시스템 프로그래밍1

S-개발자 4기 2026-03-10(화)

서울 송파구 동남로 130, 2층 제 4강의실

대표이사 전상현

크로스 플랫폼 핵심 모듈 설계의 기술
모바일, 리눅스, 맥, 윈도우를 아우르는 C++ 라이브러리 구축 바이블
전상현 지음

크로스 플랫폼 핵심 모듈 설계의 기술

모바일, 리눅스, 맥, 윈도우를 아우르는
C++ 라이브러리 구축 바이블

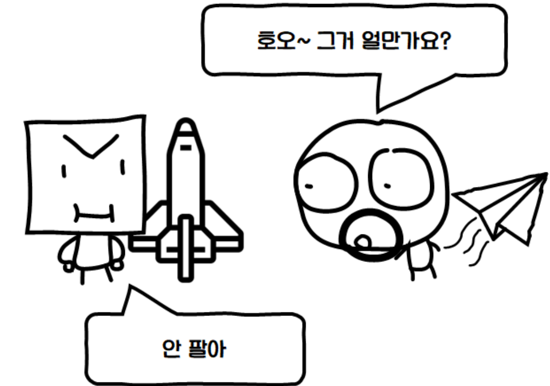
전상현 지음

들어가기 전에 수업 개요

과정명	시스템 프로그래밍	강사명	전상현
강의시간	24시간 (4일 x 6시간)		
강의목표	컴퓨터 시스템을 이해한다. 나아가 다양한 시스템을 제어하는 C++ 프로그래밍 기술을 익힌다.		
평가방식	시험 50%, 실습 50%		

강의내용

시간(H)	제목	내용
1H	오리엔테이션	강사 소개 후 간단한 퀴즈와 함께 실행파일의 구조를 이해한다.
2H	변수형과 유니코드	시스템에 존재하는 모든 종류의 변수형을 알아본다. 다국어를 표현하는 문자체계인 유니코드를 이해한다.
3H	개발환경 구축하기	윈도우/리눅스(wsl)에 빌드할 수 있는 환경을 구성한다. 기본적인 개발툴 사용법을 익힌다.
2H	문자열 정복하기	C++과 친해지기 위한 코드를 작성해본다. VisualStudio 디버깅 기술을 배운다.
2H	C++ 컴파일러/링커/디버거 정복하기	유니코드 상호 변환하는 함수를 이해한다. 문자열을 자유자재로 다루는 방법을 배운다.
2H	파일시스템 정복하기	플랫폼별 파일/디렉토리 접근 함수를 이해한다. 파일 시스템을 자유자재로 다루는 방법을 배운다.
2H	데이터 정복하기	STL 자료구조를 이해하고, 실무 응용 기술을 배운다. XML/JSON/INI 등을 자유자재로 다루는 방법을 배운다.
2H	메모리 정복하기	C++의 꽃, 스택 메모리의 장단점을 알아본다. 4가지 영역의 메모리를 자유자재로 활용한다.
2H	소켓 정복하기	UDP/TCP 통신, DATAGRAM/STREAM의 차이를 이해한다. 소켓 통신 시스템을 자유자재로 다루는 방법을 배운다.
6H	**최종실습**	**앞에서 배운 지식과 기술을 활용하여 미니 프로젝트를 진행한다.**



코딩 실력이 곧 우주선이다.
우주같이 광활한 컴퓨터 세계로 여행을 떠나자.

<참고서적>

크로스플랫폼 핵심모듈 설계의 기술(2018) – 전상현, 로드북
아무도 알려주지 않은 C++ 코딩의 기술 (2023) – 전상현, 로드북

변수형

기본적으로 제공하는 변수형

C언어나 C++이나 기본적으로 제공되는 변수형들이 존재합니다. 그것들은 다양한 크기와 형태로 존재하는데 각 플랫폼에서 지원하는 형태가 완벽하게 동일하지 않습니다. 기본적으로 지원되는 변수들이 무엇이 있고, 각 형태와 크기는 어떠한 지 한 번쯤 은 알아 둘 필요가 있습니다.

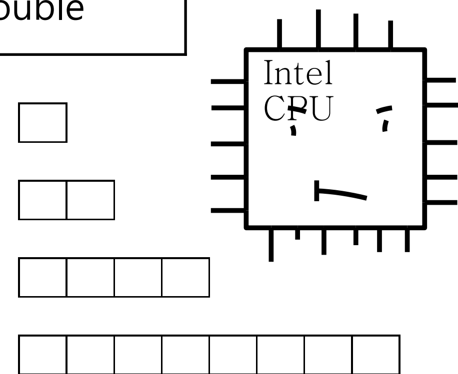
변수들은 다양한 크기에 따라 부호가 있는 것과 없는 것으로 구분할 수 있습니다. 그리고 숫자를 표현하는 방식이 정수형이냐 실수형이냐를 구분할 수 있습니다. 여기에 포인터형과 const형, 문자형이 추가적으로 존재합니다.

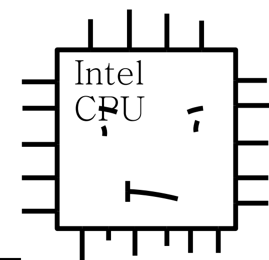
처음 C언어를 공부하면 가장 먼저 배우는 것이 기본 변수형입니다. 기본 변수형은 표현하는 숫자의 범위와 메모리 크기에 따라 달라집니다. 숫자의 범위에 따라서 정수형과 실수형으로 나뉘고, 부호가 있는 것과 없는 것으로 나뉩니다. 변수가 차지하는 메모리는 1바이트부터 길면 8바이트까지 다양하게 존재합니다.

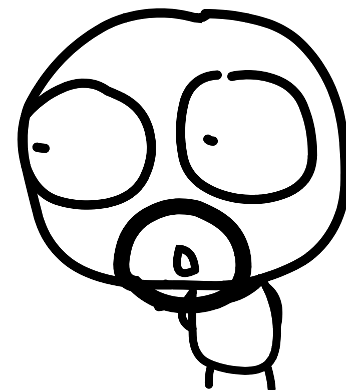
	1바이트	2바이트	4바이트	8바이트
부호 있는 정수형	signed char	short	int	long long
부호 없는 정수형	unsigned char	unsigned short	unsigned int	unsigned long long
실수형			float	double

C언어에서 기본적으로 제공하는 키워드로 각각의 변수형을 정의하면 위와 같습니다.

여기서 QUIZ. 3바이트나 5,6,7바이트 변수는 왜 존재하지 않을까요?







long 변수에 대해서

혹시 여러분은 int형 변수가 몇 바이트라고 생각하는지요? 직전에 4바이트 변수군에 int형을 넣었으니 당연히 4바이트라고 알고 있는 사람도 있을 것이고, 아닌데 하면서 고개를 갸우뚱하셨던 분도 있을 것입니다.

정확히 int형은 CPU가 연산하기 가장 쉬운 메모리를 가지도록 표준화된 변수입니다. 요즈음의 프로세서는 64비트이지만 그 전에는 32비트가 대세였고, 또 그 전에는 16비트, 8비트였던 때도 있었습니다. 8비트까지 거슬러 올라가기에는 너무 멀어서 16비트 시절까지만 내려가보더라도 그 당시의 int는 2바이트였습니다.

근래에는 32비트 프로세서에서 int형이 4바이트로 조정되었지만, 재미있는 것은 64비트 프로세서에서는 8바이트로 재조정되지는 않았다는 점입니다. 정확한 이유는 알 수 없지만 필자가 예상한 바는 기존에 작성된 수많은 코드들의 호환성 문제 때문인 것으로 보입니다. 그래서 종전의 C언어나 C++문법 책에서는 int변수형의 크기를 CPU 연산 크기에 따라 따르다고 정의했던 반면 요즈음에는 4바이트로 고정하여 설명하고는 합니다.

비슷한 변수로 long형 변수가 있습니다. 이 변수는 int보다도 더 혼동됩니다. 아마 주변의 개발자들에게 long 변수가 몇 바이트인지 질문해본다면 아마도 서로 다른 답변을 내놓는다는 것을 알 수 있을 것입니다. long 형의 정확한 크기는 32비트 이하의 프로세서에서는 4바이트, 그리고 64비트 프로세서에서는 8바이트입니다. 그러나 코딩으로 직접 실제 long 변수의 크기를 확인해본 개발자들은 이 또한 틀린 이야기라고 반박합니다.

간단히 아래와 같은 코드를 구현해서 실행해봅니다.

```
std::cout << sizeof(long) << std::endl;
```

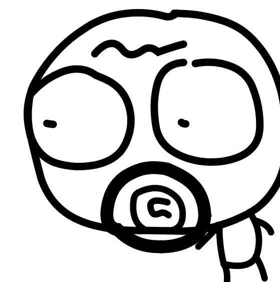
```c
#define SHOW_SIZEOF(x)\
    printf("sizeof("#x") is %u\n", sizeof(x));

int main(int argc, char* argv[])
{
    printf("Pointer size is %u\n", sizeof(void*));
    SHOW_SIZEOF(char);
    SHOW_SIZEOF(short);
    SHOW_SIZEOF(int);
    SHOW_SIZEOF(long);
    SHOW_SIZEOF(long long);
    SHOW_SIZEOF(float);
    SHOW_SIZEOF(double);
    return 0;
}
```

32비트 윈도우 실행결과	64비트 윈도우 실행결과
Pointer size is 4	Pointer size is **8**
sizeof(char) is 1	sizeof(char) is 1
sizeof(short) is 2	sizeof(short) is 2
sizeof(int) is 4	sizeof(int) is 4
sizeof(long) is 4	sizeof(long) is 4
sizeof(long long) is 8	sizeof(long long) is 8
sizeof(float) is 4	sizeof(float) is 4
sizeof(double) is 8	sizeof(double) is 8
계속하려면 아무 키나 누르십시오 . . .	계속하려면 아무 키나 누르십시오 . . .

32비트 맥 실행결과	64비트 맥 실행결과
Pointer size is 4	Pointer size is **8**
sizeof(char) is 1	sizeof(char) is 1
sizeof(short) is 2	sizeof(short) is 2
sizeof(int) is 4	sizeof(int) is 4
sizeof(long) is 4	sizeof(long) is **8**
sizeof(long long) is 8	sizeof(long long) is 8
sizeof(float) is 4	sizeof(float) is 4
sizeof(double) is 8	sizeof(double) is 8
Program ended with exit code: 0	Program ended with exit code: 0

크워워~(놀라는 척)
근데 알아 둔다고
쓸모가 있나유?



long 변수에 대해서 cont'

C++은 표준을 따르는 언어인데 어째서 플랫폼에 따라 이런 차이가 생기게 된 것일까요? 정확한 것은 아니지만 필자는 그 원인을 추정할 수 있었습니다. 그것은 바로 윈도우의 DWORD 타입 때문입니다.

```
typedef unsigned long DWORD;
```

위의 정의가 윈도우 API 내부에서 정의된 DWORD 변수입니다. DWORD는 특히나 윈도우 API에서 가장 흔히 쓰이는 변수인데, 4바이트를 가정하고 사용하는 값입니다. 이런 상황에서 프로세스가 64비트일 때 long 또한 8바이트로 변경하면 DWORD 값도 같이 크기가 변경되어 아마도 전세계의 많은 코드들에서 상상초월의 문제가 발생하게 될 것입니다. 그 이유로 윈도우 진형에서는 쉽사리 long 변수를 표준대로 적용할 수 없었을 것입니다. 윈도우의 뒤늦은 처사라고 생각되는 아래의 변수들을 보면 한 번 더 이러한 의심을 확신할 수 있습니다.

```
typedef unsigned int DWORD32;
typedef unsigned long DWORD64;
```

DWORD라는 변수 자체가 32비트인데, DWORD32와 DWORD64라는 새로운 정의를 해뒀습니다. 게다가 DWORD32는 long형이 아닌 int형입니다. 만약에 처음 DWORD를 정의할 때 long이 아닌 int를 사용했더라면 이와 같은 예외는 두지 않았을 것이라 생각됩니다.

정리하자면 long 변수는 운영체제와 프로세서에 따라 다음과 같이 크기가 달라집니다.

	32비트 머신의 long 크기	64비트 머신의 long 크기
윈도우 운영체제	4 byte	4 byte
유닉스 운영체제	4 byte	**8 byte**

개념적 변수형

사실상 변수형을 정의할 때 중요한 것은 크기와 형식입니다. 변수 이름으로 크기와 형식을 직관적으로 표현하는 것이 도움이 될 경우가 있습니다. 다음의 변수형 선언이 그렇습니다.

	1바이트	2바이트	4바이트	8바이트
부호 있는 정수형	INT8	INT16	INT32	INT64
부호 없는 정수형	UINT8	UINT16	UINT32	UINT64

또한, CPU 관점에서 변수를 표현하는 경우도 있습니다. 모두 앞에서 정의한 변수형의 이름만 바꿔 다시 정의한 변수들입니다.

	1바이트	2바이트	4바이트	8바이트
부호 없는 정수형	BYTE	WORD	DWORD	QWORD

포인터를 정의하는 변수형도 있습니다. INT8*보다 LPINT8로 쓰는 것이 가독성이 더 높습니다.

	1바이트	2바이트	4바이트	8바이트
부호 있는 정수형	LPINT8	LPINT16	LPINT32	LPINT64
부호 없는 정수형	LPUINT8	LPUINT16	LPUINT32	LPUINT64
부호 없는 정수형	LPBYTE	LPWORD	LPDWORD	LPQWORD
실수형			LPFLOAT	LPDOUBLE

한편으로는 포인터를 가리키는 접두어가 단순 P가 아니라 LP인지 궁금하기도 합니다. CPU 발전의 역사와 관련이 있는데, 과거 16비트 시절에 더 넓은 메모리를 사용하고 주소를 지정하기 위해 NearPointer(NP)와 LongPointer(LP)로 구분하던 것에서 유래되었습니다. 이제는 구분이 필요하지 않지만 과거의 호환성 문제도 있고 포인터를 P로 나타냈을 경우 생각보다 가독성이 떨어진다는 의견이 있어 계속 LP라는 표기법으로 사용되고 있는 것입니다.

```
typedef POINT* PPOINT;
```

☹ 그다지 가독성에 좋지 않다.

C언어에서 const 선언은 여러모로 중요한 역할을 합니다. 함수 인자에 const로 선언된 인자는 함수 내부에서 변경할 수 없는 읽기 전용 값이라는 것을 나타냅니다. 그 말은 입력 값임을 명시하는 뜻이기도 합니다. 다음 함수처럼 포인터형 변수는 함수가 변형하는 출력 값이거나 입력 값일 수 있는 상황에서 const는 그 것을 확실하게 구분해줍니다.

```
void* memcpy(void* pDest, const void* pSrc, size_t tSize);
```

축약형으로는 C로 나타내며 포인터와 함께 쓰이면 LPC가 됩니다. 단순한 접두어로 복잡한 표현식을 나타낼 수 있어 용이하고 익숙해지면 높은 가독성을 제공합니다.

	1바이트	2바이트	4바이트	8바이트
부호 있는 정수형	LPCINT8	LPCINT16	LPCINT32	LPCINT64
부호 없는 정수형	LPCUINT8	LPCUINT16	LPCUINT32	LPCUINT64
부호 없는 정수형	LPCBYTE	LPCWORD	LPCDWORD	LPCQWORD

	일반형	포인터형	const 포인터형
void	VOID	LPVOID	LPCVOID

문자 변수형

앞서 정의한 포인터형과 상수형은 문자열 표현에 자주 쓰입니다. 아래의 표현은 반드시 기억해두길 바랍니다.

	일반형	포인터형	const 포인터형
char	CHAR	LPSTR	LPCSTR
wchar_t	WCHAR	LPWSTR	LPCWSTR

간혹 잘 쓰이지는 않지만 MBS, WCS 표기를 이용한 문자열 표기법도 있습니다.

	일반형	포인터형	const 포인터형
char	MBS	LPMBS	LPCMBS
wchar_t	WCS	LPWCS	LPCWCS

유닉스 계열 변수형

개념은 갖지만 소문자를 지향하는 유닉스 계열 개발자들에게 익숙한 네이밍입니다.

여기서 잠깐, 윈도우 개발자들은 왜 대소문자를 쓰고 유닉스 계열 개발자들은 소문자를 쓸까요?

	1바이트	2바이트	4바이트	8바이트
부호 있는 정수형	int8	int16	int32	int64
부호 없는 정수형	uint8	uint16	uint32	uint64

	1바이트	2바이트	4바이트	8바이트
부호 없는 정수형	byte	word	dword	qword

	1바이트	2바이트	4바이트	8바이트
부호 있는 정수형	pint8	pint16	pint32	pint64
부호 없는 정수형	puint8	pulnt16	puint32	puint64
부호 없는 정수형	pbyte	pword	pdword	pqword
실수형			pfloat	pdouble

	1바이트	2바이트	4바이트	8바이트
부호 있는 정수형	pcint8	pcint16	pcint32	pcint64
부호 없는 정수형	pcuint8	pcuint16	pcuint32	pcuint64
부호 없는 정수형	pcbyte	pcword	pcdword	pcqword

	일반형	포인터형	const 포인터형
void	void	pvoid	pcvoid

	일반형	포인터형	const 포인터형
char	char	pstr	pcstr
wchar_t	wchar	pwstr	pcwstr

	일반형	포인터형	const 포인터형
char	mbs	pmbs	pcmbs
wchar_t	wcs	pwcs	pcwcs

유니코드

?댑씼踰꾩뒿 媛€밤 < ???섑린

?대ㅓ ?곶????꾬퀎吏€瑗섏꽌 ?겹— 吏뫼븞???죮옜???€??맷 < 留맷???二쉽꿂??
?댑씼踰꾩뒿 媛€??밤 < 媛€씽땜??寃껎? ?대쁜 寃껎쓿 ?섎??섎뒙媛€
洹몃—怨?臾댁뽉??吏€ 쒿빞留??섎뒙媛€??€寃껎뢿.
洹??ъ땜??留맷???二쉽땡 ?꾬퀎吏€??좎븞踰꾩?猿섏꽌 留맷???二쉽꿂怨?
洹??좎븞踰꾩???怨좎“遺€瑗섏꽌 留맷???二쉽꿂?ㅺ퀬 ?씽땜.

?댑씼踰꾩뒿 媛??ъ궦???랁”?댐땜.
遺?ъ쌭 ?섎뒙 ?몄뱻??留룡븸寃좎?.
洹몃윮???섏뽇寃??댑뺄??洹멸퀬??ъ냼??寃껎뽇 遺댁넉?섏땜.
?댑씼踰꾩뒿 媛€??쒿뺄?ㅻ땜??寃껎?, 媛€대? 吏€ㅿ뒙 以뮔???梨깜엥??吏€??댐땜??寃껎씼湲겹뢿 ?댐땜.
洹몃옜, 洹?4媛쒿쓿 ?⑹꽉??吏€⑤땜??梨깜엥?꾫€€
?⑹꽉?€댑씼踰꾩뒿 媛€?珥댐? ?밤 < 媛€몃IX遺€댄궎??媛닿뎽?맷씤 踰좎븯?명뱶 ?븯뽇寃?謏쏌? 寃껎씼硫?

洹??댄쌘 ?€€??겹— 吏뫼븞??怨꾩뒣?섏봽??

?숇갑?맷꽌 ?꾬옜?????뜻? ?븯땜??湲댑쟼?곹씤 媛€섎뢿 ?겟뼢?ㅻ땜.
洹몃윮??洹??댁긷?쇰죿, ?④꺼???댐뒙 嫄겹????섏뽇 ?섎?媛€댐땜.

吏€댐???200???? 踰좎빘?명뱶 ?븯냭 ?④퍡 ?맷젙???섏켭
?€댑닏???ㅼ쭒 ?ㅼ죿?ㅽ룯?쒿? '?쇰땶?ㅼ씀 臾???遊됱씤?섏린 ?꾩빞
???⑹꽉???섏쓿 ?댁숏?댐땜怨??씽땜.
?좎? ?밤 < ?????꾬퀎吏€??대젆寃?留맸벱??

"?섏뽇???좎뢿 ?꺾뢿 ?놀땜. 寃€????맵? ?좎벐????寃껎씤吏€?깃벐???€吏€?寃껎씤吏€??뱤븯??寃껎씼??
洹몃젆湲??뚮ㅽ?? 踰좎빘?명뱶 ?븯? 臾댁뽉蹂대땜???좎:?????댐뒙 湲겟궓?댁뿭 ?댑퇋??移쒿뎽?닿린???댐뚪
?댑씼踰꾩뒿 媛€?珥댐? ?밤 < ?맵퀣 ?⑹꽉??留↔릿 寃껎씼?쇳퀬.

인코딩(N) 언어(L) 설정(T) 도구(O) 매크로 실행 플러그인 창 관리 ?

ANSI로 표시
● UTF-8 (BOM 없음)로 표시
UTF-8로 표시
UCS-2 빅 엔디언으로 표시
UCS-2 리틀 엔디언으로 표시
글자 집합 >
ANSI로 변환
UTF-8 (BOM 없음)로 변환
UTF-8로 변환
UCS-2 빅 엔디언으로 변환
UCS-2 리틀 엔디언으로 변환

아랍어 >
발트어 >
켈트어 >
키릴어 >
중유럽어 >
중국어 >
동유럽어 >
그리스어 >
히브리어 >
일본어 >
한국어 >
북유럽어 >
태국어 >
터키어 >
서유럽어 >
베트남어 >

Windows 949
EUC-KR

ASCII 코드와 EUCKR, 멀티바이트 스트링

유니코드란, 쉽게 말해 전세계의 다양한 언어들을 저장하거나 표현할 수 있도록 구축한 문자 코드 체계라고 생각하시면 됩니다. 아직 그 개념을 잘 모르는 분들이라면 이번 기회에 확실히 개념을 잡고 넘어 가시길 바라겠습니다.

우리가 가장 많이 쓰는 특수기호나 숫자, 알파벳은 다음과 같은 바이트 코드와 매핑되어 있습니다.

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x00	NUL	SOH	STX	ETX	EOT	ENQ	ANK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0x10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0x20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

이 코드는 ASCII(American Standard Code for Information Interchange)라고 불리며 아마도 대부분의 컴퓨터 전공자들은 익히 알고 있을 문자 체계일 것입니다. 하지만 축약어에서도 알 수 있듯이 알파벳을 사용하는 영어권 국가를 위한 것입니다.

그렇다면 우리가 웹에서 밥 먹듯이 읽고 쓰는 한글은 어떻게 표현할 수 있을까요? 당연히 알파벳처럼 한글도 각각의 문자별 고유의 값을 매겨 사용합니다. 대신 한글은 영어에 비해 문자의 종류가 더 많아서 대소문자 총 52개인 영어와 달리 11172자의 문자를 매핑해 두고 사용합니다. 이것은 1바이트로 표현할 수 있는 최대 256개를 초과하기 때문에 2바이트로 확장하여 표현해야 합니다. 2바이트는 최대 65536개를 표현할 수 있어서 모든 한글 문자를 넣을 수 있습니다.

하지만 무턱대고 한글을 0~65535 사이의 아무 값으로 매핑시키면 기존의 ASCII 코드와의 충돌을 피할 수 없습니다. 따라서 먼저 지정된 문자 코드와 중복되지 않도록 배치해야 합니다. ASCII에서 사용하는 문자들은 0xF7까지만 정의돼 있어 0x80 이후의 값을 사용하면 겹치지 않습니다. 그래서 한글의 경우에는 두 바이트 중 첫번째 바이트 값을 0x80이 초과되는 값으로 선언하게 했습니다. 일단 0x80이 초과되는 문자가 발견되면 ASCII는 아니므로 이후의 한 바이트까지 같이 하나의 문자로 처리하게 만듦으로써 ASCII로 작성된 문자들 사이에 한글이 끼어들더라도 혼동되지 않게 구분할 수 있습니다.

CODE	48	65	6C	6C	6F	2C	20	C7	D1	B1	DB	21	0
CHAR	H	e	l	l	o	,		한		글		!	₩0

이렇게 기존의 ASCII체계에 한글 코드를 추가한 것이 EUC-KR(Extended Unix Code KoRean)입니다.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0xB0A0		가	각	간	갇	갈	갉	갊	감	갑	값	갓	갔	강	갖	갗
0xB0B0	같	갚	갛	개	객	갠	갤	갬	갭	갯	갰	갱	갸	갹	갼	걀
0xB0C0	걋	걍	걔	걘	걜	거	걱	건	걷	걸	걺	검	겁	것	겄	겅
0xB0D0	겆	겉	겊	겋	게	겐	겔	겜	겝	겟	겠	겡	겨	격	겪	견
0xB0E0	겯	결	겸	겹	겻	겼	경	곁	계	곈	곌	곕	곗	고	곡	곤

마찬가지로 ASCII에 중국어를 추가한 것은 EUC-CN, 일본어를 추가한 것은 EUC-JP, 대만어를 추가한 것은 EUC-TW입니다. 이렇게 자국에서 쓰이는 문자를 추가로 만들어 코드화한 문자열을 멀티바이트 스트링(Multi-Byte String)이라고 부릅니다.

코드 페이지(Code page)와 유니코드 문자 집합(Character set)

Code Page	내용	Code Page	내용
437	원래의 IBM PC 코드 페이지	862	히브리어
737	그리스어	866	키릴 자모 알파벳
850	다중 언어 (라틴 1) (서양 유럽 언어)	869	그리스어
852	슬라브어 (라틴 2) (동유럽 언어)	932	일본어 지원
855	키릴 자모 알파벳	936	GBK 중국어 간체자 지원
857	터키어	949	한국어 지원
858	다중 언어	950	중국어 번체자 (대만) 지원
860	포르투갈어	1200	UCS-2LE 유니코드 little-endian
861	아이슬란드어	1201	UCS-2BE 유니코드 big-endian
863	프랑스어 캐나다어	65000	UTF-7 유니코드
865	북유럽어	65001	UTF-8 유니코드

앞서 설명 드린 확장된 유닉스 코드(EUC, Extended Unix Code)인 멀티바이트 스트링은 단점이 있습니다. 다름 아닌 둘 이상의 언어를 혼용해서 사용할 수 없다는 점입니다.

각 국가의 문자들을 자신들만의 방식으로 코드를 지정하다 보니 같은 코드에 서로 다른 국가의 문자가 할당되기도 했습니다.

그래서 멀티바이트 문자인 경우에는 코드 페이지라는 개념이 필요하게 됩니다. 같은 바이트 스트림이라도 코드 페이지에 따라 문자의 해석이 달라지는 것이지요.

유니코드 | 코드 페이지(Code page)와 유니코드 문자 집합(Character set)

이 문제를 해결하기 위해 등장한 것이 전세계의 모든 문자를 고유한 값으로 매핑하는 유니코드 문자 집합니다. 이 문자 집합들로 문자열을 표현하면 그 어떤 언어라도 섞어서 표현하는 것이 가능해집니다. 그래서 EUC-KR에서 사용한 한글 고유 값도 유니코드 문자 집합에서는 새로운 값으로 재정의 되었습니다.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0xAC00	가	각	갂	갃	간	갅	갆	갇	갈	갉	갊	갋	갌	갍	갎	갏
0xAC10	감	갑	값	갓	갔	강	갖	갗	갘	같	갚	갛	개	객	갞	갟
0xAC20	갠	갡	갢	갣	갤	갥	갦	갧	갨	갩	갪	갫	갬	갭	갮	갯
0xAC30	갰	갱	갲	갳	갴	갵	갶	갷	갸	갹	갺	갻	갼	갽	갾	갿
0xAC40	걀	걁	걂	걃	걄	걅	걆	걇	걈	걉	걊	걋	걌	걍	걎	걏
0xAC50	걐	걑	걒	걓	걔	걕	걖	걗	걘	걙	걚	걛	걜	걝	걞	걟
0xAC60	걠	걡	걢	걣	걤	걥	걦	걧	걨	걩	걪	걫	걬	걭	걮	걯
0xAC70	거	걱	걲	걳	건	걵	걶	걷	걸	걹	걺	걻	걼	걽	걾	걿

EUC-KR에서의 한글 `가`의 값은 B0 A1였던 반면, 유니코드에서는 AC 00가 된 것입니다. 각국 언어의 코드가 재편성됨에 따라 코드를 값을 매핑하는 조견표도 마이크로소프트에서 제공했습니다.

EUC-KR과 유니코드에 이어 CP-949라는 한글체계도 있습니다. 949는 앞에서 본 것처럼 한글의 코드 페이지 번호이므로 유니코드와는 관계없이 EUC-KR의 확장이라고 보시면 됩니다. 단, EUC-KR과의 차이점은 기존에는 표현되지 않은 문자가 추가되었다는 점입니다. 다음 표에서 나타난 문자들은 EUC-KR에 포함되지 않은 CP-949에 새롭게 정의된 것들입니다.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x8140		갂	갃	갅	갆	갋	갌	갍	갎	갏	갘	갞	갟	갡	갢	갣
0x8150	갥	갦	갧	갨	갩	갪	갫	갮	갲	갳	갴					
0x8160		갵	갶	갷	갺	갻	갽	갾	갿	걁	걂	걃	걄	걅	걆	걇
0x8170	걈	걉	걊	걌	걎	걏	걐	걑	걒	걓	걕					
0x8180		걖	걗	걙	걚	걛	걝	걞	걟	걠	걡	걢	걣	걤	걥	걦
0x8190	걧	걨	걩	걪	걫	걬	걭	걮	걯	걲	걳	걵	걶	걹	걻	걼
0x8200	걽	걾	걿	겂	겇	겈	겎	겏	겑	겒	겓	겕	겖	겗	겘	겙

유니코드 인코딩

유니코드 문자 집합이 준비되었지만 그것을 어떤 식으로 기록할 것인지는 시스템적인 문제이기 때문에 몇 가지 인코딩이 추가로 고안되었습니다. 어디선가 많이 들어봤을 만한 이름들인데 자세한 내용을 몰랐다면 이 기회에 한 번 정리해보십시오.

인코딩 이름	문자당 바이트 수	비고
UTF8	가변	유닉스계열 char
UTF16 or UCS2	2바이트	윈도우 wchar_t
UTF32 or UCS4	4바이트	유닉스계열 wchar_t
UTF16 BE	2바이트	빅엔디안 시스템
UTF32 BE	4바이트	빅엔디안 시스템

빅엔디안(Big-Endian)은 여러 바이트로 이루어진 값을 표기시 높은 자리의 값의 바이트를 제일 왼쪽에 배치한다는 뜻입니다. 즉 16진수 값으로 0xABCD라는 값이 있으면 이것을 바이트로 표기할 때 AB CD 순으로 배치하면 빅엔디안이 됩니다. 반대로 CD AB로 배치하면 리틀엔디안(Little-Endian)이 되는 것이고요. 일반적으로 쓰이는 대부분의 플랫폼은 리틀엔디안이기 때문에 별다른 언급이 없다면 리틀엔디안으로 생각하시면 됩니다.

이어서 각 인코딩에 대해서 알아보겠습니다. **UTF8**(Universal Coded Character Set + Transformation Format 8-bit)은 자주 쓰이는 문자열이 알파벳일 경우 기록될 바이트의 손실을 최소화하기 위해 고안된 인코딩 기법입니다. 가변 길이인 관계로 알고리즘이 조금 복잡한 것이 단점입니다. 아래는 인코딩 규칙에 관한 설명입니다.

코드 범위	UTF8 표현(이진법)	비고
0000-007F	0xxxxxxx	ASCII와 동일한 범위
0080-07FF	110xxxxx 10xxxxxx	첫 바이트는 110으로 시작하고, 다음 바이트는 10으로 시작함
0800-FFFF	1110xxxx 10xxxxxx 10xxxxxx	첫 바이트는 1110으로 시작하고, 나머지 바이트들은 10으로 시작함

위에 나열된 범위를 넘어가는 코드의 경우에는 추가적으로 고안된 규칙이 있는데, 한글을 표현하기에는 3바이트면 충분하기 때문에 여기서는 이정도만 나열하기로 합니다. 자세한 정보는 웹이나 기타 문헌 자료를 통해 보완하시면 됩니다.

UTF16 방식은 고정된 2바이트의 값만 사용하므로 복잡한 인코딩 과정이 생략됩니다. 유니코드 문자 집합을 그대로 기입하여 사용합니다.

앞서 한글 '가'는 고유 코드가 AC 00이라고 했었는데, 리틀엔디안으로 표기할 때는 00 AC과 같이 됩니다. 만일 코드 값이 FF FF를 넘어가는 경우에 서러게이트(Surrogate)라는 규칙으로 연속된 두 16바이트를 하나의 문자셋으로 구분해 쓰입니다.

UTF32 방식은 UTF16과 동일한 대신, 고정된 4바이트를 사용합니다. 사람이 인식할 때는 비트보다는 바이트 단위가 더 편할 수도 있어서 **UCS4**(Universal Character Set 4-byte)라고도 표현합니다. 마찬가지로 **UTF16**은 UCS2이구요.

2바이트 유니코드 문자를 UTF8로 표현해보면 다음 그림과 같습니다.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1
유니코드 A
0x0041

0 1 0 0 0 0 0 1
7 6 5 4 3 2 1 0
UTF-8 A
0x41

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0
유니코드 π
0x03C0

1 1 0 0 1 1 1 1
7 6 5 4 3 2 1 0
1 0 0 0 0 0 0 0
7 6 5 4 3 2 1 0
UTF-8 π
0xCF 0x80

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 1 0 1 0 1 1 1 0 0
유니코드 한
0xD55C

1 1 1 0 1 1 0 1
7 6 5 4 3 2 1 0
1 0 0 1 0 1 0 1
7 6 5 4 3 2 1 0
1 0 0 1 1 1 0 0
7 6 5 4 3 2 1 0
UTF-8 한
0xED 0x95 0x9C

BOM(Byte Order Marker)이란?

파일에 저장된 문자 데이터가 어떤 인코딩인지 판별하기 위해 나타내는 표식입니다. 파일의 가장 첫머리에 존재합니다.

인코딩 이름	문자당 바이트 수	BOM	비고
UTF8	가변	EF BB BF	리눅스 char
UTF16 or UCS2	2바이트	FF FE	윈도우 wchar_t
UTF32 or UCS4	4바이트	FF FE 00 00	유닉스계열 wchar_t
UTF16 BE	2바이트	FE FF	빅엔디안 시스템
UTF32 BE	4바이트	00 00 FE FF	빅엔디안 시스템

고생 많으셨습니다!