Zawartość:

- wskaźniki

- tablice

- stringi

Wskaźnik – zmienna zawierająca adres wskazujący konkretny bajt w pamięci. Wskaźnik zawsze wskazuje na bajt, nie ważne jaki jest typ wskaźnika. Wskaźnik nie niesie ze sobą informacji ile przestrzeni od wskazanego miejsca możemy użyć/dostaliśmy/zaalokowaliśmy/zużyliśmy.

Komputer 32 bitowy może przechowywać liczby od 0x00000000 do 0xFFFFFFFF. Czyli 4GB. Każdy bajt ma swój unikalny adres. Adresy najlepiej zapisywać w systemie szesnastkowym, ponieważ jest on czytelniejszy dla dużych liczb.

Komputer 64 bitowy będzie miał zakres adresów. 0x0000000000000000 do 0xFFFFFFFFFFFFFFFF.

Opiszę tutaj 32 bity dla uproszczenia.

Dwie cyfry w liczbie szesnastkowej to bajt. Zakres bajta to 0x00 do 0xFF. Cyfra w systemie szesnastkowym nazywana jest nybblem.

Little Endian:

Konwencja zapisu liczb w pamięci polegająca na odwracaniu kolejności bajtów:

unsigned char a = 0xAB; //w pamięci AB

unsigned short b = 0xABCD; //w pamięci CD AB

unsigned int c = 0xABCDEF98; //w pamięci 98 EF CD AB

Konwencja ta stosowana jest oprogramowaniu.

Big Endian:

To normalna konwencja zapisu liczb. Stosowana w sieciach (w pakietach, ramkach itp.)

Przypomnienie:

Zakres unsigned char to 0x00 do 0xFF, unsigned short to 0x0000 do 0xFFFF, unsigned int to 0x00000000 do 0xFFFFFFFF.

Zakres liczb ze znakiem są analogicznie szerokie, jednak przesuwają się w stronę minus nieskończoności tak, że 0 jest w środku każdego.

Poniższe informacje są przedstawione za pomocą C/C++. Języki wyższego poziomu oprócz C# nie dają możliwości korzystania ze wskaźników co nie oznacza, że zaprezentowane mechanizmy są tam inne.

Załóżmy, że mamy tyle pamięci RAM ile na obrazku (tak zdarzają się takie układy):

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Widać na nim adresy każdego bajtu.

Zróbmy w funkcji main następujące zmienne:

unsigned char a = 0xAB;

unsigned short b = 0xFEBA;

unsigned int c = 0xDEADC0DE;

Dla uproszczenia uznajmy, że zostaną one rozmieszczone dowolnie.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010	00	00	00	00	AB	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	BA	FE	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	DE	C0	AD	DE	00	00	00	00	00	00

Zgodnie z opisem na poprzedniej stronie bajty zostały odwrócone. Oczywiście gdy odczytamy te wartości bajty wrócą do odpowiedniej kolejności. Zróbmy teraz wskaźniki na te wartości.

unsigned char* a_ptr = &a; //a_ptr ma wartość 0x00000014

unsigned short* b_ptr = &b; //b_ptr ma wartość 0x00000035

unsigned int* c_ptr = &c; //c_ptr ma wartość 0x00000056

Proste? Tak jak mówiłem sam wskaźnik(adres) wskazuje tylko na **JEDEN** bajt. Typ wskaźnika mówi nam dwie rzeczy:

- na co wskazuje adres/co możemy tym wskaźnikiem pobrać z pamięci

- o ile się przesunąć w pamięci jeżeli dodamy/odejmiemy coś do/od adresu (o tym dalej)

Odzyskajmy teraz wartości z pod adresów do nowych zmiennych:

unsigned char f = *a_ptr; //f ma wartość 0xAB

unsigned short g = *b_ptr; //g ma wartość 0xFEBA

unsigned int h = *c_ptr; //h ma wartość 0xDEADCODE

Tak jak napisałem wcześniej język wiedział, co pobrać z pamięci na podstawie typu wskaźnika. Wyłuskanie unsigned char* wyłuska 1 bajt itd.

Ale typy wskaźników można dowoli castować. Przykład:

unsigned char* m_ptr = (unsigned char*)c_ptr; //zapisuje adres z c_ptr w m_ptr z nowym typem

unsigned char m = *m_ptr; //m ma wartość 0xDE;

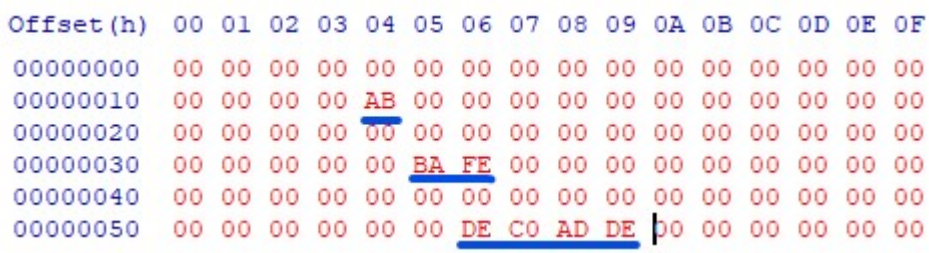
Co się stało? W sumie nic strasznego. Tak jak napisałem wskaźniki są bardzo elastyczne (są też bardzo niebezpieczne, ale to teraz jest nie ważne). c_ptr zawierał adres 0x00000056. Zmieniając typ wskaźnika poinformowałem język, że chce pobrać z tego adresu bajt a nie jak to było wcześniej 4 bajty. Proste?

Wskaźniki można castować dowoli. Można zmienić wskaźnik na inta we wskaźnik na stringa. Można zmienić wskaźnik na obiekt klasy w wskaźnik na znak. Nie zawsze to ma sens, ale czasem ma. Jest to poprawna konstrukcja języka.

Typ wskaźnika ma drugie bardzo ważne zadanie. Wskaźnik to adres a adres to liczba. Można więc na niej wykonywać operacje arytmetyczne (dodawanie i odejmowanie) by przesuwać się po pamięci. Oczywiście nie możemy biegać sobie wszędzie i dowoli, ale to bardzo zaawansowany temat.

Typ wskaźnika daje językowi jak wspomniałem dwie informacje. Pierwsza opisana już to to, że język wie co wyłuskać z pod danego adresu. Druga informacja to to o ile przesunąć adres w momencie dodania i odjęcia od niego wartości.

Przypomnę tutaj wcześniejsze zmienne i wskaźniki:



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010	00	00	00	00	AB	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	BA	FE	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	DE	C0	AD	DE	00	00	00	00	00	00

Zgodnie z opisem na poprzedniej stronie bajty zostały odwrócone. Oczywiście gdy odczytamy te wartości bajty wrócą do odpowiedniej kolejności. Zróbmy teraz wskaźniki na te wartości.

unsigned char* a_ptr = &a; //a_ptr ma wartość 0x00000014

unsigned short* b_ptr = &b; //b_ptr ma wartość 0x00000035

unsigned int* c_ptr = &c; //c_ptr ma wartość 0x00000056

Wykonam teraz następujące operacje:

a_ptr++; //a_ptr ma nową wartość równą bez zaskoczenia 0x00000015

b_ptr++; //b_ptr ma nową wartość (UWAGA) 0x00000037

c_ptr++;//c_ptr ma nową wartość (UWAGA) 0x0000005A

Co się stało? To ile zostanie dodane lub odjęte do/od adresu liczymy ze wzoru:

N = dodana/odjęta wartość * sizeof(typ_wskaźnika)

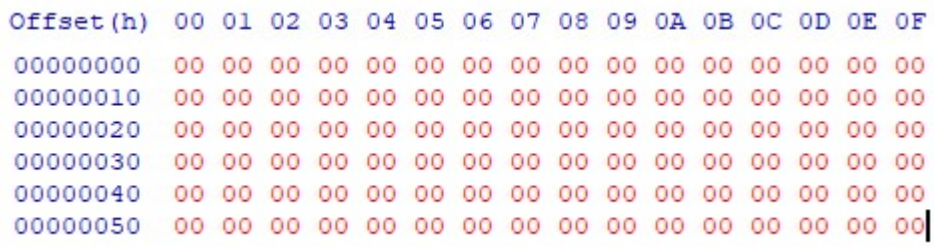
unsigned char ma rozmiar 1 bajtu więc w przypadku ++ adres zostanie zwiększony o 1*1

unsigned short ma rozmiar 2 bajtów więc w przypadku ++ adres zostanie zwiększony o 1*2

unsigned int ma rozmiar 4 bajtów więc w przypadku ++ adres zostanie zwiększony o 1*4

Tablice i Stringi

Tablica to pewien wycinek pamięci. Definiując tablicę 3 intów alokujemy 3*sizeof(int) bajtów czyli 3*4=12 bajtów.

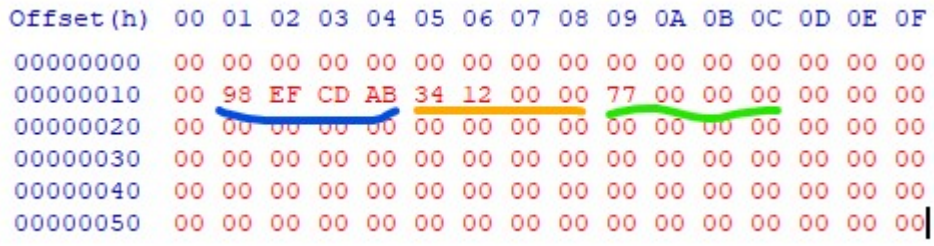


Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Wracamy do pustej pamięci. Zaalokujmy statyczną tablice 3 intów (rodzaje dynamiczne/statyczne opiszę niżej).

unsigned int tablica[3] = {0xABCDEF98, 0x1234, 0x77}; //celowo używam tu liczb nie koniecznie wykorzystujących potencjał zakresu inta

W pamięci wygląda to tak:



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010	00	98	EF	CD	AB	34	12	00	00	77	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Pierwsza ciekawostka to to, że tablica zapisuje wartości ciągiem jedna za drugą (w przeciwieństwie do listy). Druga ważna uwaga to Little Endian na każdej wartości. Trzecia uwaga to to, że wszystkie liczby zostały wyrównane do rozmiaru unsigned int czyli 4 bajtów (dlatego zawsze zalecam myśleć nad tym jaki typ się dobiera bo nie ma sensu trzymać ocen w skali szkolnej w tablicy intów jak unsigned char to za dużo na takie wartości).

Czwarta uwaga to fakt, że nazwa tablica to tak naprawdę wskaźnik: unsigned int*

Piątka uwaga to to, że tak jak mówiłem wskaźnik nie niesie ze sobą informacji na ile miejsca wskazuje. Tak więc programista musi pamiętać i podawać dalej ile elementów zaalokował i zapisał.

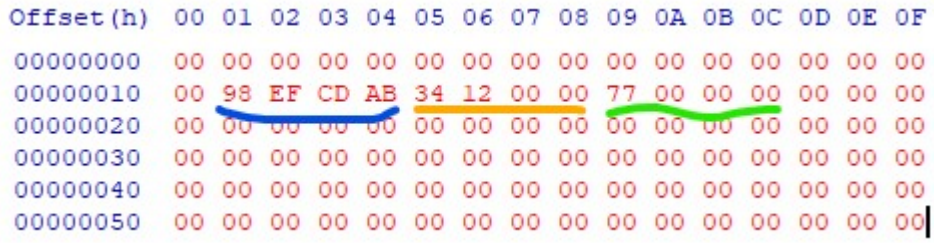
Tak jak napisałem nazwa tablica to wskaźnik unsigned int* z wartością w tym przypadku 0x00000011. Wyłuskanie go da nam wartość 0xABCDEF98.

Dodanie do wskaźnika 1 przesunie nas o 4 bajty (1*rozmair inta).

unsigned int k = *(tablica+1); //k ma wartość 0x00001234

unsigned int l = *(tablica+2); //l ma wartość 0x00000077

Ale tak jak mówiłem wskaźniki można castować do woli, więc pobawmy się:



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010	00	98	EF	CD	AB	34	12	00	00	77	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

unsigned char* tablica2 = (unsigned char*)tablica;

*(tablica2) da nam 0x98

*(tablica2+2) da nam 0xCD

*(tablica2+8) da nam 0x77

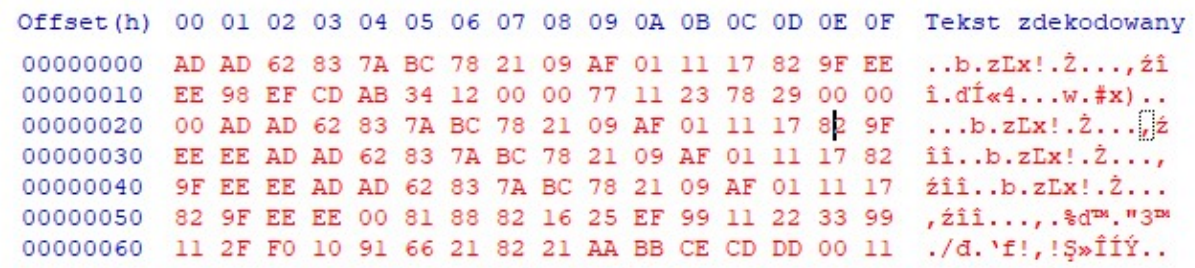
String

String to tak naprawdę tablica znaków. Wysoko poziomowe implementacje to oczywiście tylko nakładki na stare dobre stringi z C.

Zobaczmy taki przykład:

std::cout<<”ASD”<<endl;

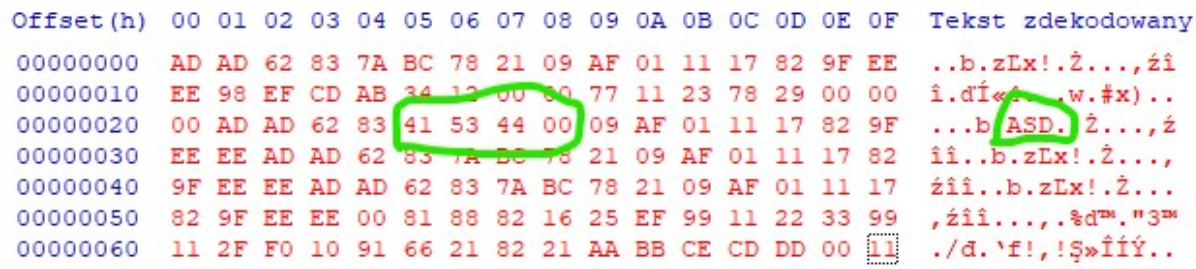
Gdzieś w pamięci program umieścił tablicę 4 charów, a w tym miejscu tylko i wyłącznie się do niej odwołuje za pomocą wskaźnika/adresu. Zobaczmy pamięć:



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Tekst zdekodowany
00000000	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	82	9F	EE	..b.zLx!.Ż...,źî
00000010	EE	98	EF	CD	AB	34	12	00	00	77	11	23	78	29	00	00	î.dÍ«4...w.#x)..
00000020	00	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	82	9F	...b.zLx!.Ż...,ź
00000030	EE	EE	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	82	îî..b.zLx!.Ż...,
00000040	9F	EE	EE	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	źîî..b.zLx!.Ż...
00000050	82	9F	EE	EE	00	81	88	82	16	25	EF	99	11	22	33	99	,źîî...,.%d™."3™
00000060	11	2F	F0	10	91	66	21	82	21	AA	BB	CE	CD	DD	00	11	./đ.‘f!,!Ş»ÎÍÝ..

Oczywiście pamięć nie jest czysta jak łza to znaczy wyzerowana. Znajduje się tam wiele śmieci, jakiś pozostałości albo faktycznie jakieś istotne rzeczy. Nie wiemy gdzie program umieszcza nasze dane i co jest obok. Wcześniej wyzerowałem obrazek, żeby poprawić czytelność.

Umieszczając powyższą linie w kodzie programu w pamięci pojawia się umieszczony tam string.

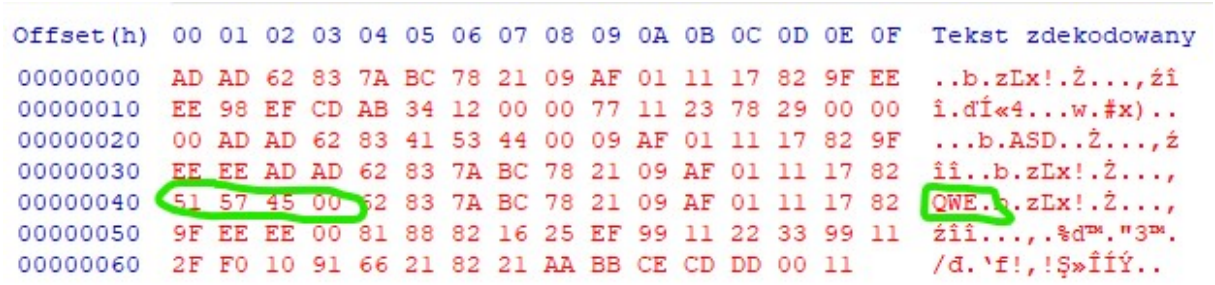


Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Tekst zdekodowany
00000000	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	82	9F	EE	..b.zLx!.Ż...,źî
00000010	EE	98	EF	CD	AB	34	12	00	00	77	11	23	78	29	00	00	î.dÍ«4...w.#x)..
00000020	00	AD	AD	62	83	41	53	44	00	09	AF	01	11	17	82	9F	...b.ASD..Ż...,ź
00000030	EE	EE	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	82	îî..b.zLx!.Ż...,
00000040	9F	EE	EE	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	źîî..b.zLx!.Ż...
00000050	82	9F	EE	EE	00	81	88	82	16	25	EF	99	11	22	33	99	,źîî...,.%d™."3™
00000060	11	2F	F0	10	91	66	21	82	21	AA	BB	CE	CD	DD	00	11	./đ.‘f!,!Ş»ÎÍÝ..

Napisałem wcześniej, że „ASD” zajmuje 4 bajty. Jest tak, ponieważ każdy string jest kończony bajtem 0x00 by język, mógł określić jego koniec. Takie luźne zapisanie „ASD” samo doda 0 na koniec, ale nie zawsze tak jest.

Dodajmy manualny string za pomocą tablicy:

char napis[] = {‘Q’,’W’,’E’,0}; //tutaj już musimy dodać 0 na końcu, samo się nie doda



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Tekst zdekodowany
00000000	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	82	9F	EE	..b.zLx!.Ż...,źî
00000010	EE	98	EF	CD	AB	34	12	00	00	77	11	23	78	29	00	00	î.dÍ«4...w.#x)..
00000020	00	AD	AD	62	83	41	53	44	00	09	AF	01	11	17	82	9F	...b.ASD..Ż...,ź
00000030	EE	EE	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	82	îî..b.zLx!.Ż...,
00000040	51	57	45	00	62	83	7A	BC	78	21	09	AF	01	11	17	82	QWE.b.zLx!.Ż...,
00000050	9F	EE	EE	00	81	88	82	16	25	EF	99	11	22	33	99	11	źîî...,.%d™."3™.
00000060	2F	F0	10	91	66	21	82	21	AA	BB	CE	CD	DD	00	11		/đ.‘f!,!Ş»ÎÍÝ..

Wykonanie lini:

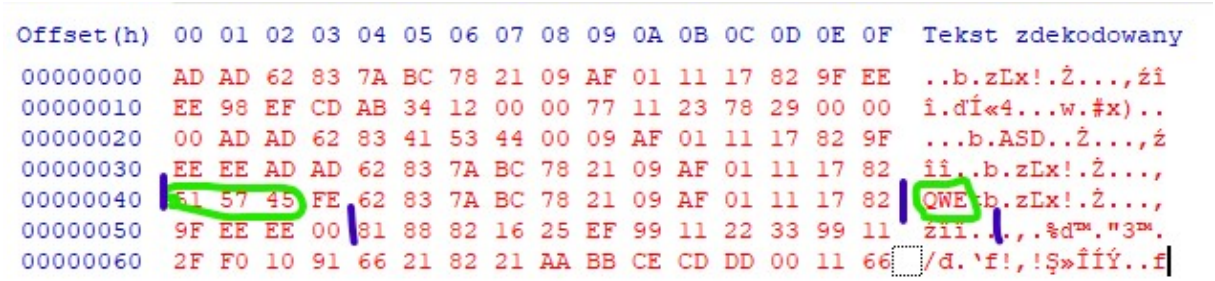
printf(napis); //wypisze QWE

Tak jak mówiłem nazwa tablicy jest wskaźnikiem na nią. Zobaczmy jaki ty przyjmuje printf w pierwszym parametrze:

Wskaźnik na chara. Dzięki temu 0 na końcu printf wie kiedy ma zatrzymać się i przestać wypisywać kolejne znaki z pamięci.

Zobaczmy co się stanie jak go zabraknie:

char napis[] = {‘Q’,’W’,’E’};



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Tekst zdekodowany
00000000	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	82	9F	EE	..b.zLx!.Ż...,źî
00000010	EE	98	EF	CD	AB	34	12	00	00	77	11	23	78	29	00	00	î.dÍ«4...w.#x)..
00000020	00	AD	AD	62	83	41	53	44	00	09	AF	01	11	17	82	9F	...b.ASD..Ż...,ź
00000030	EE	EE	AD	AD	62	83	7A	BC	78	21	09	AF	01	11	17	82	îî..b.zLx!.Ż...,
00000040	51	57	45	FE	62	83	7A	BC	78	21	09	AF	01	11	17	82	QWEb.zLx!.Ż...,
00000050	9F	EE	EE	00	81	88	82	16	25	EF	99	11	22	33	99	11	źîî...,.%d™."3™.
00000060	2F	F0	10	91	66	21	82	21	AA	BB	CE	CD	DD	00	11	66	/đ.‘f!,!Ş»ÎÍÝ..f

Na zielono zaznaczono nasz string. Na niebiesko zaznaczono to co zostanie wyświetlone. printf i std::cout poleci po pamięci do napotkania pierwszego bajtu 0x00. Z tego biorą się często u początkujących „Krzaczki” na ekranie.