시스템 프로그래밍3

S-개발자 4기 2026-03-11(수)

서울 송파구 동남로 130, 2층 제 4강의실

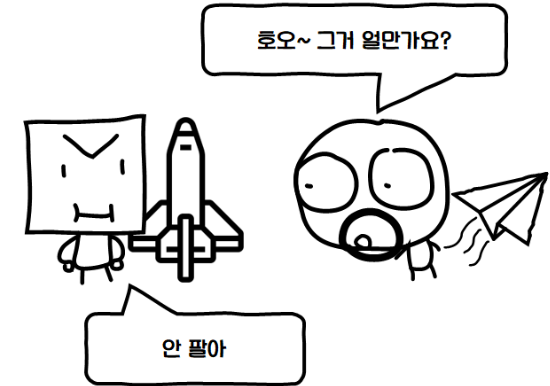
대표이사 전상현

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>

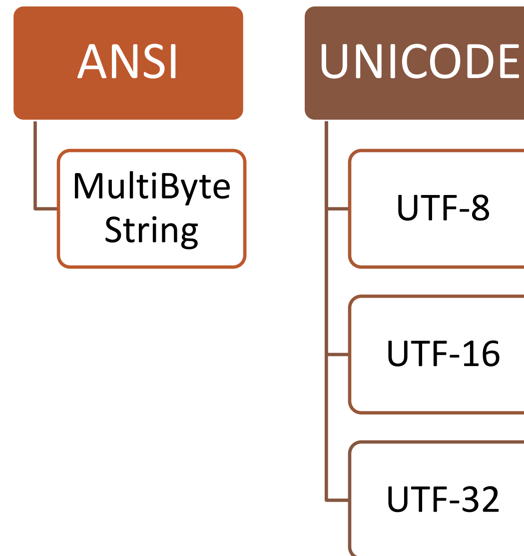
크로스플랫폼 핵심모듈 설계의 기술(2018) – 전상현, 로드북
아무도 알려주지 않은 C++ 코딩의 기술 (2023) – 전상현, 로드북

유니코드 정복하기

문자셋과 인코딩의 차이

문자셋(Charset)은 동일한 문자를 표현하는 서로 다른 번호 체계를 뜻합니다.

유니코드 인코딩(Unicode Encoding)은 유니코드를 표현하는 서로 다른 방법입니다.



유니코드 사용하기

C++ 언어에서는 문자열 변수로 char만 알고 있는 경우가 많지만 사실 wchar_t형도 있습니다. 다음 코드는 윈도우와 유닉스 계열에서 모두 실행 가능합니다.

```
#include <stdio.h>
#include <locale>

int main()
{
    std::locale::global(std::locale("Korean"));

    char szTest1[] = "안녕하세요! ABC";
    wchar_t szTest2[] = L"안녕하세요! ABC";
    printf("%s₩n", szTest1);
    wprintf(L"%s₩n", szTest2);
    return 0;
}
```

그런데 중요한 것은 **각각의 변수형이 다루는 문자열은 플랫폼별로 종류가 서로 다르다는 점**입니다. 이 부분이 유니코드를 정복하기 위한 가장 중요한 시발점이 됩니다.

	char	**wchar_t**
윈도우	ASCII 및 멀티바이트 스트링 (1 바이트)	UTF16 (**2** 바이트)
유닉스 계열	UTF8 (1 바이트)	UTF32 (**4** 바이트)

TCHAR 타입

상용 프로그램은 기본적으로 다국어를 다룰 수 있어야 하기 때문에 유니코드로 개발합니다.

그 결과 윈도우 개발자는 문자열 변수로 wchar_t를 써야 합니다. 반면에 유닉스 계열은 char, wchar_t든 모두 유니코드를 지원하므로 효율적인 인코딩인 UTF8을 다루는 char형 변수를 사용합니다. 물론 인코딩 차이를 인지하지 않고도 처음 C언어에 입문할 때부터 배웠던 char를 그대로 사용해도 문제가 없습니다.

문제는 크로스플랫폼 상황입니다. 윈도우에서 작성한 코드를 유닉스 계열에서 그대로 쓰고 싶다면 wchar_t형도 char형도 사용해서는 안 됩니다. 그래서 고안한 변수형이 다음 TCHAR 형입니다.

```
#ifdef UNICODE

typedef wchar_t TCHAR;

#else

typdef char TCHAR;

#endif
```

이 변수형은 C++ 키워드로 존재하지 않습니다. 전처리기 UNICODE 선언 여부에 따라 컴파일러가 char나 wchar_t로 결정하는 임의로 정의한 것입니다. 그 덕에 문자열 처리 변수를 TCHAR로 작성하면 윈도우와 유닉스 계열에서 모두 사용 가능한 코드를 작성할 수 있습니다. 윈도우는 UNICODE라는 매크로를 선언해서 컴파일하고, 유닉스 계열에서는 선언하지 않고 컴파일하면 되니까요.

더불어 문자열 또한 문법이 달라져야 합니다. wchar_t 타입의 변수가 받는 문자열은 따옴표 앞에 L이라는 접두어를 붙여야 하는데, 이 또한 컴파일 시점에 같이 바꿔야만 컴파일 에러를 막을 수 있습니다.

```
#ifdef UNICODE
#define TEXT(x) L##x
#else
#define TEXT(x) x
#endif
```

이렇게 선언해두면 TCHAR 문자형에 대해서는 TEXT 문자열로 받도록 구현하면 됩니다.

```
TCHAR szBuffer[] = TEXT("Hello world!!");
```

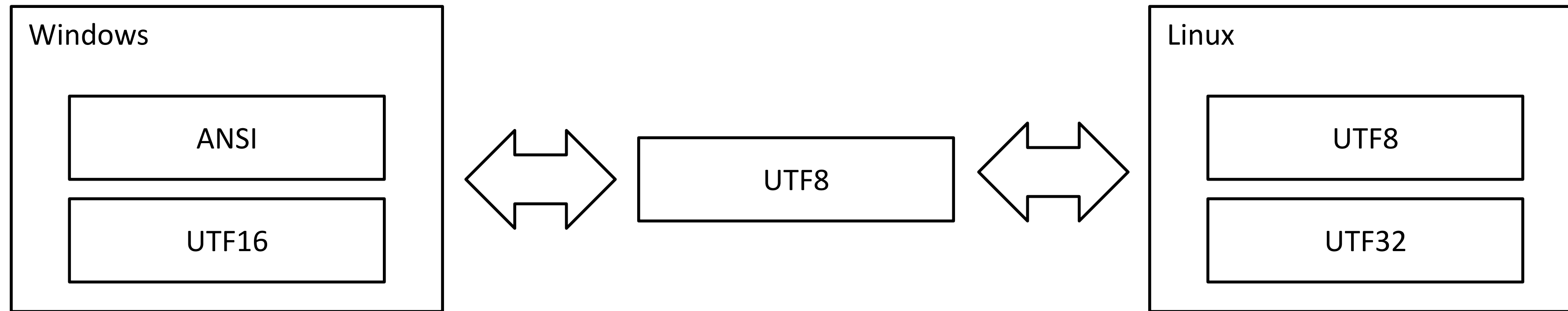
아마 TCHAR나 TEXT에 대해서 이미 알고 있는 분들도 계실겁니다. 사실 이것은 윈도우 API 내부에 선언된 매크로입니다. 유닉스 계열과는 달리 wchar_t를 자주 써야하기 때문입니다.

더불어 윈도우에서는 TCHAR와 관련된 타입도 새로 정의합니다. 앞으로 자주 보게 될 것이므로 눈에 익혀둡시다.

	일반형	포인터형	const 포인터형
윈도우 방식 정의	TCHAR	LPTSTR	LPCTSTR

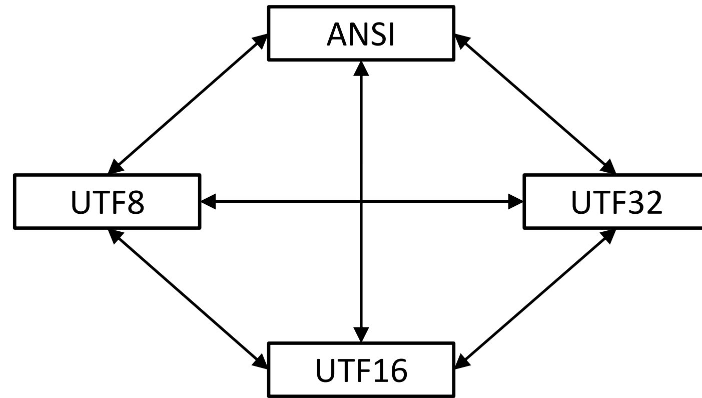
서로 다른 시스템(장치)끼리 문자열을 공유한다면,

다음은 윈도우 브라우저와 유닉스 계열 웹서버 간의 통신상황입니다. 윈도우에서 저장한 문자열은 유닉스 계열이 그대로 읽을 수 없습니다. 다루는 캐릭터셋과 인코딩 방식이 다르기 때문이죠. 따라서 통일된 규격이 필요한데 **사실상 모든 시스템은 UTF8을 공통 규격으로 정했습니다.**



QUIZ. 왜 UTF8이어야 할까요?

유니코드를 정복하기 위해 필요한 변환 함수들



플랫폼에서 주로 사용하는 인코딩 사이의 변환

ANSI -> ANSI	ANSI -> UTF8	ANSI -> UTF16	ANSI -> UTF32
UTF8 -> ANSI	UTF8 -> UTF8	UTF8 -> UTF16	UTF8 -> UTF32
UTF16 -> ANSI	UTF16 -> UTF8	UTF16 -> UTF16	UTF16 -> UTF32
UTF32 -> ANSI	UTF32 -> UTF8	UTF32 -> UTF16	UTF32 -> UTF32

인코딩이 필요한 모든 경우의 수

단 16개의 함수만 있으면 충분합니다!

실제 구현된 유니코드 변환 함수들

```
size_t ANSI_TO_ANSI(const char* pSrc, size_t tSrcCch, int nSrcCodePage, char* pDest, size_t tDestCch, int nDestCodePage);
size_t ANSI_TO_UTF16(const char* pSrc, size_t tSrcCch, int nCodePage, WORD* pDest, size_t tDestCch, size_t* ptReadSize = NULL);
size_t ANSI_TO_UTF32(const char* pSrc, size_t tSrcCch, int nCodePage, DWORD* pDest, size_t tDestCch, size_t* ptReadSize = NULL);

size_t UTF8_TO_UTF8(const char* pSrc, size_t tSrcCch, char* pDest, size_t tDestCch);
size_t UTF8_TO_UTF16(const char* pSrc, size_t tSrcCch, WORD* pDest, size_t tDestCch, size_t* ptReadSize = NULL);
size_t UTF8_TO_UTF32(const char* pSrc, size_t tSrcCch, DWORD* pDest, size_t tDestCch, size_t* ptReadSize = NULL);

size_t UTF16_TO_ANSI(const WORD* pSrc, size_t tSrcCch, char* pDest, size_t tDestCch, int nCodePage);
size_t UTF16_TO_UTF8(const WORD* pSrc, size_t tSrcCch, char* pDest, size_t tDestCch);
size_t UTF16_TO_UTF16(const WORD* pSrc, size_t tSrcCch, WORD* pDest, size_t tDestCch);
size_t UTF16_TO_UTF32(const WORD* pSrc, size_t tSrcCch, DWORD* pDest, size_t tDestCch);

size_t UTF32_TO_ANSI(const DWORD* pSrc, size_t tSrcCch, char* pDest, size_t tDestCch, int nCodePage);
size_t UTF32_TO_UTF8(const DWORD* pSrc, size_t tSrcCch, char* pDest, size_t tDestCch);
size_t UTF32_TO_UTF16(const DWORD* pSrc, size_t tSrcCch, WORD* pDest, size_t tDestCch);
size_t UTF32_TO_UTF32(const DWORD* pSrc, size_t tSrcCch, DWORD* pDest, size_t tDestCch);
```

참고로 16개가 아니라 14개인 이유는 다음 두 함수가 빠져있기 때문입니다.

- ANSI_TO_UTF8

- UTF8_TO_ANSI

각 함수는 다음의 조합으로 해결할 수 있으므로 구현에서 제외된 것입니다.

- ANSI_TO_UTF8 = ANSI_TO_UTF16 -> UTF16_TO_UTF8

- UTF8_TO_ANSI = UTF8_TO_UTF16 -> UTF16_TO_ANSI

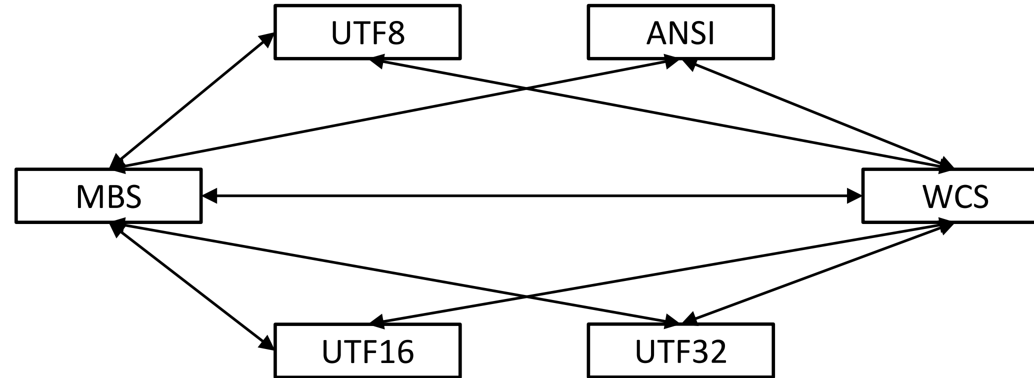
윈도우 API에서 자주 쓰이는 축약어, 반드시 외워둡시다.

줄임 표현	의미	변수형
MBS	Multi Byte String	char
WCS	Wide Character String	wchar_t
TCS	Transformable Character String	TCHAR
NTCS	Negative Transformable Character String	NTCHAR

코딩하는 입장에서 우리가 쓰는 표현은 MBS, WCS 특히 TCS를 가장 많이 쓰게 될 겁니다.
그렇다면 MBS와 WCS로 선언한 함수를 구현해야 합니다.

MBS와 WCS가 추가된 유니코드 변환 함수들

MBS와 WCS를 포함한 모든 변환 경우의 수



ANSI -> ANSI	ANSI -> UTF8	ANSI -> UTF16	ANSI -> UTF32	ANSI -> MBS	ANSI -> WCS
UTF8 -> ANSI	UTF8 -> UTF8	UTF8 -> UTF16	UTF8 -> UTF32	UTF8 -> MBS	UTF8 -> WCS
UTF16 -> ANSI	UTF16 -> UTF8	UTF16 -> UTF16	UTF16 -> UTF32	UTF16 -> MBS	UTF16 -> WCS
UTF32 -> ANSI	UTF32 -> UTF8	UTF32 -> UTF16	UTF32 -> UTF32	UTF32 -> MBS	UTF32 -> WCS
MBS -> ANSI	MBS -> UTF8	MBS -> UTF16	MBS -> UTF32	MBS -> MBS	MBS -> WCS
WCS -> ANSI	WCS -> UTF8	WCS -> UTF16	WCS -> UTF32	WCS -> MBS	WCS -> WCS

20개의 추가 함수가 필요하지만 이전에 구현했던 16(14)개의 함수를 이용하면 됩니다.

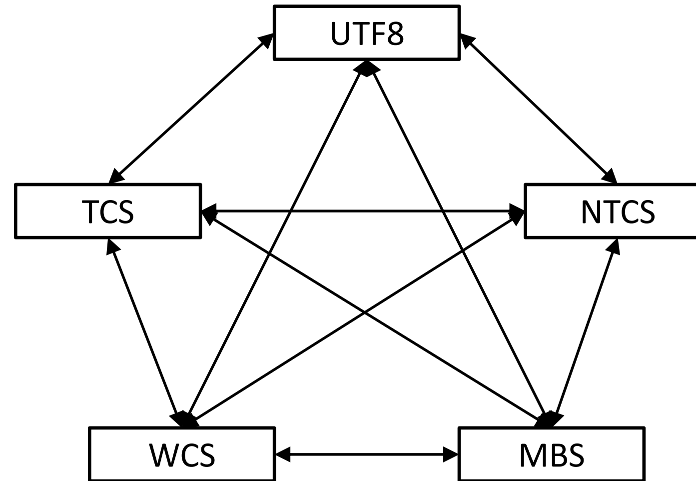
실제 구현된 유니코드 변환 함수들

```
std::string ANSIFromMBS(int nCodePage, std::string strInput);
std::string ANSIFromWCS(int nCodePage, std::wstring strInput);

std::string UTF8FromMBS(std::string strInput);
std::string UTF8FromWCS(std::wstring strInput);

std::vector<WORD> UTF16FromMBS(std::string strInput);
std::vector<WORD> UTF16FromWCS(std::wstring strInput);

std::vector<DWORD> UTF32FromMBS(std::string strInput);
std::vector<DWORD> UTF32FromWCS(std::wstring strInput);

std::string MBSFromUTF8(LPCSTR pszContext, size_t tLength, size_t* ptReadSize = NULL);
std::string MBSFromUTF16(const WORD* pszInput, size_t tInputCch);
std::string MBSFromUTF32(const DWORD* pszInput, size_t tInputCch);
std::string MBSFromANSI(LPCSTR pszContext, size_t tLength, int nCodePage, size_t* ptReadSize = NULL);
std::string MBSFromMBS(std::string strInput);
std::string MBSFromWCS(std::wstring strInput);

std::wstring WCSFromANSI(LPCSTR pszContext, size_t tLength, int nCodePage, size_t* ptReadSize = NULL);
std::wstring WCSFromUTF8(LPCSTR pszContext, size_t tLength, size_t* ptReadSize = NULL);
std::wstring WCSFromUTF16(const WORD* pszInput, size_t tInputCch);
std::wstring WCSFromUTF32(const DWORD* pszInput, size_t tInputCch);
std::wstring WCSFromMBS(std::string strInput);
std::wstring WCSFromWCS(std::wstring strInput);
```

기존 16개의 함수에 20개의 함수를 추가하여 총 36개의 함수를 구현한 것입니다.

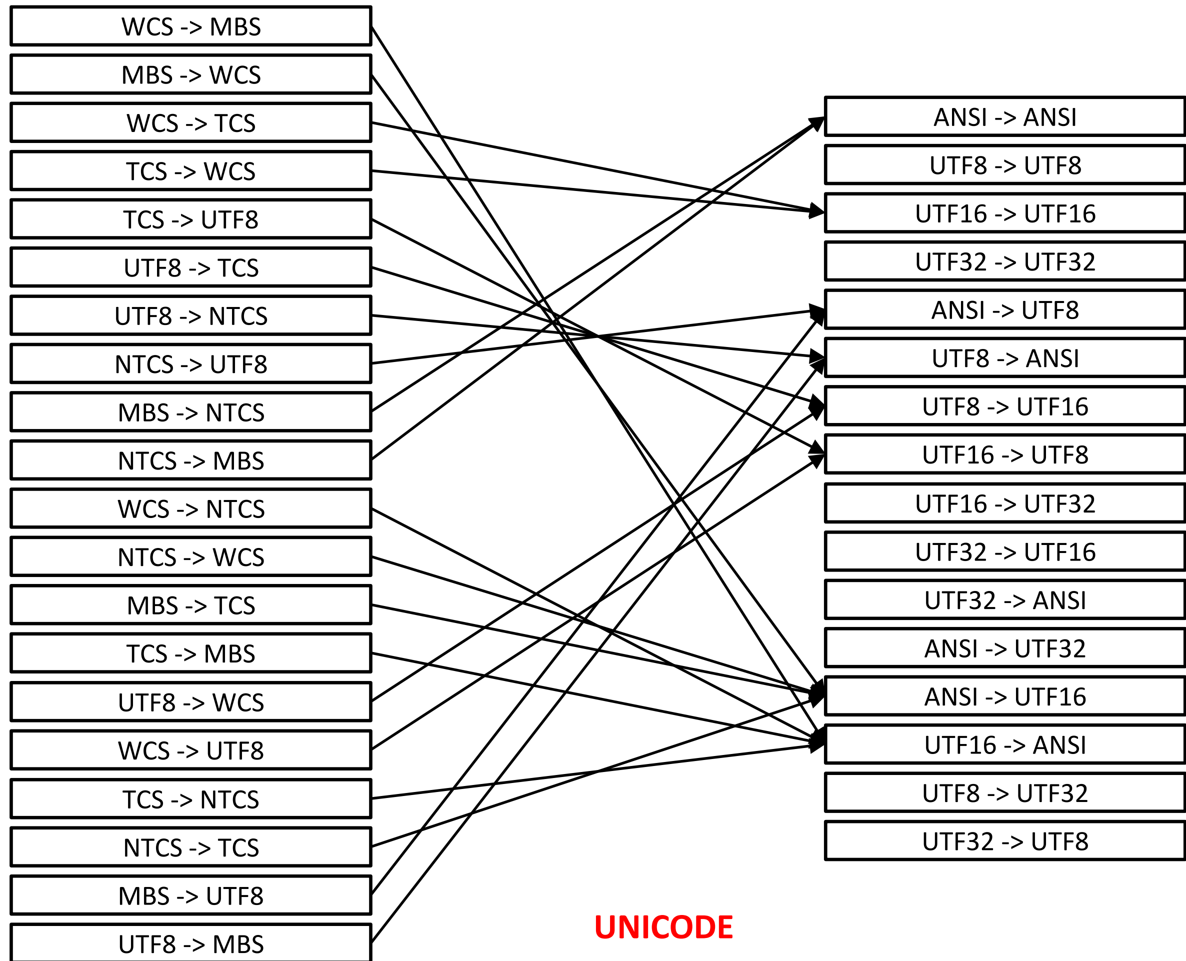
마지막으로 TCS가 포함된 유니코드 변환함수들

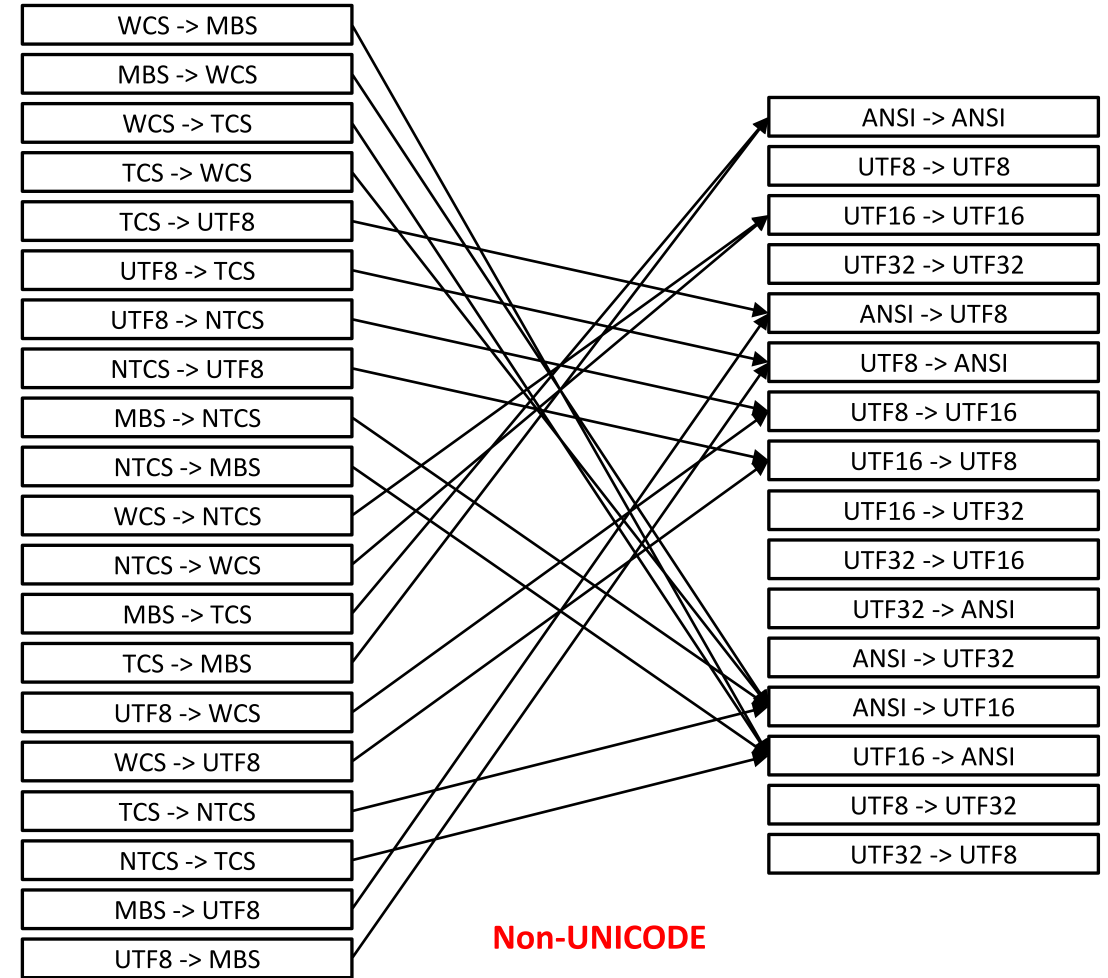
UTF8을 포함한 MBS, WCS, TCS, NTCS로 추상화된 문자열간의 변환 함수들



WCS -> MBS	TCS -> UTF8	MBS -> NTCS	MBS -> TCS	TCS -> NTCS
MBS -> WCS	UTF8 -> TCS	NTCS -> MBS	TCS -> MBS	NTCS -> TCS
WCS -> TCS	UTF8 -> NTCS	WCS -> NTCS	UTF8 -> WCS	MBS -> UTF8
TCS -> WCS	NTCS -> UTF8	NTCS -> WCS	WCS -> UTF8	UTF8 -> MBS

윈도우 환경에서 쓰이는 변환함수 매핑





유닉스 계열(맥)에서 쓰이는 변환함수 매핑

Linux UNICODE

WCS -> MBS
MBS -> WCS
WCS -> TCS
TCS-> WCS
TCS-> UTF8
UTF8 -> TCS
UTF8 -> NTCS
NTCS -> UTF8
MBS -> NTCS
NTCS -> MBS
WCS -> NTCS
NTCS -> WCS
MBS -> TCS
TCS -> MBS
UTF8 -> WCS
WCS -> UTF8
TCS-> NTCS
NTCS -> TCS
MBS -> UTF8
UTF8 -> MBS

EUCKR-> EUCKR
UTF8 -> UTF8
UTF16 -> UTF16
UTF32 -> UTF32
EUCKR-> UTF8
UTF8 -> EUCKR
UTF8 -> UTF16
UTF16 -> UTF8
UTF16 -> UTF32
UTF32 -> UTF16
UTF32 -> EUCKR
EUCKR-> UTF32
EUCKR-> UTF16
UTF16 -> EUCKR
UTF8 -> UTF32
UTF32 -> UTF8

Linux Non-UNICODE

WCS -> MBS
MBS -> WCS
WCS -> TCS
TCS -> WCS
TCS -> UTF8
UTF8 -> TCS
UTF8 -> NTCS
NTCS -> UTF8
MBS -> NTCS
NTCS -> MBS
WCS -> NTCS
NTCS -> WCS
MBS -> TCS
TCS -> MBS
UTF8 -> WCS
WCS -> UTF8
TCS -> NTCS
NTCS -> TCS
MBS -> UTF8
UTF8 -> MBS

EUCKR-> EUCKR
UTF8 -> UTF8
UTF16 -> UTF16
UTF32 -> UTF32
EUCKR-> UTF8
UTF8 -> EUCKR
UTF8 -> UTF16
UTF16 -> UTF8
UTF16 -> UTF32
UTF32 -> UTF16
UTF32 -> EUCKR
EUCKR-> UTF32
EUCKR-> UTF16
UTF16 -> EUCKR
UTF8 -> UTF32
UTF32 -> UTF8

▲ 리눅스에서의 '추상화된 인코딩 변환'과 '실제 인코딩 변환 함수'와의 매핑

사실상 기존에 구현한 함수를 define으로 선언한 것에 불과합니다.

```
#ifdef UNICODE
#define TCSFromWCS       WCSFromWCS
#define TCSFromMBS       WCSFromMBS
#define TCSFromANSI      WCSFromANSI
#define TCSFromUTF8      WCSFromUTF8
#define TCSFromUTF16     WCSFromUTF16
#define TCSFromUTF32     WCSFromUTF32
#define NTCSFromWCS      MBSFromWCS
#define NTCSFromMBS      MBSFromMBS
#define NTCSFromANSI     MBSFromANSI
#define NTCSFromUTF8     MBSFromUTF8
#define NTCSFromUTF16    MBSFromUTF16
#define NTCSFromUTF32    MBSFromUTF32

#define WCSFromTCS       WCSFromWCS
#define MBSFromTCS       MBSFromWCS
#define ANSIFromTCS      ANSIFromWCS
#define UTF8FromTCS      UTF8FromWCS
#define UTF16FromTCS     UTF16FromWCS
#define UTF32FromTCS     UTF32FromWCS
#define WCSFromNTCS      WCSFromMBS
#define MBSFromNTCS      MBSFromMBS
#define ANSIFromNTCS     ANSIFromMBS
#define UTF8FromNTCS     UTF8FromMBS
#define UTF16FromNTCS    UTF16FromMBS
#define UTF32FromNTCS    UTF32FromMBS

#define NTCSFromTCS      MBSFromWCS
#define TCSFromNTCS      WCSFromMBS
```

```
#else
#define TCSFromWCS       MBSFromWCS
#define TCSFromMBS       MBSFromMBS
#define TCSFromANSI      MBSFromANSI
#define TCSFromUTF8      MBSFromUTF8
#define TCSFromUTF16     MBSFromUTF16
#define TCSFromUTF32     MBSFromUTF32
#define NTCSFromWCS      WCSFromWCS
#define NTCSFromMBS      WCSFromMBS
#define NTCSFromANSI     WCSFromANSI
#define NTCSFromUTF8     WCSFromUTF8
#define NTCSFromUTF16    WCSFromUTF16
#define NTCSFromUTF32    WCSFromUTF32

#define WCSFromTCS       WCSFromMBS
#define MBSFromTCS       MBSFromMBS
#define ANSIFromTCS      ANSIFromMBS
#define UTF8FromTCS      UTF8FromMBS
#define UTF16FromTCS     UTF16FromMBS
#define UTF32FromTCS     UTF32FromMBS
#define WCSFromNTCS      WCSFromWCS
#define MBSFromNTCS      MBSFromWCS
#define ANSIFromNTCS     ANSIFromWCS
#define UTF8FromNTCS     UTF8FromWCS
#define UTF16FromNTCS    UTF16FromWCS
#define UTF32FromNTCS    UTF32FromWCS

#define NTCSFromTCS      WCSFromMBS
#define TCSFromNTCS      MBSFromWCS
#endif
```

참고, 변환함수와 네이밍의 중요성

Argument 순서나 사고의 흐름에 따라
To와 From을 적절히 쓰자!

std::string strUTF8 = UTF16ToUTF8(strUTF16)	X
std::string strUTF8 = UTF8FromUTF16(strUTF16)	O

UTF16ToUTF8(strUTF16, strUTF8)	O
UTF8FromUTF16(strUTF16, strUTF8)	X

퀴즈. TCHAR에 저장된 문자열 길이

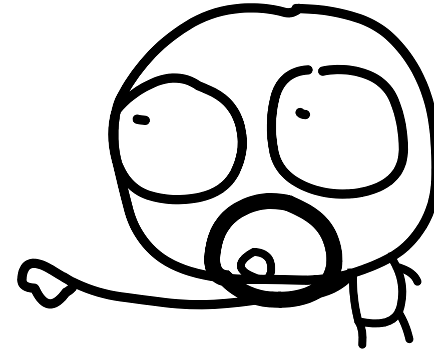
다음 코드에서 szBuffer 변수에 저장된 문자는 어떤 문자(인코딩)이며 길이는 몇 일까요?

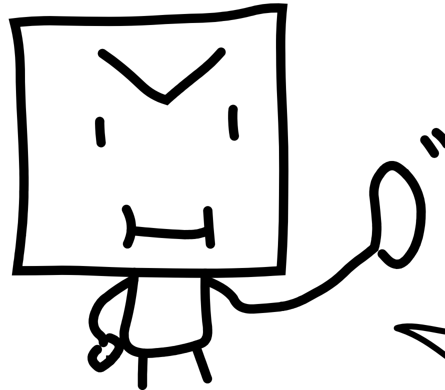
```
TCHAR szBuffer[] = TEXT(“Hello world!!”);
```

CPPCORE
문자열 함수들

C++가 다른 언어보다 어려운 이유?

1) 중급 언어라서
2) 문법이 복잡해서
3) **문자열 처리가 어려워서**
4) 지원하는 기능이 다양해서





[MFC]

CString::Format

```
void Format(LPCTSTR lpszFormat, ...);
```

[C#]

String.Format

Format(String, Object)
Format(String, Object[])
Format(IFormatProvider, String, Object)
Format(IFormatProvider, String, Object[])
Format(String, Object, Object)
Format(IFormatProvider, String, Object, Object)
Format(String, Object, Object, Object)
Format(IFormatProvider, String, Object, Object, Object)

[C언어]

??

[파이썬]

```
sub1 = "python string!"
a = "i am a %s" % sub1
```

[C++]

??

[JAVA]

```
String output = String.format("%s = %d", "joe", 35);
```

C언어 문자열 API들... 안해! ㅠㅠ

아마 대학교 1학년에 C언어 배우면서 한두 번은 써봤을 함수들일 겁니다.
네이밍도 이해 안 가지만 모르는 함수가 있으면 안 되던 현실.

strcpy	strlen	strrev
strcat	strlwr	strset
strchr	strncat	strspn
strcmp	strncmp	strstr
strcmpi	strncmpi	strtod
strcoll	strncpy	strtok
strcpy	strnicmp	strtol
strcspn	strset	strtoul
strdup	strpbrk	strupr
stricmp	strrchr	sprintf

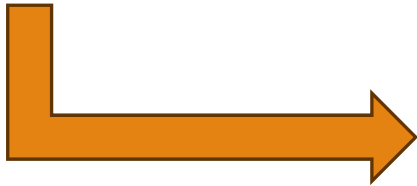
wcscpy	wcslen	wcsrev
wcscat	wcslwr	wcsset
wcschr	wcsncat	wcsspn
wcscmp	wcsncmp	wcswcs
wcscmpi	wcsncmpi	wcstod
wcscoll	wcsncpy	wcstok
wcscpy	wcsnicmp	wcstol
wcscspn	wcsset	wcstoul
wcsdup	wcspbrk	wcsupr
wcsicmp	wcsrchr	wsprintf

문자열 함수군은 새로 만들어야 한다, 왜?

1. STL에서 지원하는 문자열 연산자에 부족함이 있다.
2. TCHAR 변수형 지원이 필요하다.
3. 유닉스 계열 플랫폼에서 몇몇 wcs함수가 구현되지 않았다.
4. 오버플로우 공격에 취약한 함수들이 있다.

```
char* strcpy(char* pszDestination, const char* pszSource);
```

- 기본 제공 함수: BOF 공격에 사용됨



```
HRESULT StringCbCopy(LPTSTR  pszDest, size_t  cbDest,
LPCTSTR pszSrc);
```

- 윈도우 보완 함수: 기존 인자 사이에 사이즈 변수가 추가됨

```
size_t strlcpy(char *dst, const char *src, size_t size);
```

- 유닉스 계열 보완 함수: 마지막 인자로 사이즈 변수가 추가됨

양 진형에서 사용하는 보완 함수의 생김새가 상이함.

제정된 표준이 없어 통일 불가.

플랫폼별 BOF를 대비하는 cppcore의 함수들, Windows 함수 형식으로 통일함

```
ECODE StringCchCopy(char* pszDest, size_t tDestCch, const char* pszSrc);
ECODE StringCchCopy(wchar_t* pszDest, size_t tDestCch, const wchar_t* pszSrc);
ECODE StringCchCopyN(char* pszDest, size_t tDestCch, const char* pszSrc, size_t tCopyCch);
ECODE StringCchCopyN(wchar_t* pszDest, size_t tDestCch, const wchar_t* pszSrc, size_t tCopyCch);
ECODE StringCchCat(char* pszDest, size_t tDestCch, const char* pszSrc);
ECODE StringCchCat(wchar_t* pszDest, size_t tDestCch, const wchar_t* pszSrc);
ECODE StringCchCatN(char* pszDest, size_t tDestCch, const char* pszSrc, size_t tAppendCch);
ECODE StringCchCatN(wchar_t* pszDest, size_t tDestCch, const wchar_t* pszSrc, size_t tAppendCch);
ECODE StringCchPrintf(char* pszDest, size_t tDestCch, const char* pszFormat, ...);
ECODE StringCchPrintf(wchar_t* pszDest, size_t tDestCch, const wchar_t* pszFormat, ...);
ECODE StringCchVPrintf(char* pszDest, size_t tDestCch, const char* pszFormat, va_list vaList);
ECODE StringCchVPrintf(wchar_t* pszDest, size_t tDestCch, const wchar_t* pszFormat, va_list vaList);
ECODE StringCchGets(char* pszDest, size_t tDestCch);
ECODE StringCchGets(wchar_t* pszDest, size_t tDestCch);
ECODE StringCchLength(const char* pszContext, size_t tMaxCch, size_t* pLengthCch);
ECODE StringCchLength(const wchar_t* pszContext, size_t tMaxCch, size_t* pLengthCch);
```

```
ECODE StringCbCopy(char* pszDest, size_t tDestCb, const char* pszSrc);
ECODE StringCbCopy(wchar_t* pszDest, size_t tDestCb, const wchar_t* pszSrc);
ECODE StringCbCopyN(char* pszDest, size_t tDestCb, const char* pszSrc, size_t tCopyCb);
ECODE StringCbCopyN(wchar_t* pszDest, size_t tDestCb, const wchar_t* pszSrc, size_t tCopyCb);
ECODE StringCbCat(char* pszDest, size_t tDestCb, const char* pszSrc);
ECODE StringCbCat(wchar_t* pszDest, size_t tDestCb, const wchar_t* pszSrc);
ECODE StringCbCatN(char* pszDest, size_t tDestCb, const char* pszSrc, size_t tAppendCb);
ECODE StringCbCatN(wchar_t* pszDest, size_t tDestCb, const wchar_t* pszSrc, size_t tAppendCb);
ECODE StringCbPrintf(char* pszDest, size_t tDestCb, const char* pszFormat, ...);
ECODE StringCbPrintf(wchar_t* pszDest, size_t tDestCb, const wchar_t* pszFormat, ...);
ECODE StringCbVPrintf(char* pszDest, size_t tDestCb, const char* pszFormat, va_list vaList);
ECODE StringCbVPrintf(wchar_t* pszDest, size_t tDestCb, const wchar_t* pszFormat, va_list vaList);
ECODE StringCbGets(char* pszDest, size_t tDestCb);
ECODE StringCbGets(wchar_t* pszDest, size_t tDestCb);
ECODE StringCbLength(const char* pszContext, size_t tMaxCb, size_t* pLengthCb);
ECODE StringCbLength(const wchar_t* pszContext, size_t tMaxCb, size_t* pLengthCb);
```

Cb와 Cch의 차이는 가리키는 size의 규격을 나타냄. Cb는 바이트 단위, Cch는 문자개수 단위

기본적인 문자열 연산을 지원해주는 STL 공통 클래스

멤버 함수	설명
append	문자열 끝에 문자를 추가합니다.
assign	문자열의 내용에 새 문자 값을 할당합니다.
at	문자열의 지정된 위치에 있는 요소에 대한 참조를 반환합니다.
back	
begin	문자열의 첫 번째 요소 주소를 지정하는 반복기를 반환합니다.
c_str	**문자열의 내용을 C 스타일의 null로 종료되는 문자열로 변환합니다.**
capacity	문자열의 메모리 할당을 늘리지 않고도 문자열에 저장할 수 있는 요소의 최대 수를 반환합니다.
cbegin	문자열의 첫 번째 요소 주소를 지정하는 const 반복기를 반환합니다.
cend	문자열에서 마지막 요소 다음에 나오는 위치를 주소 지정하는 const 반복기를 반환합니다.
clear	**문자열의 모든 요소를 지웁니다.**
compare	문자열을 지정된 문자열과 비교하여 두 문자열이 같은지 아니면 한 문자열이 다른 문자열보다 사전순으로 더 작은지를 확인합니다.
copy	소스 문자열의 인덱싱된 위치에서 지정한 수까지의 문자를 대상 문자 배열에 복사합니다. 더 이상 사용되지 않습니다. 대신 basic_string::_Copy_s를 사용합니다.
crbegin	역방향 문자열에서 첫 번째 요소의 주소를 지정하는 const 반복기를 반환합니다.
crend	역방향 문자열에서 마지막 요소 다음에 나오는 위치의 주소를 지정하는 상수 반복기를 반환합니다.
_Copy_s	소스 문자열의 인덱싱된 위치에서 지정한 수까지의 문자를 대상 문자 배열에 복사합니다.
data	문자열의 내용을 문자 배열로 변환합니다.
empty	문자열에 문자가 있는지 테스트합니다.
end	문자열에서 마지막 요소 다음에 나오는 위치의 주소를 지정하는 반복기를 반환합니다.
erase	문자열에서 지정된 위치의 요소 또는 요소 범위를 제거합니다.
find	문자열에서 지정된 문자 시퀀스와 일치하는 첫 번째 하위 문자열을 정방향으로 검색합니다.
find_first_not_of	문자열에서 지정된 문자열의 요소가 아닌 첫 번째 문자를 검색합니다.
find_first_of	문자열에서 지정된 문자열의 요소와 일치하는 첫 번째 문자를 검색합니다.
find_last_not_of	문자열에서 지정된 문자열의 요소가 아닌 마지막 문자를 검색합니다.
find_last_of	문자열에서 지정된 문자열의 요소인 마지막 문자를 검색합니다.
front	문자열의 첫 번째 요소에 대한 참조를 반환합니다.
get_allocator	문자열을 생성하는 데 사용된 allocator 개체의 복사본을 반환합니다.
insert	요소 하나 또는 여러 개나 요소의 범위를 문자열의 지정된 위치에 삽입합니다.
length	**문자열의 현재 요소 수를 반환합니다.**
max_size	문자열이 포함할 수 있는 최대 문자 수를 반환합니다.
pop_back	문자열의 마지막 요소를 지웁니다.
push_back	문자열 끝에 요소를 추가합니다.
rbegin	역방향 문자열에서 첫 번째 요소의 주소를 지정하는 반복기를 반환합니다.
rend	역방향 문자열에서 마지막 요소 바로 다음을 가리키는 반복기를 반환합니다.
replace	문자열에서 지정된 위치의 요소를 다른 범위 또는 문자열이나 C 문자열에서 복사한 문자 또는 지정된 문자로 바꿉니다.
reserve	문자열의 용량을 최소한 지정된 숫자보다 크게 설정합니다.
resize	문자열의 새 크기를 지정하고 필요에 따라 요소를 추가하거나 지웁니다.
rfind	문자열에서 지정된 문자 시퀀스와 일치하는 첫 번째 하위 문자열을 역방향으로 검색합니다.
shrink_to_fit	문자열의 초과 용량을 삭제합니다.
size	문자열의 현재 요소 수를 반환합니다.
substr	**지정된 위치부터 시작하여 문자열의 하위 문자열을 최대 특정 문자 수만큼 복사합니다.**
swap	두 문자열의 내용을 교환합니다.

유니코드를 위해서
앞으로 std::tstring을 주로 쓰기로 한다.

```
namespace std
{
#ifdef UNICODE
    typedef std::wstring  tstring;
    typedef std::string   ntstring;
#else
    typedef std::string   tstring;
    typedef std::wstring  ntstring;
#endif
}
```

아래 함수들을 만든 순간
이미 C++ 난이도는 파이썬 수준으로 떨어졌다.

```
std::string Format(const char* pszFormat, ...);
std::string& Trim(std::string& strText, LPCSTR pszWhiteSpace = "₩t₩n₩r ");
std::string& TrimLeft(std::string& strText, LPCSTR pszWhiteSpace = "₩t₩n₩r ");
std::string& TrimRight(std::string& strText, LPCSTR pszWhiteSpace = "₩t₩n₩r ");
std::string& MakeLower(std::string& strText);
std::string& MakeUpper(std::string& strText);
std::string& Replace(std::string& strText, std::string strTarget, std::string strReplace);
std::string Tokenize(const std::string& strText, int& nOffset);
std::string Tokenize(const std::string& strText, std::string strDelimiter, int& nOffset);
void Split(const std::string& strContext, std::string strDelimiter, std::string& strFront, std::string& strBack);
```

```
std::wstring Format(const wchar_t* pszFormat, ...);
std::wstring& Trim(std::wstring& strText, LPCWSTR pszWhiteSpace = L"₩t₩n₩r ");
std::wstring& TrimLeft(std::wstring& strText, LPCWSTR pszWhiteSpace = L"₩t₩n₩r ");
std::wstring& TrimRight(std::wstring& strText, LPCWSTR pszWhiteSpace = L"₩t₩n₩r ");
std::wstring& MakeLower(std::wstring& strText);
std::wstring& MakeUpper(std::wstring& strText);
std::wstring& Replace(std::wstring& strText, std::wstring strTarget, std::wstring strReplace);
std::wstring Tokenize(const std::wstring& strText, int& nOffset);
std::wstring Tokenize(const std::wstring& strText, std::wstring strDelimiter, int& nOffset);
void Split(const std::wstring& strContext, std::wstring strDelimiter, std::wstring& strFront, std::wstring& strBack);
```

문자열 검색식

아스테리크(*)와 퀘스천 마크(?)를 이용한 검색식에 익숙하시죠? 예를들어 다음과 같이 dir *.zip을 실행하면 확장자가 zip인 파일들의 목록을 볼 수 있습니다. 검색식으로 문자열 비교하는 함수를 만들 수 있겠습니까?

```
C:₩WINDOWS₩system32₩cmd.exe

D:₩Sample>dir *.zip
 D 드라이브의 볼륨: Source
 볼륨 일련 번호: 92F3-662A

 D:₩Sample 디렉터리

2023-01-03  오후 10:37         1,257,525 2023-01-03.zip
2023-01-07  오전 09:31            16,459 2023-01-07.zip
2023-01-31  오후 11:32            16,061 2023-01-31.zip
2023-03-25  오전 12:41         3,231,144 2023-03-24.zip
2023-03-25  오후 06:36           183,202 2023-03-25.zip
2023-03-27  오후 05:04         9,999,266 2023-03-27.zip
2023-03-31  오후 08:58            17,864 2023-03-31.zip
2023-04-02  오후 12:33         4,767,470 2023-04-02.zip
2023-04-03  오전 12:44           156,741 2023-04-03.zip
2023-12-08  오후 11:47        19,181,553 error(PW_1004).zip
2022-11-24  오전 12:34        92,215,951 NoSpearTargetSample(xlm_emul분리).zip
2024-04-22  오후 02:45        60,696,372 sample_for_QuadMiners(PW_1004).zip
2024-04-24  오후 05:25         1,233,076 sample_for_QuadMiners_reports(PW_1004).zip
2023-12-04  오후 02:36     2,134,429,978 tta.zip
              14개 파일       2,327,402,662 바이트
               0개 디렉터리  778,704,166,912 바이트 남음

D:₩Sample>
```

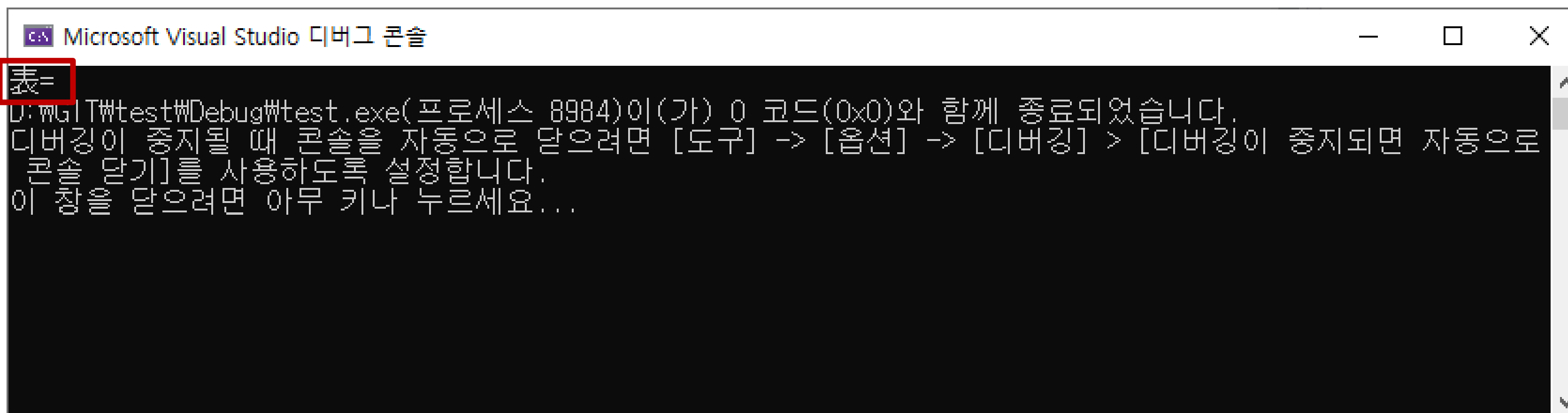
```
bool StrCmpWithWildcard(std::string strContext, std::string strPattern);
bool StrCmpWithWildcard(std::wstring strContext, std::wstring strPattern);
bool SafeStrCmpWithWildcard(const char* pszDest, size_t tDestCch, const char* pszPattern);
bool SafeStrCmpWithWildcard(const wchar_t* pszDest, size_t tDestCch, const wchar_t* pszPattern);
```

STL 문자열은 std::string과 std::wstring이 존재합니다. 문자열끼리 더하는 연산이 기본으로 제공됩니다.

```
#include <string>

int main()
{
    std::string a = "Hello";
    std::string b = "world!!";
    std::string c = a + " " + b;
    printf("%s", c);
    return 0;
}
```

위 코드의 실행 결과는?

안타깝게도 다음과 같이 예상밖의 값이 출력됩니다.



올바르게 출력하기 위해서는 다음과 같이 c_str() 메소드를 호출해 native 문자열 포인터를 얻어와야 합니다.

```
#include <string>

int main()
{
    std::string a = "Hello";
    std::string b = "world!!";
    std::string c = a + " " + b;
    printf("%s", c.c_str());
    return 0;
}
```

```
Microsoft Visual Studio 디버그 콘솔
Hello world!!
D:\GIT\test\Debug\test.exe(프로세스 21220)이(가) 0 코드(0x0)와 함께 종료되었습니다.
이 창을 닫으려면 아무 키나 누르세요...
```

그 외에 특정 위치의 문자만 변경하는 것은 배열처럼 첨자 연산자[]를 이용하면 됩니다.

```
#include <string>

int main()
{
    std::string a = "Hello world!!";
    a[0] = 'h';
    printf("%s", a.c_str());
    return 0;
}
```

```
Microsoft Visual Studio 디버그 콘솔
hello world!!
D:\GIT\test\Debug\test.exe(프로세스 2228)이(가) 0 코드(0x0)와 함께 종료되었습니다.
이 창을 닫으려면 아무 키나 누르세요...
```

리눅스에서 빌드하기

윈도우에 WSL 설치하기

1. WSL 활성화

- [관리자 권한]으로 PowerShell을 열고 다음 명령어 실행:

```
wsl --install
```

2. 설치 후 재부팅

- Ubuntu 24.04 LTS 설치
- 설치 가능한 배포판 확인:

```
wsl --list –online
```

- Ubuntu 24.04 설치:

```
wsl --install -d Ubuntu-24.04
```

3. 초기 설정

- 사용자 이름과 비밀번호 설정
- Windows 시작 메뉴에서 Ubuntu 실행

혹은 Microsoft Store 에서 Ubuntu 검색해서 설치할 수도 있음

개발도구 설치하기

1. 우분투 터미널로 들어가서 다음 명령어 실행

```
sudo apt update && sudo apt upgrade -y
```

2. C++ 컴파일러(g++) 설치

```
sudo apt install build-essential -y
```

- build-essentia에는 g++, gcc, make 등이 포함되어 있어 C++ 개발에 필수입니다.

3. CMake 설치

```
sudo apt install cmake -y
```

- 설치확인: cmake --version

호스트 경로의 소스파일 이용하기

1. Windows 디렉토리를 WSL에서 접근하기

- WSL에서는 기본적으로 Windows 파일 시스템을 /mnt 아래에 마운트합니다.

- 예시:

```
cd /mnt/d/GIT
```

2. WSL 디렉토리를 Windows에서 접근하기

- WSL의 홈 디렉토리나 프로젝트 폴더를 Windows에서 열고 싶다면 WSL 우분투 터미널에서:

```
explorer.exe .
```

- 현재 디렉토리를 Windows 파일 탐색기로 엽니다.

CMake를 이용해 간단한 실행파일 빌드하기

1. 스크립트 설정

- main.cpp가 있는 디렉토리에 다음과 같은 CMakeLists.txt 파일을 생성합니다.

```
cmake_minimum_required(VERSION 3.10)
project(lecture-04)
set(CMAKE_CXX_STANDARD 17)
add_executable(${PROJECT_NAME} main.cpp)
```

2. CMake 구동

- 빌드에 필요한 임시 파일들이 저장될 디렉토리를 생성하여 이동합니다:

```
mkdir -p ./Output/LinuxRelease
cd ./Output/LinuxRelease
```

- CMakeList.txt 파일이 존재하는 디렉토리를 지정하여 빌드 임시 파일을 생성합니다:

```
cmake ../../Src/lecture-04
```

- 빌드합니다:

```
cmake --build .
```

- (옵션) 향후 위 두 명령을 한 번에 입력하는 방법은 다음과 같습니다:

```
cmake ../../Src/lecture-04 && cmake --build .
```

CMake를 이용해 여러 소스파일이 존재하는 실행파일 빌드하기

1. 스크립트 설정

- main.cpp가 있는 디렉토리에 다음과 같은 CMakeLists.txt 파일을 생성합니다.

```
cmake_minimum_required(VERSION 3.10)
project(lecture-04)
set(CMAKE_CXX_STANDARD 17)

file(GLOB_RECURSE SOURCES *.cpp *.c)

foreach(SOURCE IN LISTS SOURCES)
		message(STATUS "SOURCE - " ${SOURCE})
endforeach(SOURCE)

add_executable(${PROJECT_NAME} $(SOURCES})
```

1. 스크립트 설정

- 라이브러리 소스코드가 담긴 디렉토리에 다음과 같은 CMakeLists.txt 파일을 생성합니다.

```
cmake_minimum_required(VERSION 3.10)
project(lecture-04-lib)
set(CMAKE_CXX_STANDARD 17)

file(GLOB_RECURSE SOURCES *.cpp *.c)

foreach(SOURCE IN LISTS SOURCES)
		message(STATUS “SOURCE – “ ${SOURCE})
endforeach(SOURCE)

add_library(${PROJECT_NAME} $(SOURCES})
```

CMake를 이용해 여러 프로젝트 빌드하기

1. 스크립트 설정

- 소스파일 상위 디렉토리에 CMakeLists.txt 파일을 생성합니다.
- 예를들어, /mnt/d/GIT/profrog/Src/CMakeLists.txt 파일에:

```
cmake_minimum_required(VERSION 3.1)

add_subdirectory(lecture-04-lib)
add_subdirectory(lecture-04)
```

- 먼저 빌드되어야 하는 순서대로 기입합니다.

VisualStudio의 솔루션과 동일한 기능입니다.

1. 스크립트 설정

- main.cpp가 있는 디렉토리에 다음과 같은 CMakeLists.txt 파일을 생성합니다.

```
cmake_minimum_required(VERSION 3.10)
project(lecture-04)
set(CMAKE_CXX_STANDARD 17)

file(GLOB_RECURSE SOURCES *.cpp *.c)

foreach(SOURCE IN LISTS SOURCES)
		message(STATUS "SOURCE - " ${SOURCE})
endforeach(SOURCE)

add_executable(${PROJECT_NAME} $(SOURCES})
target_link_libraries(${PROJECT_NAME} lecture-04-lib)
```

CMake를 이용해 외부의 정적 라이브러리를 링크하는 실행파일 빌드하기

1. 스크립트 설정

- main.cpp가 있는 디렉토리에 다음과 같은 CMakeLists.txt 파일을 생성합니다.

```
cmake_minimum_required(VERSION 3.10)
project(lecture-04)
set(CMAKE_CXX_STANDARD 17)

file(GLOB_RECURSE SOURCES *.cpp *.c)

foreach(SOURCE IN LISTS SOURCES)
		message(STATUS "SOURCE - " ${SOURCE})
endforeach(SOURCE)

include_directories(${CMAKE_CURRENT_SOURCE_DIR}/../../../cppcore/Inc)
link_directories(${CMAKE_CURRENT_SOURCE_DIR}/../../../cppcore/Build/LinuxRelease)

add_executable(${PROJECT_NAME} ${SOURCES})
target_link_libraries(${PROJECT_NAME} lecture-04-lib cppcore)
```

- 이때 중요한 것은 종속성이 낮은 라이브러리 부터 기입해야 한다는 점입니다.(순서 중요)

  즉, lecture-04-lib가 cppcore를 이용한다면 먼저 기입되어야 합니다.

- 참고로 외부 라이브러리는 link_directory를 설정했지만 앞서 lecture-04-lib은 설정하지 않은 이유는

  같은 CMakeLists.txt 에서 빌드된 것이기 때문입니다.(VisualStudio의 참조 기능과 유사함)

CMake를 이용해 빌드한 산출물이 Build 디렉토리에 생성되도록 만들기

1. 스크립트 설정

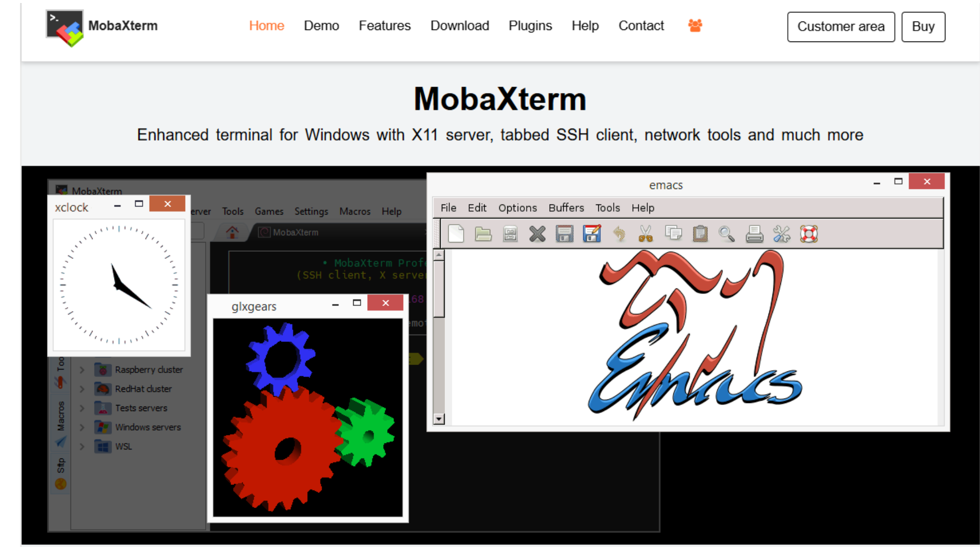
- 리눅스에서 빌드된 산출물을 지정하는 방법은 VisualStudio보다 훨씬 간단합니다.

```
cmake_minimum_required(VERSION 3.10)
project(lecture-04)
set(CMAKE_CXX_STANDARD 17)

set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/../../Build/LinuxRelease)
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/../../Build/LinuxRelease)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/../../Build/LinuxRelease)

file(GLOB_RECURSE SOURCES *.cpp *.c)

foreach(SOURCE IN LISTS SOURCES)
		message(STATUS "SOURCE - " ${SOURCE})
endforeach(SOURCE)

include_directories(${CMAKE_CURRENT_SOURCE_DIR}/../../../cppcore/Inc)
link_directories(${CMAKE_CURRENT_SOURCE_DIR}/../../../cppcore/Build/LinuxRelease)

add_executable(${PROJECT_NAME} ${SOURCES})
target_link_libraries(${PROJECT_NAME} lecture-04-lib cppcore)
```

리눅스 빌드 원격 서버에 빌드한 실행 파일을 복사해서 실행해보기

1. MobaXTerm을 다운로드 받아 설치합니다. (설치하지 않는 portable 버전 추천)

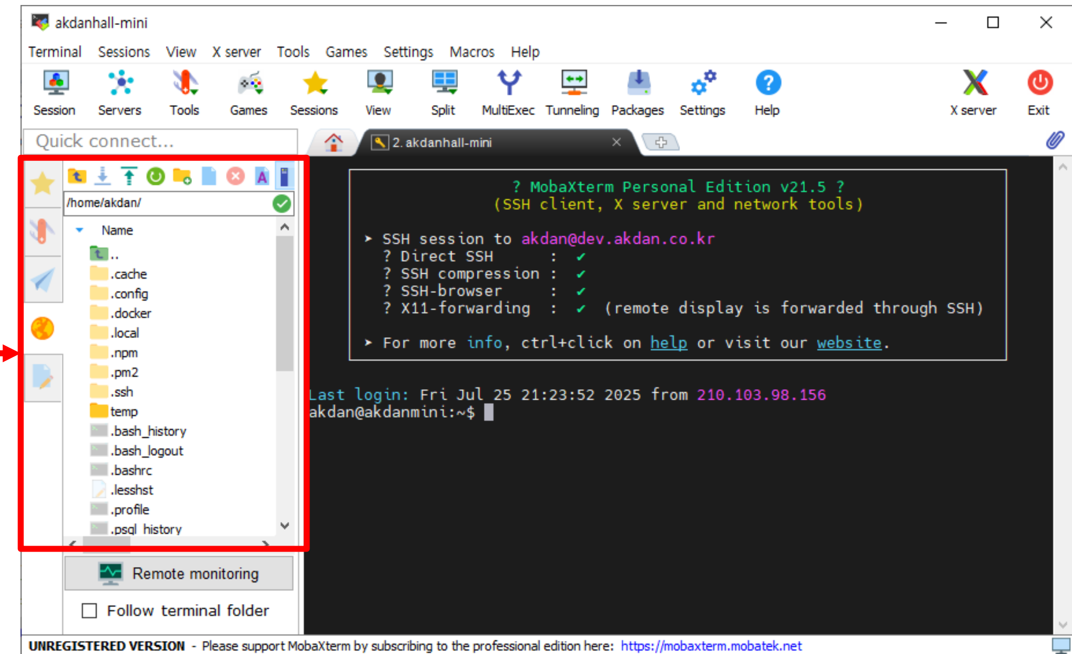
- 기존 putty 대비 장점: GUI로 파일을 송수신 할 수 있음



2. 접속하고 나면 왼쪽에 파일탐색기 창이 뜨는데,

1) 이곳에 실행파일을 드래그인하면 서버 내부로 파일이 업로드됩니다.

2) 탐색된 원격서버 파일을 더블클릭하면 내 PC로 파일이 다운로드됩니다.



고생 많으셨습니다!