GCC obstack

정원교
weongyo@hotmail.com

2004년 2월 23일

목 차

제 1 절 11 주째 강의를 시작하며 1

제 2 절 Obstacks 1
- 2.1 Obstacks 생성하기 . . . 2
- 2.2 Obstacks 을 사용하기 위한 준비 . . . 4
- 2.3 Obstack 에서의 할당(allocation) . . . 5
- 2.4 Obstack 내의 object 들 해제(freeing) . . . 6
- 2.5 Obstack 함수들과 매크로들 . . . 6
- 2.6 성장하는 object 들 . . . 7
- 2.7 특히 빠르게 성장하는 object 들 . . . 8
- 2.8 Obstack 의 상태 . . . 9
- 2.9 Obstack 들의 데이타 정렬(alignment) . . . 10
- 2.10 Obstack Chunk 들 . . . 10
- 2.11 Obstack 함수들의 요약 . . . 11
- 2.12 Obstack 함수 혹은 매크로들의 내부 . . . 13

제 3 절 12 주 강의를 마치며 16

제 1 절 11 주째 강의를 시작하며

그 동안 강의를 작성하면서 높임말을 주로 사용하여 작성하였는데, 이제부터는 그러지 않을 생각입니다. "... 시작하며", "... 마치며" 와 같이 글의 시작과 끝을 나타내는 부분만 평소대로 사용하고 나머지 부분은 반말(_._;)을 사용하도록 하겠습니다.

이번주에 살펴볼 내용은 GCC 에서 사용하는 obstack 에 관한 것입니다. 하지만 이 문서의 경우 이미 GCC 메뉴얼중 'libiberty' 에 자세히 나와 있습니다. 그렇기 때문에 그것을 놔두고 새로이 쓰는 것은 않좋기 때문에 그 문서를 기반으로 해서 제 나름대로의 내용을 추가하는 형식으로 작성하도록 하겠습니다.

끝으로 'libiberty' 문서의 번역판이 되지 않기를 간절히 바랍니다. :-)

제 2 절 Obstacks

obstack 은 object 들의 스택을 포함하는 메모리 pool 이다. 여러분은 여러개로 분리된 obstack 들을 만든 다음 지정된 obstack 들내에 object 들을 할당할 수 있다. 각 obstack 범위에서, 마지막에 할당된 object 는 항상 반드시 처음 free 되는 것이지만 '개별적 obstack' 들은 다른 것들로부터 독립적이다.

Free 순서에 관한 이 제약말고는 obstack 들은 전체적으로 일반적이다: obstack 은 어느 크기의 object 들을 몇 개라도 저장할 수 있다. 그들은 매크로로 실행되며 그래서 object 가 작은 한 보통 매우 빠르다. 그리고 각 object 당 상층에 있는 공간은 적당한 경계점에서 각 object 를 시작하는데 필요해서 채워넣은 (padding) 것이다.

2.1 Obstacks 생성하기

obstacks 를 다루기 위한 유틸리티들은 헤더파일 'obstack.h'에 정의되어 있다.

struct obstack [Data Type]

obstack 은 타입 struct obstack 의 데이터 구조로 표현된다. 이 구조는 작은 고정 크기를 가진다; obstack 의 상태와 obstack 들이 할당되어 있는 공간에서 어떻게 찾을 지를 기록한다. 어떠한 object 그 자신을 포함하지 않는다. 당신은 직접적으로 그 구조체의 내용을 접근하려고 하면 안되다. 오직 이 장에서 설명하는 함수들을 사용하라.

아래 그림은 struct obstack 구조체에 대해 개략적으로 나타낸 그림이다.

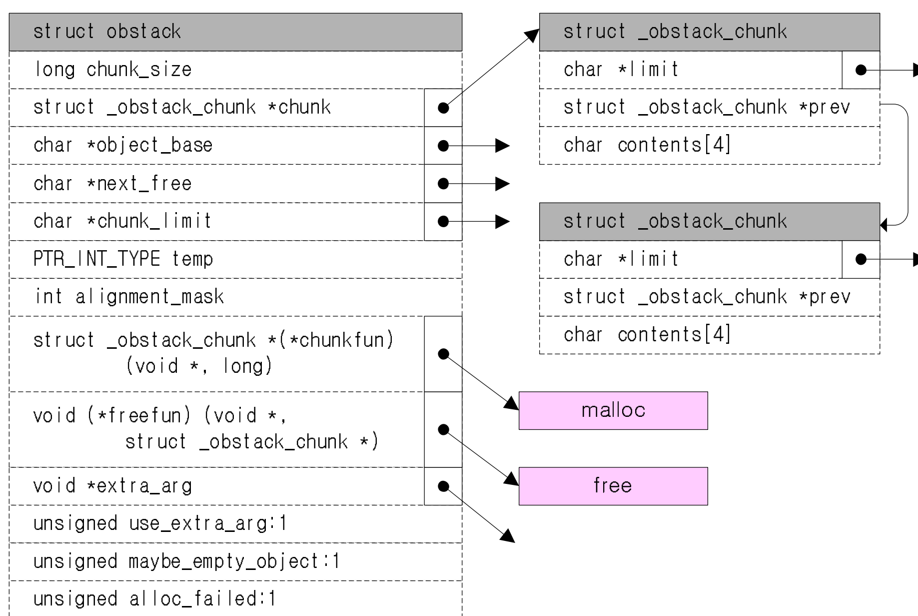

그림 1: struct obstack 구조체

위 구조체를 구성하는 요소와 그 기능에 대해 살펴보자.

long *chunk_size*;

chunk 들에 할당될 우선적인 크기

struct _obstack_chunk **chunk*;

현재 struct obstack_chunk 의 주소

char **object_base*;

우리가 형성중인 object 의 주소

char **next_free*;

현재 object 에 넣을 다음 문자가 있는 곳

char **chunk_limit*;

현재 chunk 바로 뒤 문자의 주소

PTR_INT_TYPE *temp*;

몇몇 매크로들을 위한 임시공간

int *alignment_mask*;

각 object 에 대한 alignment mask

struct _obstack_chunk *(**chunkfun*) (void *, long);

chunk 할당 관련 사용자 지정 함수

void (**freefun*) (void *, struct _obstack_chunk *);

chunk 해제 관련 사용자 지정 함수

void **extra_arg*;

chunk alloc/dealloc func 들의 첫번째 인자

unsigned *use_extra_arg:1*;

chunk alloc/dealloc func 들이 여분의 인자를 가진다

unsigned *maybe_empty_object:1*;

현재 chunk 가 길이가 0 인 object 를 가질 가능성이 있다. 이것은 우리가 그것을 대체할 더 큰 chunk 를 할당한다면 chunk 를 해제하는 것을 방지한다

unsigned *alloc_failed:1*;

더 이상 사용되지 않음, 오류시 handler 를 이제 호출하는데, 바이너리 호환성을 위해 유지시키고 있다.

이제 다른 구조체에 대해서 살펴봐야 하는데, struct obstack 구조체내에 포함되는 구조체인 struct _obstack_chunk 가 그것이다. 구조체에 대해서는 그림 1 에 잘나와 있으니 생략하고 각 구성요소의 기능에 대해 언급하도록 하겠다.

char *limit;

이 chunk 끝을 지나 첫번째 바이트, 즉 현재 chunk 의 주소에 chunk_size 크기를 더한 값이다.

struct _obstack_chunk *prev;

이전 chunk 의 주소 혹은 NULL 값을 가진다.

char contents[4];

object 들은 여기서 시작한다.

당신은 struct obstack 타입으로 변수들을 선언할 수 있고 obstack 들처럼 그들은 사용할 수 있다. 또한 다른 종류의 object 처럼 동적으로 obstack 들을 할당할 수 있다. obstack 의 동적 할당은 여러분의 프로그램으로 하여금 여러개의 다른 stack 들을 가질 수 있도록 한다. (당신은 심지어 한 obstack 안에 다른 obstack 구조를 할당할 수 있다. 하지만 이것은 그리 유용하지는 않다.)

obstack 들과 함께 작동하는 모든 함수들은 어떤 obstack 을 사용할지 당신이 정해줘야 한다. 당신은 타입 **struct obstack *** 의 포인터로 이것을 할 수 있다. 따라서 우리가 종종 "**obstack**" 라고 말할 때 엄격히 말하면 그것은 포인터와 같다.

obstack 안의 object 들은 *chunks* 라고 불리는 큰 block 들 안에 pack 되어 있다. **struct obstack** 구조는 현재 사용 중인 chunks 의 chain 을 가르킨다.

obstack 라이브러리는 당신이 앞의 chunk 에 맞추길 원하지 않는 object 를 할당할 때마다 새로운 chunk 를 만든다. obstack 라이브러리가 chunk 들을 자동으로 관리하기 때문에 당신은 그것들에 많은 주의를 쏟을 필요는 없지만 obstack 라이브러리가 chunk 를 얻기 위해 사용해야 하는 함수를 공급해 줄 필요는 있다. 보통 당신은 직접적이거나 간접적이게 **malloc** 을 사용하는 함수를 공급한다. 또한 당신은 chunk 를 해제하기 위한 함수를 공급해야만 한다. 이 문제들은 다음 절에 설명되어 있다.

2.2 Obstacks 을 사용하기 위한 준비

당신이 obstack 을 사용하려고 계획했다면 각 소스파일 안에 'obstack.h'라는 헤더파일을 포함해야만 한다. 이것은 다음과 같다.

#include <obstack.h>

또한 source 파일에 매크로 obstack_init 를 사용한다면 obstack 라이브러리에 의해 불려질 매크로 혹은 두 개의 함수를 선언하거나 정의해야만 한다. 하나는 **obstack_chunk_alloc** 으로써 object 들이 pack 된 메모리의 chunk 들을 할당하는데 사용된다. 다른 하나는 **obstack_chunk_free** 로써 chunk 에서 object 들을 해제하고 chunk 를 반환하는데 사용한다.

보통 이들은 **xmalloc** 이라는 매개자를 거쳐 **malloc** 을 사용하도록 정의되어 있다. (*The GNU C Library Reference Manual* 에서의 "Unconstrained Allocation" 절을 참조) 이것은 밑에 있는 두 개의 매크로 정의로 한다.

#define obstack_chunk_alloc xmalloc
#define obstack_chunk_free free

비록 당신이 obstack 들을 사용하기 위해 얻어낸 저장공간은 실제로 **malloc** 을 통해 할당된 것이지만 메모리의 큰 블럭을 다룰 때는 **malloc** 가 덜 호출되기 때문에 obstack 을 사용하는 것이 더 빠르다. 자세한 설명은 2.10 [Obstack Chunk 들] 절, 10 쪽을 참고해라.

실행시 프로그램이 obstack 으로 타입 struct obstack 인 object 를 사용하기 전에 반드시 obstack_init 를 호출해서 obstack 을 초기화해야만 한다.

int **obstack_init** (struct obstack **obstack_ptr*) [Function].

> object 들의 할당을 위해 obstack *obstack_ptr* 를 초기화한다. 이 함수는 obstack 의 **obstack_chunk_alloc** 함수를 호출한다. 만약 메모리 할당이 실패할 경우 **obstack_alloc_failed_handler** 가 가르키는 함수가 호출된다. **obstack_init** 함수는 항상 1 을 반환한다. (호환성 공지: 이전 버전의 obstack 의 경우 만약 할당에 실패했을 경우 0 을 반환 했었다.)

아래에 어떻게 obstack 을 위한 공간을 할당하고 그것을 초기화하는지에 대한 두 예가 있다. 첫번째, static 변수인 obstack 일 경우:

```
static struct obstack myobstack;
...
obstack_init (&myobstack);
```

두번째, 그 자신이 동적으로 할당되는 obstack 일 경우:

```
struct obstack *myobstack_ptr
  = (struct obstack *) xmalloc (sizeof (struct obstack));

obstack_init (myobstack_ptr);
```

obstack_alloc_failed_handler [Variable].

> 이 변수의 값은 **obstack_chunk_alloc** 가 메모리를 할당하는데 실패하였을 때 **obstack** 이 사용할 함수에 대한 포인터이다. 기본 행동은 메세지를 출력하고 멈추는 것이다. 당신은 적어도 **exit** (*The GNU C Library Reference Manual* 중 "Program Termination" 절을 참고) 혹은 **longjmp** (*The GNU C Library Reference Manual* 중 "Non-Local Exits" 절 참고) 중 하나를 호출하여야 하고 return 해서는 안된다.

```
void my_obstack_alloc_failed (void) ...
obstack_alloc_failed_handler = &my_obstack_alloc_failed;
```

2.3 Obstack 에서의 할당(allocation)

obstack 에 object 를 할당하기 위한 가장 직접적인 방법을 obstack_alloc 을 사용하는 건데 이것은 거의 malloc 과 같다.

void * **obstack_alloc** (struct obstack **obstack-ptr*, int *size*) [Function].

> 이것은 obstack 형태의 *size* 크기 만큼의 초기화되지 않은 block 을 할당하고 그것의 주소를 반환한다. 여기서 *obstack-ptr* 은 block 내에 할당하기 위한 어떤 obstack 을 지정한다; obstack 을 나타내는 **struct obstack** 의 주소이다. 각각의 obstack 함수들과 매크로들은 여러분이 첫번째 인자로 *obstack-ptr* 를 지정해주기를 요구한다. 이 함수는 메모리의 새로운 chunk 를 할당하는게 필요하다면 obstack 의 **obstack_chunk_alloc** 함수를 호출한다; **obstack_chunk_alloc** 의 메모리 할당이 실패할 경우 **obstack_alloc_failed_handler** 를 호출한다.

예를 들면, 아래는 변수 **string_obstack** 인 지정 obstack 에 문자열 *str* 의 복사본을 할당하는 함수이다.

```
struct obstack string_obstack;

char *
copystring (char *string)
{
  size_t len = strlen (string) + 1;
  char *s = (char *) obstack_alloc (&string_obstack, len);
  memcpy (s, string, len);
  return s;
}
```

정해진 내용을 가지는 block 을 할당하기 위해서는 아래와 같이 선언되어 있는 **obstack_copy** 를 사용하라.

void * **obstack_copy** (struct obstack **obstack-ptr*, void **address*, int *size*) [Function]

> 이것은 block 을 할당하고 *address* 로 시작하는 *size* 바이트 크기만큼의 복사함으로써 초기화한다. 만약 **obstack_chunk_alloc** 에 의한 메모리 할당이 실패한다면 **obstack_alloc_failed_handler** 를 호출한다.

void * **obstack_copy0** (struct obstack **obstack-ptr*, void **address*, int *size*) [Function]

> **obstack_copy** 와 거의 같지만, NULL 문자를 포함하는 여분의 바이트를 추가한다. 이 여분의 바이트는 인자 *size* 에 반영되지 않는다.

obstack_copy0 함수는 널 문자로 끝나는 문자열과 같은 문자 열들을 obstack 에 넣기 편리하다. 아래에 사용에 관한 예제가 있다:

```
char *
obstack_savestring (char *addr, int size)
{
  return obstack_copy0 (&myobstack, addr, size);
}
```

malloc 를 사용한 앞의 **savestring** 예제와 비교해 보라. (*The GNU C Library Reference Manual* 의 "Basic Allocation" 절 참고)

2.4 Obstack 내의 object 들 해제(freeing)

obstack 내에 할당된 object 를 해제 (free) 하기 위해서는 **obstack_free** 함수를 사용하라. obstack 은 object 들의 stack 이기 때문에, 하나의 object 를 해제하는 것은 자동적으로 같은 obstack 에 최근 할당된 모든 다른 object 들을 해제한다.

void **obstack_free** (struct obstack **obstack-ptr*, void **object*) [Function]

> 만약 *object* 가 NULL 포인터라면, obstack 내에 할당된 모든 것들을 해제한다. 그렇지 않다면 *object* 는 반드시 obstack 내에 할당된 object 의 주소여야만 한다. 그러면 *object* 는 해제되고, 그 *object* 이후 *obstack* 안에 할당된 모든 것도 따라 해제된다.

object 가 NULL 포인터라면 결과가 초기화되지 않은 obstack 임을 알아라. obstack 내의 모든 메모리를 해제하지만 나중의 할당을 위해 남겨놓고 싶다면 obstack 상에 할당된 첫번째 object 의 주소로 **obstack_free** 를 호출하라:

```
obstack_free (obstack_ptr, first_object_allocated_ptr);
```

obstack 내 object 들이 chunk 들로 그룹화된 것을 재호출하라. chunk 내 모든 object 들이 해제가 되었을 때, obstack 라이브러리가 자동으로 chunk 를 해제한다. (2.2 절 [Obstacks 을 사용하기 위한 준비], 4 쪽을 참고) 그러면 다른 obstack 들이나 obstack 할당이 아닌 것들이 chunk 의 공간을 재사용할 수 있다.

2.5 Obstack 함수들과 매크로들

obstack 들의 사용을 위한 인터페이스는 컴파일러에 따라 함수나 매크로로 정의되어진다. obstack 기능은 ISO C 와 traditional C 를 포함하여 모든 C 컴파일러들에서 동작하지만 만약 당신이 GNU C 보다 다른 컴파일러를 사용할 계획이라면 많은 주의를 해야 한다.

만약 당신이 오래된 형태의 비-ISO C 컴파일러를 사용중이라면 모든 obstack "함수들"은 실제로 단지 매크로로 정의된다. 함수처럼 그러한 매크로들을 사용할 수 있겠지만 당신은 그것들을 다른 방법으로 사용할 수 없다. (예를 들면, 당신은 그것의 주소를 얻을 수 없다.)

매크로를 호출하는 것은 특별한 주의가 필요하다: 즉, 첫번째 operand (obstack 포인터) 가 부작용을 포함하고 있다고 말 할 수 있을지도 모르는데, 그것은 여러 번 연산될 수 있기 때문이다. 예를 들면, 만약 당신이 아래와 같이 코드를 작성한다면:

```
obstack_alloc (get_obstack (), 4);
```

당신은 **get_obstack** 가 여러번 호출된다는 것을 알 것이다. 만약 당신이 obstack 포인터 인자로써 ***obstack_list_ptr++** 를 사용한다면 증가가 여러번 일어날 수 있기 때문에 당신은 아주 이상한 결과를 얻을 것이다.

ISO C 에서는 각 함수는 '매크로 정의'와 '함수 정의' 둘다 가지고 있다. 함수 정의는 만약 당신이 그것을 호출하지 않고 함수의 주소를 사용할 경우 사용된다. 보통의 호출은 기본으로 매크로 정의를 사용하지만 당신은 괄호 안에 함수의 이름을 씀으로써 함수 정의를 요청할 수 있다. 아래처럼 말이다:

```
char *x;
void *(*funcp) ();
/* 매크로를 사용. */
x = (char *) obstack_alloc (obptr, size);
/* 함수를 호출. */
x = (char *) (obstack_alloc) (obptr, size);
/* 함수의 호출을 얻음. */
funcp = obstack_alloc;
```

이것은 표준 라이브러리 함수 관련 ISO C 내에 존재하는 것과 같은 상황이다. *The GNU C Library Reference Manual* 의 "Macro Definitions" 절을 참조.

경고: 당신이 매크로들을 사용할 때는 ISO C 에서 조차도 반드시 첫번째 operand 내에서의 부작용을 피하는 것에 주의를 해야 한다. 만약 GNU C 컴파일러를 사용한다면 GNU C 내의 여러 언어로의 확장 기능이 오직 한번 각 인자를 실행하도록 그 매크로를 정의하는 것을 허락하기 때문에 이러한 주의는 필요없다.

2.6 성장하는 object 들

obstack chunk 들내의 메모리는 순차적으로 사용되기 때문에, object 들을 차례로 쌓아올리고 한번에 object 의 끝에 하나 혹은 여러 바이트를 더하는 것이 가능하다. 이러한 기술로 당신은 그 object 의 끝에 다다를 때까지 그 대상물에 얼마만큼의 데이타를 넣을 수 있는 알 필요가 없다. 우리는 이 기술을 "성장하는 object 들"라고 부른다. 성장하는 object 에 데이타를 더하는 특별한 함수들은 이 절에서 설명된다.

당신은 어떤 object 를 성장시키고자 할 때 어떠한 특별한 조치도 할 필요가 없다. 그 object 에 데이타를 추가하는 함수들 중 하나를 사용할 때 자동으로 그것이 시작된다. 하지만 object 가 끝날 때는 분명하게 말해줄 필요가 있다. 이것은 **obstack_finish** 함수에서 이루어진다.

이렇게 해서 만들어진 object 의 실제 주소는 object 가 끝날 때까지 알려지지 않는다. 그 때까지는 object 가 반드시 새로운 chunk 에 복사되어져야 할 때 많은 데이타를 추가할 수 있을 가능성을 항상 남겨둡니다.

obstack 이 성장하는 object 로 사용되는 동안은 당신은 다른 object 의 보통 할당에 그것을 사용할 수 없다. 만약 당신이 그렇게 한다면 성장하는 object 에 이미 추가된 공간은 다른 object 의 부분이 될 것입니다.

void **obstack_blank** (struct obstack **obstack-ptr*, int *size*) [Function]

성장하는 object 에 추가하기 위한 가장 기초적인 함수는
obstack_blank 이며, 이것은 대상물을 초기화하지 않고 공간을 추가한다.

void **obstack_grow** (struct obstack **obstack-ptr*, void **data*, int *size*) [Function]

초기화된 공간의 한 블럭을 추가하기 위해서는 **obstack_copy** 의 성장하는-object 유사물인 **obstack_grow** 를 사용하라. 이것은 성장하는 object 에 *size* 바이트의 데이타를 추가하고 *data* 로부터 그 내용을 복사한다.

void **obstack_grow0** (struct obstack **obstack-ptr*, void **data*, int *size*) [Function]

이것은 **obstack_copy0** 의 성장하는-object 유사물이다. 이것은 추가적인 NULL 문자가 따라오는 *data* 에서 복사한 *size* 바이트를 추가한다.

void **obstack_1grow** (struct obstack **obstack-ptr*, char *c*) [Function]

한번에 하나의 문자를 추가하려면 함수 **obstack_1grow** 를 사용하라. 성장하는 대상물에 *c* 를 담아둘 1 바이트를 추가한다.

void **obstack_ptr_grow** (struct obstack **obstack-ptr*, void **data*) [Function]

하나의 포인터 값을 추가하기 위해서는 **obstack_ptr_grow** 함수를 사용할 수 있다. *data* 값을 포함하는 **sizeof (void *)** 바이트를 추가한다.

void **obstack_int_grow** (struct obstack **obstack-ptr*, int *data*) [Function]

타입 int 인 하나의 값은 **obstack_int_grow** 함수를 사용하여 추가될 수 있다. 성장하는 object 에 **sizeof (int)** 를 더하고 *data* 의 값으로 그것을 초기화한다.

void * **obstack_finish** (struct obstack **obstack-ptr*) [Function]

> 당신이 object 를 자라게 하는 것을 끝마쳤을 때 **obstack_finish** 함수를 사용하여 그것을 모두 닫고 그것의 마지막 주소를 반환하라.
> 당신이 그 object 를 완료하면, obstack 은 보통의 할당이나 다른 object 의 자라는 것에 이용될 수 있다.
> 이 함수는 **obstack_alloc** 와 같은 조건하에서는 NULL 포인터를 반환할 수 있다. (2.3 절 [Obstack 에서의 할당(allocation)], 5 쪽을 참조)

당신이 하나의 object 을 성장시켜서 만들게 되면, 그것이 얼마 만큼 길어 졌는지를 나중에 알 필요가 생길 것이다. 당신은 그 object 가 성장하는 동안에는 그 과정을 추적할 필요가 없다. 왜냐하면 당신은 아래에 선언되어 있는 **obstack_object_size** 함수를 사용하여 그 object 의 완료 직전에 obstack 으로 부터의 길이를 알 수 있기 때문이다.

int **obstack_object_size** (struct obstack **obstack-ptr*) [Function]

> 이 함수는 성장하는 object 의 현재 크기를 바이트 수로 반환한다. 이 함수는 object 이 완료되기 *전에* 호출하여야 함을 기억하라. 완료되고 난 다음에는 이 함수는 zero 를 반환할 것이다.

만약 당신이 object 를 자라나게 한 후 나중에 이를 취소하고 싶다면, 당신은 아래와 같이 끝내고 해제하라:

```
obstack_free (obstack_ptr, obstack_finish (obstack_ptr));
```

이것은 아무런 object 도 자라게 하지 않을 때에는 아무런 효과가 없는 것이다.

현재의 object 을 작아지게 만들고 싶으면 음수의 size 인수로 **obstack_blank** 를 사용할 수 있다. 그러나 그 object 를 0 보다 작게 만들려고 하지는 말아야 한다. 그렇게 되면 무슨 일이 일어날지 알 수가 없다.

2.7 특히 빠르게 성장하는 object 들

성장하는 object 들을 위한 보통의 함수들은 현재의 chunk 에 새로운 성장을 위한 방(room)이 있는지를 검사해야 하는 overhead 를 초래한다. 만약 당신이 object 를 조금씩 단계별로 만들고자 한다면 이 overhead 는 큰 문제가 될 수 있다.

당신은 검사없이 object 를 성장시키는 특별한 "빠른 성장" 함수들을 사용함으로써 overhead 를 줄일 수 있다. 견고한 프로그램을 하기 위해서는 당신 스스로 검사를 해야 한다. 만약 당신이 object 에 data 를 추가할 때 마다 가장 간단한 방법으로 검사를 하더라도 보통의 성장 함수들이 하는 일이기 때문에 별로 이득이 없을 것이다. 하지만 검사 횟수는 줄이되 더 효율적으로 검사를 배열할 수 있다면 프로그램을 더 빠르게 할 수 있을 것이다.

obstack_room 함수는 현재 chunk 내에 사용 가능한 방(room) 의 양을 반환한다. 그것은 다음과 같이 선언되어 있다:

int **obstack_room** (struct obstack **obstack-ptr*) [Function]

> 이것은 빠른 성장 함수들을 사용하여 obstack *obstack* 내의 현재 성장하는 object (또는 이제 막 시작된 object) 가 안전하게 추가될 수 있는 바이트의 크기를 반환한다.

이것을 통해서 방(room) 이 있는 지를 알 수 있으므로 당신은 성장하는 object 에 data 를 추가하기 위해 다음의 빠른 성장 함수들을 사용할 수 있다.

void **obstack_1grow_fast** (struct obstack **obstack-ptr*, char *c*) [Function]

> **obstack_1grow_fast** 함수는 obstack *obstack-ptr* 내 성장하는 object 에 문자 *c* 를 포함하는 1 바이트를 추가한다.

void **obstack_ptr_grow_fast** (struct obstack **obstack-ptr*, void **data*) [Function]

obstack_ptr_grow_fast 함수는 obstack *obstack-ptr* 내 성장하는 object 에 *data* 값을 포함하는 **sizeof (void *)** 바이트를 추가한다.

void **obstack_int_grow_fast** (struct obstack **obstack-ptr*, int *data*) [Function]

obstack_int_grow_fast 함수는 obstack *obstack-ptr* 내 성장하는 object 에 *data* 값을 포함하는 **sizeof (int)** 바이트를 추가한다.

void **obstack_blank_fast** (struct obstack **obstack-ptr*, int *size*) [Function]

obstack_black_fast 함수는 obstack *obstack-ptr* 내 성장하는 object 에 그것을 초기화하는 것 없이 *size* 바이트를 추가한다.

당신이 **obstack_room** 을 사용하여 공간을 체크하여 당신이 추가하기를 원하는 만큼의 충분한 공간이 없을 때는, 빠른 성장 함수들은 안전하지 못하다. 이러한 경우에는 대신 적합한 보통 성장 함수를 사용하라. 곧바로 이것은 object 를 새로운 chunk 에 복사할 것이다: 그럼 다시 사용 가능한 방(room) 이 생길 것이다.

그래서 당신이 보통 성장 함수를 사용할 때 마다 **obstack_root** 을 사용하여 앞으로 충분한 공간이 있는지를 검사하라. 일단 object 가 새로운 chunk 에 복사되고 나면 다시 풍부한 공간이 생길 것이고, 그래서 프로그램은 다시 빠른 성장 함수들을 사용하기 시작할 것이다.

여기 예제가 있다:

```
void
add_string (struct obstack *obstack, const char *ptr, int len)
{
  while (len > 0)
    {
      int room = obstack_room (obstack);
      if (room == 0)
        {
          /* 충분한 공간이 없음. 새로운 chunk 로 복사해서
             방을 만들수 있도록 천천히 문자 하나를
             추가한다. */
          obstack_1grow (obstack, *ptr++);
          len--;
        }
      else
        {
          if (room > len)
            room = len;
          /* 우리가 방을 가지고 있는 동안은 빠른 추가. */
          len -= room;
          while (room-- > 0)
            obstack_1grow_fast (obstack, *ptr++);
        }
    }
}
```

2.8 Obstack 의 상태

다음 함수들은 현재 obstack 내의 할당 상태에 관한 정보를 제공한다. 당신은 object 가 여전히 성장하고 있는지를 아래의 것들을 이용하여 알 수 있다.

void * **obstack_base** (struct obstack **obstack-ptr*) [Function]

이 함수는 *obstack-ptr* 내 현재 성장하는 object 의 임시 시작 주소를 반환한다. 만약 당신이 즉시 object 를 종결한다면, 그것은 그 주소를 가질 것이다. 만약 당신이 그것을 더 크게 만든다면 그것은 현재 chunk 를 넘어설 수 있다–그럼 그 주소는 변할 것이다!
만약 아무런 object 가 성장하고 있지 않다면 이 값은 당신이 할당한 다음 object 가 어디서부터 시작할 것인가를 알려준다.(다시 한번 더 가정하건데 그것은 현재 chunk 에 맞춰져 있다.)

void * **obstack_next_free** (struct obstack **obstack-ptr*) [Function]

이 함수는 obstack *obstack-ptr* 의 현재 chunk 에서 해제된 첫 바이트의 주소를 반환한다. 이것은 현재 성장하고 있는 object 의 끝 부분이다. 만약 아무런 object 도 성장하고 있지 않다면 **obstack_next_free** 는 **obstack_base** 와 같은 값을 반환한다.

int **obstack_object_size** (struct obstack **obstack-ptr*) [Function]

이 함수는 현재 성장하고 있는 object 의 바이트로의 크기를 반환한다. 이 값은 아래와 같다.

obstack_next_free (*obstack_ptr*) obstack_base (*obstack_ptr*)

2.9 Obstack 들의 데이타 정렬(alignment)

각각의 obstack 는 *alignment boundary* 를 가지고 있다; obstack 내에 할당되어 있는 각각의 object 는 자동적으로, 지정된 경계의 배수인 주소에서 시작한다. 기본적으로 이 경계는 4 바이트이다.

obstack 의 alignment boundary 에 접근하기 위해서는 아래와 같은 함수 prototype 을 가질 수 있는 *obstack_alignment_mask* 매크로를 사용하라.

int **obstack_alignment_mask** (struct obstack **obstack-ptr*) [Macro]

이 값은 한 bit 의 mask 이다; 1 로 설정된 bit 는 object 의 주소에 대응하는 bit 가 0 이어야 함을 나타낸다. mask 의 값은 2 의 제곱보다 작은 값이어야 한다; 그 결과 모든 object 의 주소는 2 의 제곱의 배수이다. mask 의 기본값이 3 이면 주소는 4 의 배수가 된다. mask 의 값이 0 인 것은 object 가 1 의 배수인 어느 것에서도 시작될 수 있음을 의미한다. (즉, alignment 가 필요없다.)
obstack_alignment_mask 매크로의 확장 (expansion) 이 lvalue 이므로 당신은 mask 를 assignment 로 변경할 수 있다. 예를 들면, 다음 문장은:

```
obstack_alignment_mask (obstack_ptr) = 0;
```

지정된 obstack 내에서 처리중인 alignment 를 꺼버리는 효과를 가지고 있다.

alignment mask 에서 일어난 변화는 어떤 object 가 obstack 에서 할당 되거나 종료되기 *까지*는 아무런 효과도 없음을 알아라. 만약 당신이 어떤 object 를 성장시키고 있지 않다면, 당신은 **obstack_finish** 를 호출함으로써 새 alignment mask 가 즉각 효과를 갖도록 만들수 있다. 이것은 길이가 0 인 object 를 종료하고 그런 뒤 다음 object 을 위한 적절한 alignment 를 한다.

2.10 Obstack Chunk 들

obstack 들은 자체적으로 큰 chunk 들에 공간을 할당하고 당신의 요청에 부응하여 chunk 들에서 공간을 분배한다. Chunk 들은 당신이 다른 크기를 지정하지 않는 이상 4096 바이트의 크기이다. Chunk 크기는 object 들을 저장하는 데에 실제로 사용되지 않는 8 바이트의 overhead 를 포함한다. 이 overhead 에 대해 설명을 한다면 4 바이트는 현재 chunk 의 한계 즉 limit 를 저장하는데 사용되고, 다른 4 바이트는 현재 obstack 내에 chunk 가 여러개 일 경우 이전 chunk 를 가르키는데 사용된다. 설정된 크기에 관계없이, 큰 object 를 위해서 필요하다면 더 큰 chunk 들이 할당되어질 것이다.

obstack 라이브러리는 함수 **obstack_chunk_alloc** 을 호출함으로써 chunk 들을 할당하는데, 당신이 그것을 정의해야 한다. 당신이 어떤 chunk 에 있는 모든 object 들을 해제하여 그 chunk 가 더이상 필요치 않게 되면, obstack 라이브러리는 함수 **obstack_chunk_free** 를 호출하여 그 chunk 를 해제한다. 물론 이것도 당신이 반드시 정의해야 한다.

다음 두개의 정의은 반드시 **obstack_init** (2.1 절 [obstack 생성하기], 2 쪽 참조) 을 사용하는 각 소스 파일에서 (매크로로) 정의되거나 (함수로) 선언되어야만 한다. 이것들은 거의 대부분 아래와 같이 매크로로 정의된다:

```
#define obstack_chunk_alloc xmalloc
#define obstack_chunk_free free
```

이들은 단순히 매크로임을 주의하라. (인자가 없다.) 인자를 가지게 매크로를 정의한다면 작동하지 않을 것이다! 다만 **obstack_chunk_alloc** 나 **obstack_chunk_free** 는 그 자체가 함수 명칭이 아닐 경우에 함수 명칭으로 확장될 필요가 있다.

만약 당신이 **malloc** 로 chunk 들을 할당하면 chunk 크기는 2 의 제곱이여야 한다. 기본 chunk 크기는 4096 으로 정해졌는데, 이 크기로는 현재 작은 object 에서 요구되어질 수 있는 대부분의 전형적인 요청들을 만족시킬 수가 있으며, 아직 사용되지 않은 마지막 chunk 부분에 너무 많은 메모리가 낭비되지 않도록 할 만큼 작기 때문이다.

int **obstack_chunk_size** (struct obstack **obstack-ptr*) [Macro]

이것은 주어진 obstack 의 chunk 크기를 반환한다.

이 매크로는 lvalue 로 확장되기 때문에 당신은 그것에 새로운 값을 배당함으로써 새로운 chunk 크기를 지정할 수 있다. 그렇게 하게 되면 이미 할당되어있는 chunk 들에는 영향이 없고, 다만 나중에 할당되는 특정 obstack 을 위해 할당되는 chunk 들의 크기에만 영향을 주게 된다. Chunk 크기를 작게 하는 것은 별로 유용해 보이지 않지만 chunk 크기와 비슷한 많은 object 들을 할당한다면 크게 만드는 것이 효율성을 증가 시킬 수도 있다. 아래는 어떻게 하는지 분명히 보여준다:

```
if (obstack_chunk_size (obstack_ptr) < new-chunk-size)
  obstack_chunk_size (obstack_ptr) = new-chunk-size;
```

2.11 Obstack 함수들의 요약

다음은 obstack 과 연관되는 모든 함수의 요약이다. 각각은 그것의 첫번째 인자로 obstack **(struct obstack *)** 의 주소를 가진다.

void obstack_init (struct obstack **obstack-ptr*)

obstack 을 초기화. 2.1 절 [Obstack 생성하기], 2 쪽 참조.

void *obstack_alloc (struct obstack **obstack-ptr*, int *size*)

초기하되지 않은 *size* 바이트의 object 를 할당한다. 2.3 절 [Obstack 에서의 할당(allocation)], 5 쪽 참조.

void *obstack_copy (struct obstack **obstack-ptr*, void **address*, int *size*)

size 바이트의 object 를 할당하고, *address* 로부터 내용을 복사한다. 2.3 절 [Obstack 에서의 할당(allocation)], 5 쪽 참조.

void *obstack_copy0 (struct obstack **obstack-ptr*, void **address*, int *size*)

size+1 바이트의 object 를 할당하고, *address* 로부터 *size* 크기만큼의 내용을 복사하고 끝에 NULL 문자를 추가한다. 2.3 절 [Obstack 에서의 할당(allocation)], 5 쪽 참조.

void obstack_free (struct obstack **obstack-ptr*, void **object*)

> *object* (와 지정된 obstack 에서 *object* 보다 최근에 할당된 모든 것) 을 해제한다. 2.4 절 [Obstack 내의 object 들 해제(freeing)], 6 쪽 참조.

void obstack_blank (struct obstack **obstack-ptr*, int *size*)

> 성장하는 object 에 초기화되지 않은 *size* 바이트를 추가한다. 2.6 절 [성장하는 Object 들], 7 쪽 참조.

void obstack_grow (struct obstack **obstack-ptr*, void **address*, int *size*)

> 성장하는 object 에 *address* 에서 *size* 바이트를 추가한다. 2.6 절 [성장하는 Object 들], 7 쪽 참조.

void obstack_grow0 (struct obstack **obstack-ptr*, void **address*, int *size*)

> 성장하는 object 에 *address* 에서 *size* 바이트를 추가하고 그런 후 NULL 문자를 포함하는 다른 바이트를 추가한다. 2.6 절 [성장하는 Object 들], 7 쪽 참조.

void obstack_1grow (struct obstack **obstack-ptr*, char *data-char*)

> 성장하는 object 에 *data-char* 를 가지는 1 바이트를 추가한다. 2.6 절 [성장하는 Object 들], 7 쪽 참조.

void *obstack_finish (struct obstack **obstack-ptr*)

> 성장 중인 object 를 종결시키고, 최후 주소를 반환한다. 2.6 절 [성장하는 Object 들], 7 쪽 참조.

int obstack_object_size (struct obstack **obstack-ptr*)

> 현재 성장하는 object 의 현재 크기를 얻습니다. 2.6 절 [성장하는 Object 들], 7 쪽 참조.

void obstack_blank_fast (struct obstack **obstack-ptr*, int *size*)

> 방(room) 이 있는지에 대한 검사없이 성장하는 object 에 초기화되지 않은 *size* 바이트를 추가한다. 2.7 절 [특히 빠르게 성장하는 object 들], 8 쪽 참조.

void obstack_1grow_fast (struct obstack **obstack-ptr*, char *data-char*)

> 방(room) 이 있는지에 대한 검사없이 성장하는 object 에 *data-char* 를 함하는 1 바이트를 추가한다. 2.7 절 [특히 빠르게 성장하는 object 들], 8 쪽 참조.

int obstack_room (struct obstack **obstack-ptr*)

> 현재 object 가 성장하는데 있어서 이용 가능한 방(room) 의 양을 얻는다. 2.7 절 [특히 빠르게 성장하는 object 들], 8 쪽 참조.

int obstack_alignment_mask (struct obstack **obstack-ptr*)

> object 의 시작점을 alignment 하는데 사용하는 mask. 이것은 lvalue 이다. 2.9 절 [Obstack 들의 데이타 정렬(alignment)], 10 쪽 참조.

int obstack_chunk_size (struct obstack **obstack-ptr*)

> 할당한 chunk 들의 크기. 이것은 lvalue 이다. 2.10 절 [Obstak Chunk 들], 10 쪽 참조.

void *obstack_base (struct obstack **obstack-ptr*)

> 현재 성장하는 object 의 임시 시작 주소. 2.8 절 [Obstack 의 상태], 9 쪽 참조.

void *obstack_next_free (struct obstack **obstack-ptr*)

> 현재의 성장하는 object 가 끝나는 바로 다음의 주소. 2.8 절 [Obstack 의 상태], 9 쪽 참조.

2.12 Obstack 함수 혹은 매크로들의 내부

위 부분까지는 사용자가 Obstack 을 자신의 프로그램상에서 어떻게 가져와서 사용할 지에 대해서 보았다. 또한 위 부분에서는 Obstack 에서 제공하는 함수들이 실제로 함수의 모양을 가지고 있는 것도 있고 아니면 달리 매크로로 제공하는 것도 있다고 하였다.

이제 그 함수들이 어떻게 구성되어 있는지에 대해서 언급하겠다.

Obstack 이라는 것이 개념적으로 복작한 프로그램이 아니라 확실히 spec 이 정해져 있는 것 즉, stack 이라는 점에서 각각의 obstack 함수 혹은 매크로들이 의존하는 몇 개의 함수들만 살펴보면 나머지들은 모두 그 함수들의 다른 함수들과 묶어서 사용하는 것이나, 일반적인 obstack 포인터를 연산하는 내용들로 구성되어 있는 것이 대부분이다.

의존하는 함수는 아래와 같은 것이 존재한다.

- obstack_begin
- obstack_newchunk
- obstack_free

하지만 실제로 $prefix/gcc/libiberty/obstack.c 파일을 보면 알겠지만 함수 앞에 '_' 가 붙어서 같은 이름의 함수가 존재하는 걸 볼 수 있는데, 이렇게 한 이유는 ANSI code 혹은 non-ANSI code 로 구분하기 위해서이다.

위에서 언급한 각 함수에 대해서 설명을 하겠다.

- obstack_begin (struct obstach *h, int size, int alignment, POINTER (*chunkfun) (), void (*freefun) ())

  사용할 obstack H 를 초기화한다. chunk size 를 SIZE 로 지정한다. (0 값은 기본값을 의미) Object 들은 ALIGNMENT 의 배수들상에서 시작한다. (0 값은 기본값을 의미) CHUNKFUN 은 chunk 들을 할당하는데 사용하는 함수이고, FREEFUN 은 그것들을 해제하는 함수이다.

  만약 성공시 0 이 아닌값을, 메모리 고갈시 0 을 반환한다. 메모리 고갈 오류에서 복구할려면 몇몇 메모리를 해제하고 다시 이것을 호출하라.

  이 함수와 이름이 유사한 함수 _obstack_begin_1 가 존재하는데, 이 함수는 struct obstach *h 의 구성요소 extra_arg 를 사용할 경우 이를 사용하게 된다. 또한 이것을 사용한다는 것을 표현해줄 use_extra_arg 를 1 로 설정해 주게된다.

  그림 2 에 이 함수를 사용하여 처음 obstack 을 초기화한 후 구성이 어떻게 되는가에 대한 모습이 나와 있다.

- obstack_newchunk (struct obstack *h, int length)

  obstack *H 에 새로운 현재 chunk 를 할당하는데 다음과 같이 LENGTH 바이트가 현재 object 에 추가되는게 필요하거나 길이 LENGTH의 새로운 object 가 할당되었다고 가정한다. 옛날 chunk 의 끝에서 새로운 것의 처음로 어떤 특정 object 를 복사한다.

  그림 3 에서는 새로운 chunk 를 할당받고 그 크기가 어떻게 정해지는지와 다시 어떻게 연결되는지에 대해서 표현해 보았다.

- obstack_free (struct obstack *h, POINTER obj)

  obstack H 에서 object 를 해제하는데 OBJ 보다 더 최근에 할당된 모든 것와 OBJ 를 포함한다. 만약 OBJ 가 0 이면, H 내의 모든 것을 해제한다.

위 함수들이 모든 Obstack 함수 혹은 매크로들의 구조에 기반이 되는 함수이다. stack 에서 필요한 "초기화", "추가", "해제" 로 구성되어 있는 것이다.

이제 실제 Obstack 함수들이 어떻게 되어 있는지를 보고 끝마치도록 하겠다.

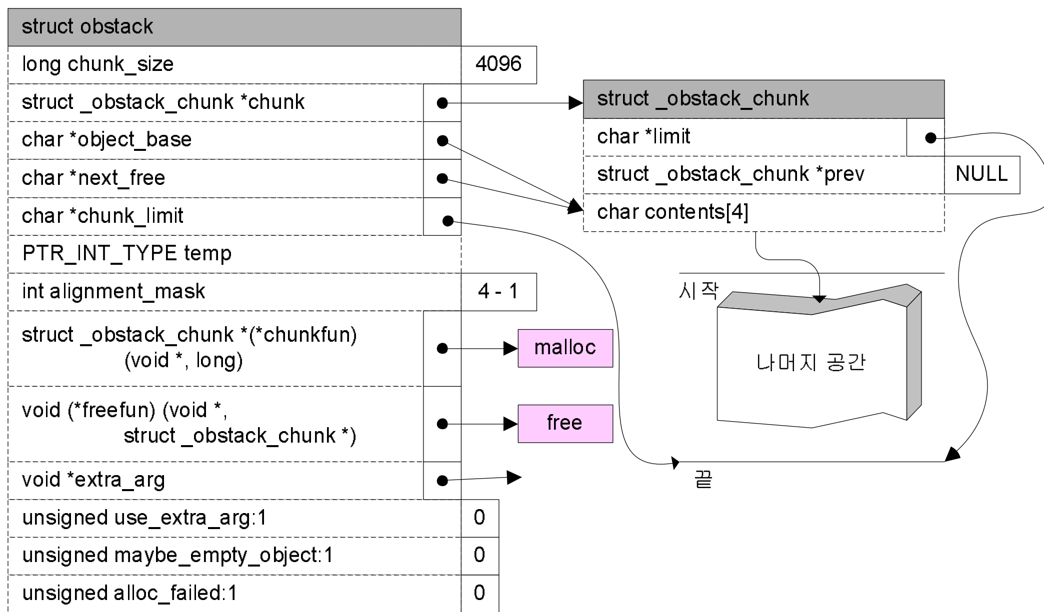

그림 2: obstack_begin 을 수행한 후 obstack 의 상태

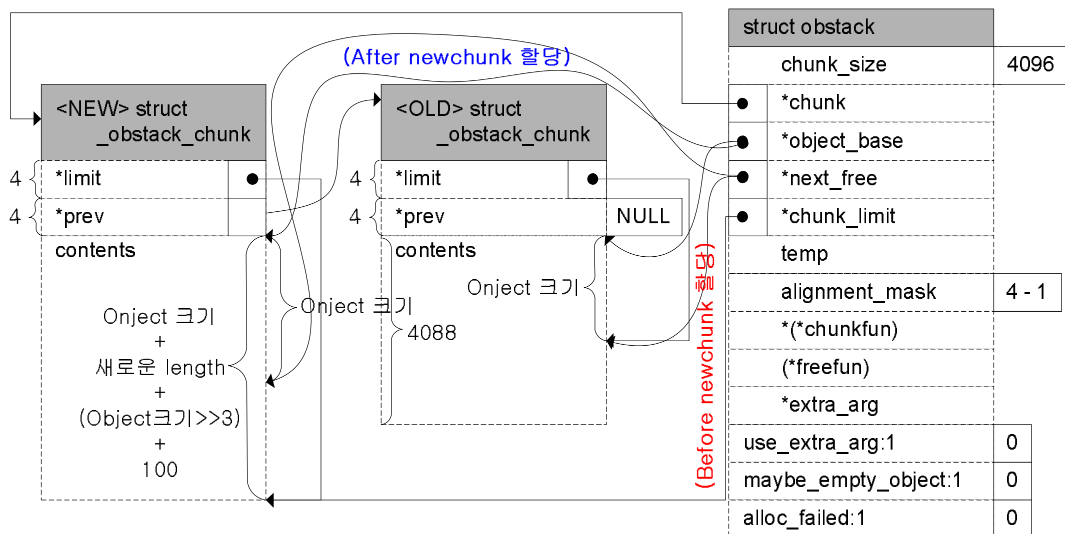

그림 3: obstack_newchunk 로 새로운 chunk 를 할당받은 후의 모습

```
# define obstack_blank(OBSTACK,length)                \
__extension__                                         \
({ struct obstack *__o = (OBSTACK);                   \
   int __len = (length);                              \
   if (__o->chunk_limit - __o->next_free < __len)     \
     _obstack_newchunk (__o, __len);                  \
   __o->next_free += __len;                           \
   (void) 0; })

# define obstack_finish(OBSTACK)                      \
__extension__                                         \
({ struct obstack *__o1 = (OBSTACK);                  \
   void *value;                                       \
   value = (void *) __o1->object_base;                \
   if (__o1->next_free == value)                      \
     __o1->maybe_empty_object = 1;                    \
   __o1->next_free                                    \
     = __INT_TO_PTR ((__PTR_TO_INT                    \
             (__o1->next_free)+__o1->alignment_mask)\
             & ~ (__o1->alignment_mask));             \
   if (__o1->next_free - (char *)__o1->chunk          \
       > __o1->chunk_limit - (char *)__o1->chunk)     \
     __o1->next_free = __o1->chunk_limit;             \
   __o1->object_base = __o1->next_free;               \
   value; })

# define obstack_init(h) \
  _obstack_begin ((h), 0, 0, \
                  (void *(*) (long)) obstack_chunk_alloc,
                  (void (*) (void *)) obstack_chunk_free)

# define obstack_alloc(OBSTACK,length)                \
__extension__                                         \
({ struct obstack *__h = (OBSTACK);                   \
   obstack_blank (__h, (length));                     \
   obstack_finish (__h); })

# define obstack_grow(OBSTACK,where,length)               \
__extension__                                             \
({ struct obstack *__o = (OBSTACK);                       \
   int __len = (length);                                  \
   if (__o->next_free + __len > __o->chunk_limit)         \
     _obstack_newchunk (__o, __len);                      \
   _obstack_memcpy (__o->next_free, (where), __len);      \
   __o->next_free += __len;                               \
   (void) 0; })
```

위를 보면 알겠지만 모든 것이 연산과 위에서 언급한 3 가지 함수에서 벗어나지 않는다. _obstack_memcpy 와 같은 함수로 혼동하실 분들이 계실 수 있는데, 이것은 단순한 memcpy 함수라서 따로 언급할 필요가 없겠다.

제 3 절 12 주 강의를 마치며

설마 했는데, 결국 번역판+α 가 되어 버렸습니다. :'(이 문서의 출처는 *Phil Edwards 외 여러명* 이 작성한 ***GNU libiberty*** 임을 밝히는 바입니다. 이번 12 주 강의는 이것으로 마치겠습니다.

참고 문헌

[1] Phil Edwards et al: *GNU libiberty* (2001)

[2] 직장인을 위한 씨모임: *GNU 라이브러리 참조 안내서*
http://database.sarang.net/study/c/gcc_korean/01.htm (unknown)