※ 본 아티클은 CMP MEDIA LLC와의 라이선스 계약에 의해 국문으로 제공됩니다

Gamasutra.com

고성능, 저수준 문자열 라이브러리

Joe Linhoff

2007년8월15일

http://www.gamasutra.com/view/feature/1607/a_high_performance_low_level_.php

본 아티클은 문자열 처리와 저수준 C 라이브러리에 대한 성능 위주의 접근법을 제시한다. 안전하고 빠른 문자열 처리를 사용함으로써 도구(Tool), 게임 엔진, 게임에서 성능적인 이득을 얻을 수 있다. 많은 수의 문자열 라이브러리들은 간혹 특정 상황에서 불안정해지고 느려지며, 멀티 스레드 프로그래밍에 적합하지 않는 표준 C 문자열 함수를 기반으로 하고 있다. 이런 표준 C 문자열 함수는 프로그램의 안정성, 깔끔함, 효율적인 문제 해결에 걸림돌이 되기도 한다.

다수의 고수준 문자열 라이브러리가 C의 표준 접근법을 기반으로 작성되어 이런 문제점을 고스란히 물려받고 말았다. 본 아티클에서는 이러한 문제점을 지적하고 이를 극복할 수 있는 새로운 저수준 라이브러리에 대해 소개하고 있다. 문자열 라이브러리의 코드는 이곳에서 다운로드받을 수 있다.

문자열은 문자 집합을 구성할 때 사용되는 파생 자료형이다. 워드 프로세서, 텍스트 편집기, 프로그래밍 도구, 게임을 포함한 다양한

종류의 응용 프로그램들에서 문자열은 광범위하게 사용된다. 그리고 대부분의 게임에서 문자열은 메시지, 플레이어 정보, 설정, 스크립트, 자원의 내부 인식 등에 사용된다.

문자열은 문자의 배열로 구성된다. 문자열 처리의 가장 보편적인 접근 방식은 C 언어가 처음 소개되던 날까지 거슬러 올라간다.[1] 이 접근법은 문자열의 첫 문자를 가리키는 포인터와 문자열의 끝을 나타내기 위해 0(₩0)을 사용한다.

문자열의 첫 번째 문자가 “0”이라면 해당 문자열은 비어있는 것이다. 비어있지 않은 문자열에는 "A"와 같이 단 하나의 문자만 들어있을 수도 있고 "hello"와 같은 단어가 들어있을 수도 있으며, 문장 전체가 들어있을 수도 있고, 심지어 파일 전체가 들어있을 수도 있다. 표준 큰따옴표(")는 문자열을 구분 지을 때 사용되며 작은 따옴표(')는 한 개의 문자를 구분 지을 때 사용된다.

문자열	문자열에 들어있는 문자와 0 종결자(zero-terminator, ₩0)											문자열 길이
	0											0
"A"	'A'	0										1
"hello"	'h'	'e'	'l'	'l'	'o'	0						5
"set x 3.14"	's'	'e'	't'	' '	'x'	' '	'3'	'.'	'1'	'4'	0	10

문자형

새로운 라이브러리에서는 문자를 위해 'chr'이라는 형식을 정의하고 사용한다. 필요하다면 2의 n승의 크기를 가지는 문자를 위해 라이브러리를 다시 생성할 수도 있다.

문자열 종결

새 라이브러리는 표준 0 종결 C 문자열을 지원하며 기존의 모든 코드와 함께 사용할 수 있도록 설계되었다. 새 라이브러리와 표준 C 라이브러리 간의 가장 큰 차이점은 문자열 처리 루틴에 선택적인 포인터 종결자 매개변수가 추가되었다는 것이다. 포인터 종결자는 문자열에서 사용 가능한 마지막 문자 다음의 위치를 가리키는 포인터이다. 그러므로 이 포인터는 0 종결자의 위치를 가리키거나 0 종결자가 들어가야 할 위치를 가리키고 있어야 한다.

포인터 종결 문자열(Pointer terminated string)의 지원으로 인해 상당한 이득을 얻을 수 있다. 우선은, 포인터 종결 문자열을 사용하면 버퍼 자체를 수정하지 않아도 공유 버퍼에 안전하게 단어를 정의할 수 있다는 것이다. 예를 들어 버퍼를 탐색해서 개별적인 문자열로 취급할 하나의 단어를 찾는다고 할 때, C의 표준 루틴에서는 다음과 같은 두 가지 방법을 사용할 수 있다. 일시적으로 버퍼에서 단어의 끝부분에 0을 삽입하던지, 다른 버퍼에 단어를 복사하는 것이다. 하지만 버퍼 수정은 불가능한 경우도 있어 가능하면 피하는 것이 좋다. 단어를 다른 버퍼에 복사하는 경우는 일반적으로 메모리 할당과 해제 과정이 추가되며 처리 시간이 더 길어지게 된다. 따라서 두 가지 방법 모두 그리 좋은 방법은 아니다. 하지만 포인터 종결 문자열을 사용하면 단어의 처음과 끝부분의 포인터만 넘겨주면 되므로 위의 문제점들을 피할 수 있다.

포인터 종결 문자열을 사용하면 프로그래머가 포인터 종결자에서 시작 포인터를 빼기 연산하면 문자열의 길이를 알아낼 수 있다는 장점이 있다. 그에 더해 문자열의 끝부분을 이미 알고 있으므로 문자열을 역방향으로 탐색하는 루틴을 간략화시킬 수 있다.

강제 0 종결

새 라이브러리가 따르는 하나의 규칙은 버퍼 생성 함수가 항상 버퍼에 0 종결자를 추가하거나 강제한다는 것이다. 이로 인해 목표 버퍼의 크기보다 길이가 긴 문자열을 집어넣었을 때 넘친 부분이 잘리게 되어 버퍼 오버플로우(Buffer Overflow) 문제 처리가 간단해진다. 또한 새 라이브러리를 기존 코드와 통합하는 것도 더 간단해진다.

포인터 종결 매개변수

포인터 종결 문자열을 사용할 때 잠재적인 단점도 존재한다. 단점 중에 하나는 간혹 문자열의 끝을 가리키는 포인터가 없을 경우에 해당 포인터를 얻기 위해 문자열 전체를 수동으로 탐색해야 한다는 것이다. 이 단점을 피하려면 포인터 종결자에 항상 널 포인터(null pointer)를 넘겨주면 된다. 포인터 종결자에 널 포인터를 넘겨주면 루틴은 문자열을 표준 0 종결 문자열로 취급할 것이다.

또 다른 단점으로는 루틴에 널 포인터를 넘겨주면 성능이 떨어진다는 것이다. 이 문제의 해결을 위해 대다수의 루틴이 여러 버전을 가지고 있으며, 그 중 몇몇은 포인터 종결자를 받지 않고 모든 문자열을 0 종결 문자열로 취급한다.

길이와 크기

용어를 혼동할 경우에는 찾기 힘든 버그가 생길 수도 있다. 표준 루틴과 같이 새 라이브러리의 루틴들은 '길이'라는 용어의 정의를 문자열에 있는 문자의 개수라고 간주한다. 하지만 표준 루틴과 다르게 '크기'라는 용어는 종결자를 포함한 문자열 버퍼의 바이트(byte) 단위 크기를 나타낸다. 예를 들어 "abc"라는 문자열은 길이가 3이고 크기는

4*sizeof(chr)의 크기를 가진다. “abc”라는 문자열의 크기에는 3개의 문자와 0 종결자가 포함된다.

```
int n;
n = szlen("abc"); // n == 3
n = szlen2("abc",NULL); // n == 3
n = szsize("abc"); // n == 4*sizeof(chr)
n = szsize2("abc",NULL); // n == 4*sizeof(chr)
```

메모리 관리와 문자열

표준 문자열 루틴과 이 접근법의 또 다른 근본적인 차이점은 이 라이브러리에 있는 루틴들은 메모리를 할당하거나 해제하지 않는다는 것이다. 속도가 빠른 코드를 작성할 때, 특히 메모리 관리에 관련된 것을 작성할 때에는 시스템 라이브러리 함수의 호출을 최대한 자제해야 하기 때문이다. 문자열의 크기를 키워야 할 경우, 응용 프로그램이나 고수준 문자열 관리자는 반드시 해당 문자열의 크기를 키우는 방법을 사용해 처리해야 한다.

버퍼 생성

버퍼를 생성하는 모든 루틴에는 버퍼의 시작 위치를 가리키는 포인터와 전체 버퍼의 바이트(byte) 단위 크기를 넘겨줘야 한다. 그러면 루틴은 해당 버퍼에 들어있는 0 종결자를 포함하지 않는 문자의 개수를 반환한다. 목표 버퍼의 크기를 바이트 단위로 넘기는 것과 문자 개수를 반환하는 것에는 약간 일관성이 떨어져 보이지만 이것은 버퍼 조작을 더 쉽게 해준다.

다음은 0 종결 문자열을 버퍼로 복사하는 예제이다.

```
int n;
chr buf[8];
```

n = szcpy(buf,sizeof(buf),"abc",NULL); // n == 3
n = szsize(buf); // n == 4*sizeof(chr)

다음 예제는 문자열을 정의하고 그 일부를 복사하는 것이다. 주석 부분은 문자열에 들어있는 각 문자의 색인을 나타낸다. 그 다음에 나오는 코드에서는 시작 포인터를 10번째 색인 위치에 있는 문자로 지정하고 포인터 종결자를 27번째 색인 위치에 있는 문자로 지정해 “buffer that might”라는 문자열을 정의하고 있다.

```
// 문자 0 1 2 3 4 5
//                                                                색인
012345678901234567890123456789012345678901234567890123456789012345
678
chr bigstr[] = "This is a buffer that might be shared by multiple routines.";
int n;
chr buf[32];
chr *ss = &bigstr[10]; // start at "buffer"
chr *sx = &bigstr[27]; // end after "might"
```

아래의 코드는 문자열을 버퍼로 복사한다. 그리고 버퍼에 복사된 문자 개수가 반환된다. 복사 루틴은 다른 모든 버퍼 생성 루틴과 같이 목표 버퍼의 크기가 0이 아니라면 목표 문자열을 0으로 종결하거나 잘라낼 것이라는 것을 보장한다.

```
n = szcpy(buf,sizeof(buf),ss,sx); // n == 17, buf == "buffer that might"
```

문자열 비교

표준 문자열 함수와 같이 대·소문자를 구분하는 것과 구분하지 않는 것, 두 가지의 비교 함수가 제공된다. 또한 다양한 변종 함수들이 있으므로 자신의 데이터에 가장 적합한 함수를 선택해서 사용할 수

있다. szcmp() 함수는 s와 t의 두 가지 문자열을 서로 비교한다. ‘ss’ 매개변수는 문자열 s의 시작 위치를 가리키는 포인터이며 ‘ts’는 문자열 t의 시작 위치를 가리키는 포인터이다. ‘sx’와 ‘tx’는 문자열 s와 t의 포인터 종결자이다.

```
int n;
chr *ss = &bigstr[2]; // 문자열 s의 시작 위치
chr *sx = &bigstr[4]; // 문자열 s의 포인터 종결자
chr *ts = &bigstr[5]; // 문자열 t의 시작 위치
chr *tx = &bigstr[7]; // 문자열 t의 포인터 종결자
n = szcmp("is",ss,sx); // n == 0이면 문자열이 동일함
n = szcmp2(ss,sx,ts,tx); // n == 0이면 문자열이 동일함
```

탐색

표준적인 정방향 문자 탐색 루틴 역방향 탐색 루틴, 부문자열(substring) 탐색 루틴 외에도 빠른 문자 탐색을 용이하게 해주는 함수들이 제공된다. 이런 루틴들은 플래그 테이블(flag table)과 플래그의 포인터를 매개변수로 받는다. szfskip()과 szftill()과 같은 탐색 루틴은 문자 집합의 첫 번째 문자를 발견할 때까지 문자열을 탐색하거나 건너뛴다.

플래그 테이블은 각 문자 당 하나의 정수를 가지는 정수 배열이다. 각 정수는 각 문자에 대한 플래그(flag)를 저장하는데 사용된다. 플래그는 문자가 어떤 문자 집합에 속해있는가를 표시한다. 다음의 코드는 ASCII 문자에 대한 간단한 플래그 테이블을 설정하고 있다.

```
// 하위 24 비트를 사용하는 문자 집합 정의
#define M_ALPHA 0x0000001
#define M_NUMERIC 0x0000002
#define NUM(_a_) (sizeof(_a_)/sizeof(_a_[0]))
int i,flagtable[128]; // ASCII에서는 128개짜리 배열로 충분함
// 플래그 테이블 초기화
for(i=0;i
flagtable[i]=0; // 테이블 전체를 0으로 채움
flagtable[0]=NUM(flagtable); // 첫 번째 요소에는 반드시 테이블의 크기가 들어가야 함
// 문자 집합 정의 - 문자가 1~127의 범위 안에 있다고 가정
for(i='a';i<='z';i++) // 알파벳 소문자
flagtable[i]|=M_ALPHA;
for(i='A';i<='Z';i++) // 알파벳 대문자
```

```
flagtable[i]|=M_ALPHA;
for(i='0';i<='9';i++) // 숫자
flagtable[i]|=M_NUMERIC;
```

각 문자마다 플래그 비트를 설정하면 문자열 루틴에서 빠른 테이블 찾아보기(lookup)나 비트 검사(bit test)를 통해 문자가 특정 문자 집합이나 다른 집합에 속하는지 결정할 수 있게 된다. 상위 8개의 플래그 비트는 라이브러리의 내부 사용 용도로 예약되어 있다.

단어 검색

문자열 버퍼에서 단어를 찾는 것은 버퍼 전체에서 알파벳 문자를 발견할 때까지 정방향으로 탐색하면 되며 이것이 단어의 시작 위치가 된다. 루틴에서는 그 다음에 알파벳과 숫자가 아닌 문자를 발견할 때까지 나머지 문자를 탐색하며 이것이 단어의 끝 위치가 된다. 다음 코드는 이 작업에 대한 예제이다.

```
// 탐색
chr *ss,*sx;
ss="a bc de₩n₩rz₩n₩r"; // 탐색할 예제 버퍼
// 첫 번째 알파벳이 나올 때까지 정방향 탐색, 그 다음에는 알파벳, 숫자가 아닌 최초의 문자가 나올 때까지 무시
// 그 사이에 있는 모든 것은 단어로 간주함
ss=szftill(flagtable,ss,NULL,M_ALPHA); // 알파벳이 나올 때까지 탐색
sx=szfskip(flagtable,ss,NULL,M_ALPHA|M_NUMERIC);  // 알파벳, 숫자 넘김
```

위의 코드 조각에서 ‘ss’와 ‘sx’는 문자열 내에서 단어의 시작과 끝 위치를 가리키는 포인터를 가지고 있다. 이렇게 해서 발견한 단어는 szmap()을 통해 다음과 같이 토큰(token) 값으로 맵핑(mapped)할 수 있다.

```
// ss는 단어의 시작 위치, sx는 단어의 끝 위치
// 단어를 토큰으로 맵핑
switch(szmap(tokens,ss,sx))
{
case TOKEN_A:
printf("TOKEN_A ");
break; // TOKEN_A
case TOKEN_BC:
printf("TOKEN_BC ");
break; // TOKEN_BC
case TOKEN_DE:
printf("TOKEN_DE ");
break; // TOKEN_DE
case TOKEN_Z:
printf("TOKEN_Z ");
break; // TOKEN_Z
default:
printf("* unknown token ");
break;
} // switch
```

위에서 사용한 토큰은 열거형(enumerations)으로 정의되어 있으며 그에 해당하는 문자열 테이블은 다음과 같다.

```
enum { // 토큰 열거형
TOKEN_RESERVED, // 0번 토큰 예약
TOKEN_A,
TOKEN_BC,
TOKEN_DE,
TOKEN_Z,
}; // TOKEN_
chr *tokens[] = { // 문자열 테이블
```

"", // 0번 토큰 예약
"a", // TOKEN_A
"bc", // TOKEN_BC
"de", // TOKEN_DE
"z", // TOKEN_Z
NULL // 토큰 목록 끝
}; // tokens[]

플래그 테이블과 큰 문자 집합

참고로, 이곳에서 소개된 플래그 테이블은 8비트 ASCII에서 잘 동작하며 16비트 문자 집합에서 사용할 수도 있다. 하지만 현재 구현된 루틴에서는 플래그 테이블의 크기와 문자 집합의 문자 개수가 일치해야 하므로 변형된 접근법을 사용할 때에는 더 큰 문자 집합을 사용하는 것이 더 좋을 수도 있다.

숫자 변환

문자열을 숫자값으로 변환하는 것은 상당히 흔한 작업 중에 하나이다. 그리고 sztobin() 함수를 사용하면 해당 작업을 수행할 수 있다. 이 함수의 매개변수로는 문자열의 시작 위치, 문자열의 포인터 종결자, 변환 기수(conversion base), 목표 형식, 변환된 값을 저장할 목표 변수의 포인터를 넘겨주면 된다. 그러면 루틴은 변환 후에 첫 번째 문자의 포인터를 반환할 것이다. 목표 형식은 목표 변수의 형식을 나타내는 아래의 목록에 표시된 문자 코드이다.

```
// 목표 형식
// 'i' = int 포인터
// 'u' = unsigned int 포인터
// 'f' = float 포인터
// 'd' = double 포인터
```

예를 들어 다음의 코드는 문자열을 실수값으로 변환한다.

```
float fval;
chr *ss;
ss = sztobin("1.23;",NULL,10,'f',&fval); // fval == 1.23 ss == ";"
```

아래에서 설명되는 문자열 형식화(formatting) 루틴인 szfmt()를 사용하면 숫자값을 문자열로 변환할 수 있다.

형식화

라이브러리에는 sprintf와 유사한 형식화 루틴인 szfmt()가 포함되어 있다. 이 루틴은 기본 sprintf의 형식화 명령을 지원한다. 이 루틴은 버퍼 생성 루틴 중 하나이며 규칙에 따라 목표 버퍼의 바이트(byte) 단위 크기를 넘겨줘야 하고 반환값으로는 생성된 문자열의 길이를

반환한다.

다음의 예제는 szfmt()를 사용해 버퍼를 생성하는 방법을 보여준다.

```
int n;
chr buf[32],*ss;
ss = "world";
n = 2007;
n = szfmt(buf,sizeof(buf),"Hello %s %d.",ss,n); // n == 17
```

결론

표준 C 라이브러리의 문자열 처리 접근법은 안전하지 않고 느리며, 간혹 멀티 스레드(multi-threaded) 프로그래밍에 부적합한 문제성 많은 코드가 되어버리기도 한다. 본 아티클에서는 그런 문제를 극복하기 위해 포인터 종결자 기반의 새로운 문자열 처리 라이브러리를 소개했다.

본 라이브러리는 C 또는 C++ 프로젝트에서 직접 사용하거나 고수준 클래스(higher level class)의 기반으로 사용할 수 있다. 최대 효과를 얻으려면 프로그램의 모든 문자열 처리에서 포인터 종결 문자열을 지원하도록 해야 할 것이다.

나는 다른 라이브러리에서 소개된 모든 기능들을 그대로 구현하려고 하지는 않았다. 그보다는 간결하고 쉬운 확장 가능한 라이브러리의 구현에 주력했다. 그리고 본 라이브러리의 변형판을 지난 5년간 도구(tools), 프로젝트, 게임에서 사용해왔으며 그로 인해 큰 효과를 얻을 수 있었다.

참조

[1] Dennis M. Ritchie, C 언어의 개발(The Development of the C

Language)(Association for Computing Machinery, Inc., 1993)(2007/7/13 현재 http://www.cs.bell-labs.com/who/dmr/chist.html 에서 볼 수 있음)

라이브러리의 버퍼 생성 루틴은 다음과 같은 규칙을 따른다.

- 길이가 0이 아닌 버퍼를 생성할 때는 항상 0 종결자를 강제한다.
- 버퍼 생성 명령에는 버퍼의 바이트(byte) 단위 크기를 넘겨줘야 한다.
- 문자 길이 반환값에는 종결자가 포함되지 않는다.

본 아티클에서 사용된 함수의 원형(prototype)

```
extern int szlen(chr *ss); // 문자열에 있는 문자의 개수
extern int szlen2(chr *ss,chr *sx);
extern int szsize(chr *ss); // 문자열과 종결자의 바이트 단위 크기
extern int szsize2(chr *ss,chr *sx);
extern int szcpy(chr *dst,int dstsize,chr *ss,chr *sx);
extern int szcmp(chr *ss,chr *ts,chr *tx); // 문자열 s는 0으로 끝나는 문자열
extern int szcmp2(chr *ss,chr *sx,chr *ts,chr *tx);
extern chr* szfskip(int *flagtable,chr *ss,chr *sx,int flags); // 스킵 탐색(skip matching)
extern chr* szftill(int *flagtable,chr *ss,chr *sx,int flags); // 틸 탐색(till match)
extern int szmap(chr *table[],chr *ss,chr *sx);
extern chr* sztobin(chr *ss,chr *sx,int base,chr dsttype,void *dst);
extern int szfmt(chr *dst,int dstsize,chr *fmt,...);
```