この 2 進数・16 進数がコンピュータにとってわかりやすいかというと、コンピュー

タが電気のオンオフを制御して処理を行っているからなのです。

そして、オンオフの制御を 8 つあつめて 1 つのまとまりとしているのです。

オンオフ→1 ビット　　8 ビット→1 バイトと表現します。

1 ビットは 0/1 で表現するので 2 進数、1 バイトは 0/1 が 8 つあつまっているので 2 x 8 で

16 進数となります。

やってみよう!!

10 進数 2 進数 16 進数それぞれの値を書式指定して出力しています。ただし、 2 進数の値は書式指定がないので別途 Decimal4Binary 関数で出力しています。

```c
#include <stdio.h>

void Decimal4Binary(int value);

int main()
{
    printf("10進数→10進数：  %d¥n", 255);          // 10進数
    printf(" 2進数→10進数：  %d¥n", 0b11111111);   // 2進数
    printf("16進数→10進数：  %d¥n", 0xff);         // 16進数

    printf("¥n");

    int value = 255;
    printf("10進数→10進数：  %d¥n", value);        // 10進数
    printf("10進数→ 2進数：  ");
    Decimal4Binary(value);                          // 2進数
    printf("10進数→16進数：  %x¥n", value);        // 16進数

    return 0;
}

void Decimal4Binary(int value)
{
    int index = 0;
    while (0 < value)
    {
        if (index != 0 && (index % 4) == 0)printf(",");
        printf("%d", value % 2);
        value = value / 2;
        index++;
    }
    printf("¥n");
}
```

```
10進数→10進数：  255
 2進数→10進数：  255
16進数→10進数：  255

10進数→10進数：  255
10進数→ 2進数：  1111,1111
10進数→16進数：  ff
```

N 進数の変換方法

例）10 進数：58　2 進数：111010　16 進数：3A

10 進数→2 進数

2)58 ･･･0

2)29 ･･･1

2)14 ･･･0

2) 7 ･･･1

2) 3 ･･･1

2) 1

111010

値の商が 1 になるまで 2 で割り続けます。値を割り終えたら、一番下の商から順に上の余りを連結した値が 2 進数の値となります。

2 進数→10 進数

32　16　8　4　2　1

1　1　1　0　1　0

32 + 16 + 8 + 0 + 2 + 0 = 58

2 進数は各桁が 2 の N 乗の値で求められます。求めた桁ごとの値と 2 進数の 0 または 1 を掛け合わせた合計が 10 進数の値となります。

10 進数→16 進数

16)58 ･･･10

3

3 10 → 3A

値の商が割り切れなくなるまで 16 で割り続けます。値を割り終えたら、一番下の商から順に上の余りを連結した値が 16 進数の値となります。

16 進数→10 進数

16　1

3　A

48 + 10 = 58

16 進数は各桁が 16 の N 乗の値で求められます。求めた桁ごとの値と 16 進数の数値を掛け合わせた合計が 10 進数の値となります。

※中には 8 進数表記もあります。2・16 の部分を 8 に置き換えて計算します。

3・・・データ型の種類

データ型

型	説明
char	1バイトの符号付整数（-128~127）の値を記憶できる。 1バイト文字（英数字など）を1文字記憶できる。
unsigned char	1バイトの符号なし整数（0~255）の値を記憶できる。
int	2または4バイトの符号付整数の値を記憶できる。 （2バイトなら-2の15乗～2の15乗-1、4バイトなら-2の31乗～2の31乗-1）
short	2バイトの符号付整数（-2の15乗～2の15乗-1）の値を記憶できる。
long	4バイトの符号付整数（-2の31乗～2の31乗-1）の値を記憶できる。
unsigned	2バイトまたは4バイトの符号なし整数の値を記憶できる。 （2バイトなら0～2の16乗-1、4バイトなら0～2の32乗-1）
unsigned long	4バイトの符号なし整数（0～2の32乗-1）の値を記憶できる。
unsigned short	2バイトの符号なし整数（0～2の16乗-1）の値を記憶できる。
float	4バイトの単精度浮動小数点実数（有効桁数7桁）
double	8バイトの倍精度浮動小数点実装（有効桁数16桁）

※バイト数が変動する型はコンパイラによって決まる。

定数の書き方

データの種類	例	説明
文字	‘a’	1 バイト文字（英数字など）1 文字
文字列	“abcd”	「“」で囲まれた文字の並び
整数	123	Int 型データとして扱われる。
整数（8 進数）	098	先頭が「0」の場合は 8 進数として扱われる。
整数（16 進数）	0x3A	先頭が「0x」の場合は 16 進数として扱われる。
符号なし整数	987U	最後に「u」「U」をつけると符号なし整数として扱われる。
long 型整数	9876L	最後に「l」「L」をつけると long 型整数として扱われる。
実数	3.14F	最後に「f」「F」をつけると単精度浮動小数点実数（有効桁数 7 桁）として扱われる。
Double 型実数	12.3456	倍精度浮動小数点実数（有効桁数 16 桁）として扱われる。

※データ型の容量は 2 進数・16 進数の説明と関係しています。char 型の説明には（-128~127）の値を記憶できるとあります。これは 256 の値を記憶できると変換できます。1Byte(8bit)の 1111 1111 は 10 進で 255 です。0 を含めると 256 の値を表現できます。これはコンピュータにわかりやすい値で容量を割り振っているといえます。

やってみよう!!

printf()で書式指定して出力しています。書式の種別はコード上のコメント通り

上から 10 進整数・float 型実数・double 型実数・文字・文字列・8 進整数・16 進整数の順に出力しています。

```
void Sample2_1()
{
    printf("%d¥n", 100);          // 10進整数
    printf("%f¥n", 1.2f);         // float型実数
    printf("%lf¥n", 3.14);        // double型実数
    printf("%c¥n", 'A');          // 文字
    printf("%s¥n", "Hello");      // 文字列
    printf("%o¥n", 0100);         // 8進整数
    printf("%x¥n", 0x100);        // 16進整数
}
```

```
100
1.200000
3.140000
A
Hello
100
100
```

データ型の違う演算

異なるデータ型で演算して出力した処理を書きました。データ型が違う場合は

精度*が高い型に合わせて計算されます。例えば

1(int 型)+2.5(float 型)=3.5(float 型)のように計算されます。

精度*・・・　桁数や小数点以下などをどこまで表現できるかの範囲です。

```
void Sample2_2()
{
    char ch = 11;           // char型
    int it = 22;            // int型
    short sh = 33;          // short型
    long lg = 44l;          // long型
    float fl = 55.5f;       // float型
    double dbl = 66.6666;   // double型

    printf("char  + int    :%d¥n", ch + it);
    printf("short + long   :%d¥n", sh + lg);
    printf("int   + float  :%f¥n", it + fl);
    printf("int   + double :%lf¥n", it + dbl);
    printf("float + double :%lf¥n", fl + dbl);
}
```

```
char  + int    :33
short + long   :77
int   + float  :77.500000
int   + double :88.666600
float + double :122.166600
```

char 型について

char 型は文字を記憶すると書きましたが、そのままでは、文字列としては使えません。文字列を記憶するためにはもうひと手間必要になります。

配列を使った文字列

char(型) str1(変数名)[256(サイズ)];配列のサイズを指定して宣言する。

char(型) str1(変数名)[] = “Hello”;文字列を代入して、サイズ指定を省略する。

2 種類の変数宣言の方法があります。

文字列の末尾は終端文字「¥0」が付き、配列サイズも文字数＋ 1 と数えます。

```
void Sample3_1()
{
    char str1[256] = "Hello";
    char str2[] = "Hello";

    printf("%s¥n", str1);

    for (int i = 0; i < sizeof(str2); i++) {
        printf("len %d: %c¥n", i + 1, str2[i]);
    }
}
```

```
Hello
len 1: H
len 2: e
len 3: l
len 4: l
len 5: o
len 6:
```

※sizeof()のサイズは文字数+終端文字になります。

ポインタを使う場合

文字型ポインタを宣言します。アドレスは変更してはいけない値のため定数として格納しなくてはいけません。ポインタについては後に詳しく説明します。

今はこのような使い方ができるという事だけ知っておいてください。

```
void Sample3_2()
{
    const char* pntr = "Hello";

    printf("%s¥n", pntr);

    for (int i = 0; i < strlen(pntr); i++) {
        printf("len %d: %c¥n", i + 1, pntr[i]);
    }
}
```

```
Hello
len 1: H
len 2: e
len 3: l
len 4: l
len 5: o
```

※今回、for 分でサイズ指定するのに strlen()を使っています。ポインタの場合 sizeof()はポインタ型のサイズを取得してしまうため、正しいサイズが出はありません。そのため、文字列の長さを取得する関数(strlen)を使っています。

(#include <string.h>をソースの頭に記述すると strlen()が使えます。)

※文字列の操作には関数が必要です。コピーには strcopy_s()、連結には strcat_s()、比較には strcmp()を使います。興味あれば調べてみてください。

4 ・・・ASCII コード

アスキーコードとは？

前章で、2 進数はコンピュータにとってわかりやすい値と書きました。ASCII（アスキー）コード表は、その 2 進数の値と私たちがわかりやすい文字（アルファベット）を設定している表の事を指します。

アスキーコード表は左のリンクを参照してください。

やってみよう!!

アルファベットの大文字と小文字を別々の方法で出力しています。

1 つ目は文字から大文字と小文字の ASCII コード上での数値の差分を取得して

大文字小文字の出力をしています。

2 つ目は 10 進数の数値から大文字と小文字を出力しています。

3 つ目は 16 進数の数値から大文字と小文字を出力しています。

```
void Sample4_1() {
    char largeChar = 'A';
    char smallChar;

    // 大文字小文字の差分
    int offset = 'a' - 'A';

    smallChar = largeChar + offset;

    // 文字から大文字小文字を取得
    printf("大文字：%c  小文字：%c¥n", largeChar, smallChar);

    // 10進数から大文字小文字を取得
    printf("大文字：%c  小文字：%c¥n", 65, 97);

    // 16進数から大文字小文字を取得
    printf("大文字：%c  小文字：%c¥n", 0x41, 0x61);
}
```

```
大文字：A  小文字：a
大文字：A  小文字：a
大文字：A  小文字：a
```

5・・・ポインタ

ポインタとは？

変数のアドレスを記憶する変数のことです。変数宣言時にメモリ上では変数のサイズ分領域を確保します。その確保した領域の場所をアドレスと呼びます。

メモリ

アドレス：1000 番号 文字列　値：“テスト”
アドレス：1001 番号 文字列　値：“てすと”
アドレス：1002 番号 ポインタ　値：1000 番号

アドレス 1002 番号にはポインタが宣言され、1000 番号アドレスの“テスト”が取得できます。

やってみよう!!

変数の値とアドレスを出力しています。変数宣言時に「＊（アスタ）」を置くことでポインタとして使え、変数の先頭に「＆（アンパサンド）」を置くことで変数のアドレスが取得できます。

青枠では変数の値とアドレスを表示しています。

緑枠ではポインタの指すアドレスから値を参照して表示、ポインタに記憶したアドレスを表示しています。

```
void Sample5_1()
{
    int value = 1;
    int* pointer = &value;

    printf("valueの値          : %d¥n", value);
    printf("valueのアドレス    : %p¥n", &value);

    printf("¥n");

    printf("pointerの値        : %d¥n", *pointer);
    printf("pointerのアドレス  : %p¥n", pointer);

    printf("¥n");
}
```

```
valueの値          : 1
valueのアドレス    : 00F3FCBC

pointerの値        : 1
pointerのアドレス  : 00F3FCBC
```

※アドレスの書式指定は「%p」です。

参照渡し関数

関数の引数にもポインタを宣言でき参照渡しと呼びます。下記では値渡しと参照渡しの比較をしています。

サンプルを実行すると、値渡し関数では加算しても出力した result 値は変わりませんが、参照渡し関数は加算すると出力した result 値は変わっています。

これは、引数に値渡しをする場合は result 値のコピーを関数に渡して、関数内でコピーにだけ加算がされているからです。対して、参照渡しの場合には result 値のアドレスから直接値の加算をしているので、関数内で値の変更ができるようになります。

※青枠：値渡し　緑枠：ポインタ渡し

```
void Increment(int result) { result++; }
void Increment(int* result) { *result = *result + 1; }
void Sample5_2()
{
    int result = 1;
    printf("加算前：%d¥n", result);
    Increment(result);
    printf("加算後：%d¥n", result);
    Increment(&result);
    printf("加算後：%d¥n", result);

    printf("¥n");
}
```

```
加算前：1
加算後：1
加算後：2
```

配列ポインタ

関数に配列ポインタを渡して、関数内で配列の一括出力をしています。

配列の先頭アドレスを引数に渡すことで、アドレス１つ分のメモリ領域で関数に配列を渡すことができます。配列は要素サイズ分の連続したメモリ領域を確保するのでアドレスに１ずらすことで次の配列要素を参照することができます。関数内ではそのようにアドレスをずらして各要素の値を出力しています。

```
void ShowItems(int* pointer, int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("No.%-2d  Value %d¥n", i + 1, *pointer);
        pointer++;
    }
}
void Sample5_3() {
    const int SIZE = 10;
    int items[SIZE] = { 0,1,2,3,4,5,6,7,8,9 };

    ShowItems(&items[0], SIZE);

    printf("¥n");
}
```

```
No.1   Value 0
No.2   Value 1
No.3   Value 2
No.4   Value 3
No.5   Value 4
No.6   Value 5
No.7   Value 6
No.8   Value 7
No.9   Value 8
No.10  Value 9
```

※変数宣言してデータ操作をする場合、コンピュータが勝手に型を認識してメモリ確保などをしてくれるのですが、ポインタの場合はアドレス先の値を直接操作しているので、コンピュータの制御を外れてデータ操作をしているという事になります。ポインタを使う際には細心の注意で使ってください。

6・・・ビット演算

ビット演算とは？

コンピュータがわかりやすい 2 進数で行う演算のことで、ビット列の 0/1 値を切り替える操作をします。ビット単位で行うため処理速度が速くメモリ使用量も少なくできます。

演算方法

通常の計算と同じように桁ごとで判定します。

2 進数：0b00000101　10 進数：5　と

2 進数：0b00001111　10 進数：15 を使い演算をしてみます。

下記で例を記述します。

論理積（AND）

0b 0 0 0 0 0 1 0 1 & 0b 0 0 0 0 1 1 1 1 = 0b 0 0 0 0 0 1 0 1

	128	64	32	16	8	4	2	1	演算
0b	0	0	0	0	0	1	0	1	
0b	0	0	0	0	1	1	1	1	&
0b	0	0	0	0	0	1	0	1	=

「1 & 1 = 1　1 & 0 = 0　0 & 1 = 0　0 & 0 = 0」のルールで両辺が 1 の場合に 1 でそれ以外を 0 と判定します。

論理和（OR）

0b 0 0 0 0 0 1 0 1 | 0b 0 0 0 0 1 1 1 1 = 0b 0 0 0 0 1 1 1 1

	128	64	32	16	8	4	2	1	演算
0b	0	0	0	0	0	1	0	1	
0b	0	0	0	0	1	1	1	1	|
0b	0	0	0	0	1	1	1	1	=

「1 & 1 = 1　1 & 0 = 1　0 & 1 = 1　0 & 0 = 0」のルールでどちらかの辺が 1 の場合が 1 で両辺が 0 の場合が 0 と判定します。

否定（NOT）

~0b 0 0 0 0 0 1 0 1 = 0b 0 0 0 0 1 0 1 0

	128	64	32	16	8	4	2	1	演算
0b	0	0	0	0	0	1	0	1	~
0b	1	1	1	1	1	0	1	0	=

「0→1　1→0」のようにそれぞれの値を反転させることで判定します。

排他的論理和（XOR）

0b 0 0 0 0 0 1 0 1 ^ 0b 0 0 0 0 1 1 1 1 = 0b 0 0 0 0 1 0 1 0

	128	64	32	16	8	4	2	1	演算
0b	0	0	0	0	0	1	0	1	
0b	0	0	0	0	1	1	1	1	^
0b	0	0	0	0	1	0	1	0	=

「1 ^ 0 = 1　0 ^ 1 = 1　1 ^ 1 = 0　0 ^ 0 = 0」のルールで両辺が 0 または 1 の場合 0 で、左辺と右辺が別の値であれば 1 になります。

シフト演算（<<,>>）

0b 0 0 0 0 0 1 0 1 << 1 = 0b 0 0 0 0 1 0 1 0

	128	64	32	16	8	4	2	1	演算
0b	0	0	0	0	0	1	0	1	<<
0b	0	0	0	0	1	0	1	0	=

0b 0 0 0 0 0 1 0 1 >> 1 = 0b 0 0 0 0 0 0 1 0

	128	64	32	16	8	4	2	1	演算
0b	0	0	0	0	0	1	0	1	>>
0b	0	0	0	0	0	0	1	0	=

ビット列を左もしくは右にずらす（シフト）操作をします。演算子（<<,>>）を挟んで左辺がシフトする値で右辺がシフトする回数です。左シフトで掛け算（x2）されていき・右シフトで割り算（/2）されていきます。シフトした際に外に出た値は消滅し、空いた枠に 0 を入れていきます。

負のシフト

0b 1 1 1 1 1 0 1 0 << 1 = 0b 1 1 1 1 0 1 0 1

	符号	64	32	16	8	4	2	1	演算
0b	1	1	1	1	1	0	1	0	<<
0b	1	1	1	1	0	1	0	1	=

左端の値を符号判定としてシフト演算をします。違いはシフトした際に空いた枠に符号ビット値を入れていくだけです。

※詳しく知りたいときは「論理シフト」「算術シフト」「2 の補数」「1 の補数」で検索してみてください。

※「AND」「XOR」を組み合わせて足し算や「シフト演算」を使った掛け算、乗算をする方法が有名な使い方です。

やってみよう!!

説明にあった論理積からシフトまで演算した結果を出力しています。

```
void Sample6_1()
{
    char value1 = 0b00000101;
    char value2 = 0b00001111;
    char result;

    ShowResult(value1);
    ShowResult(value2);
    printf("¥n");

    printf("%-12s%-5s", "論理積", "(&)");
    result = value1 & value2;
    ShowResult(result);

    printf("%-12s%-5s", "論理和", "(|)");
    result = value1 | value2;
    ShowResult(result);

    printf("%-12s%-5s", "否定", "(~)");
    result = ~value1;
    ShowResult(result);

    printf("%-12s%-5s", "排他的論理和", "(^)");
    result = value1 ^ value2;
    ShowResult(result);

    printf("%-12s%-5s", "左シフト", "(<<)");
    result = value1 << 1;
    ShowResult(result);

    printf("%-12s%-5s", "右シフト", "(>>)");
    result = value1 >> 1;
    ShowResult(result);

    printf("¥n");
    char value3 = 0b11100110;
    ShowResult(value3);
    printf("%-12s%-5s", "負の右シフト", "(>>)");
    result = value3 >> 1;
    ShowResult(result);

    printf("¥n");
}
```

ShowBinary/ShowResult は出力用の関数です。特に ShowBinary はシフト演算でビットを 1 列ずつ取得しています。

```
void ShowBinary(char value)
{
    int len = 8;
    int bit[8];
    int x;

    for (int i = 0; i < len; i++)
    {
        x = 1 << i;
        x = value & x;
        bit[len - i - 1] = x >> i;
    }

    printf("0b");
    for (int i = 0; i < len; i++)
    {
        printf("%d", bit[i]);
    }
}

void ShowResult(char result)
{
    printf("10進数：%-4d    2進数：", result);
    ShowBinary(result);
    printf("¥n");
}
```

```
C:¥Users¥0ta52¥source¥repos¥FunCLanguage¥Debug¥Sample6.exe
10進数：5        2進数：0b00000101
10進数：15       2進数：0b00001111

論理積       (&)  10進数：5        2進数：0b00000101
論理和       (|)  10進数：15       2進数：0b00001111
否定         (~)  10進数：-6       2進数：0b11111010
排他的論理和(^)  10進数：10       2進数：0b00001010
左シフト     (<<) 10進数：10       2進数：0b00001010
右シフト     (>>) 10進数：2        2進数：0b00000010

10進数：-26      2進数：0b11100110
負の右シフト(>>) 10進数：-13      2進数：0b11110011
```

7・・・オブジェクト指向（クラス）

オブジェクト指向とは？

ざっくりと話すとより良いコーディングのための考え方で、3つの要素があります。

カプセル化

処理内容を隠して使い方だけを提供する考え方です。（ブラックボックス化・ラッピング）記述方法が決まっていて人によって記述方法がまばらになってしまう処理や、変更されたくない処理などをカプセル化することで、間違った処理方法や知らないうちに改修されてしまうことを防ぐ考え方です。

多様性（ポリモーフィズム）

処理に柔軟性を持たせる考え方です。例えば足し算は整数だけでなく負数・小数・分数でも計算できます。さらに足し算と引き算は計算する値を指定して、計算結果を返す部分が同じなので、ベース部分を同じ作りにして計算処理だけを変えるなど、柔軟に対応できるコーディングをする考え方です。

継承

性質を受け継がせる考え方です。例えばＡ店レジシステムとＢ店レジシステムを作るとなった時に、まずＡ店・Ｂ店で共通するレジシステムを作ります。そこから、レジシステムをベースとしてＡ店用とＢ店用にカスタマイズを加えることで同じコーディングをさせない考え方です。ちなみに、レジシステムでバグが出た場合はベースとなるレジシステムを修正するだけで、各店舗でコード修正する手間を減らすこともできます。

やってみよう!!

名前・電話番号が登録・削除・表示できる電話帳を作り。そこから、電話番号を入力できるダイヤル、通話機能を追加してみてください。

オブジェクト指向は考え方であり、その方法に絶対的な決まりはありません。皆さんの考える最善の方法を試してみてください。そして、作者同士でレビューしてみてください。それがモノづくりの能力に直結するかと考えています。

8・・・まとめ

今回記述した内容はとてもざっくりとした内容でお伝えしています。例えば、ポインタは本１冊出せるくらいにはボリュームがあります。ビット関係は通信負荷に関する通信量の削減でいろいろと調べていた記憶があります。オブジェクト指向は様々な考え方が調べると出てきますし、デザインパターンといった言葉も出てくるかと思います。

あとは、非同期処理・Action・デリゲートと・・・・・。

さて、楽しいＣ言語はこれで終わりです。お疲れ様です。

どんな知識が何のためになるのか、私にも知りえません。これに関しては経験とアンテナをどこまで伸ばしているかによります。頑張ってください。

楽しいＣ言語サンプルを作りました。興味があれば見てみてください。

付録・・・アスキーコード表

アスキーコード表参照

10進	16進	ASCII	10進	16進	ASCII	10進	16進	ASCII	10進	16進	ASCII
0	0	NULL	32	20	SP	64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENG	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	'	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	HT	41	29	)	73	49	I	105	69	i
10	A	LF	42	2A	*	74	4A	J	106	6A	j
11	B	VT	43	2B	+	75	4B	K	107	6B	k

12	C	FF	44	2C	,	76	4C	L	108	6C	l
13	D	CR	45	2D	-	77	4D	M	109	6D	m
14	E	SO	46	2E	.	78	4E	N	110	6E	n
15	F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{

28	1C	FS	60	3C	<	92	5C	¥	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL