22. 메모리(memory)

- 이 장에서는 메모리에 대해서 포괄적인 주제를 다룹니다.

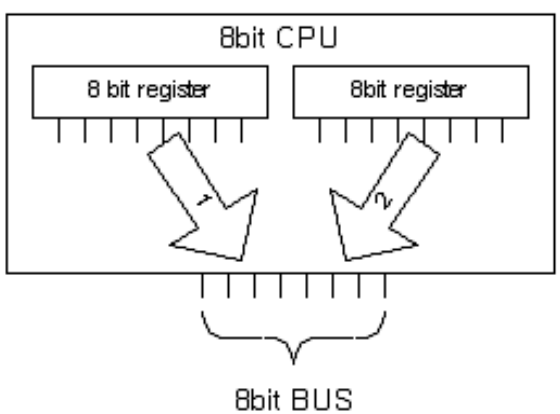
① delete 와 delete[]의 차이점은?
② 왜 Borland C++ 3.1에서는 주소가 xxxx:xxxx 형태로 찍히는가?
③ 가상 메모리(virtual memory), 스왑 파일(swap file)
④ 선형 주소(linear address), 세그멘테이션(segmentation)
⑤ 64K의 한계, 4Giga의 한계
⑥ 단편화(fragmentation), 컴팩션(compaction)

옛날 옛적 8bit와 16bit 시절

- 1983년 처음 개인용 컴퓨터를 접한 것은 SPC-1000이라고 불리는 8비트 컴퓨터였습니다.
- 8bit CPU를 사용했던 PC가 어떻게 주소를 64K까지 지원할 수 있었을까요?
- 16bit 인텔8088을 사용했던, PC가 또한 어떻게 주소를 1Mega까지 지원할 수 있었을까요?

세그멘테이션(segmentation)

- 8비트 Apple은 주소 지정을 위해 8비트 레지스터를 2개 사용했습니다.
- 비록 내부적인 데이터의 처리 및 흐름은 8비트였지만, 주소 지정을 위해 16비트를 사용했던 것입니다.



8비트 CPU에서 16비트 주소의 사용: 16비트 주소를 생성하기 위해, 2개의 8비트 주소 레지스터를 사용했습니다. 8비트 Apple 시절 최대 주소는 2^{16}=64K였습니다. 외부와 연결된 8비트 버스를 통해 8비트의 데이터가 2번 전송되었습니다.

인텔의 선택: 세그멘테이션(segmentation)

- 16bit CPU를 사용한 인텔은, 메모리의 크기를 두고 고민했습니다.
- 비록 Apple의 CPU가 사용한 아키텍쳐(architecture)를 사용할 수 있었음에도 불구하고, 그것은 4Giga(약 4×10^9)라는 거대한 주소를 지원했고, 이러한 크기의 주소는 PC에서는 필요 없다고 판단했을 것입니다.
- 이 책의 초판을 집필할 당시 1999년 3월경 PC는 Pentium II 350, 64Mega에 하드 디스크의 용량은 8Giga 였습니다.
- 64K를 지원하는 PC가 나온지, 약 20여년 만에 일반적인 메인 메모리의 요구량은 ×1000배, 하드 디스크는 약 ×100,000배 증가하였습니다.
- 이제 또 17년 정도가 흐른 지금, 당시 하드디스크보다 두 배되는 용량의 메모리를 사용하고 있으니 기술의 발달을 함부로 예측해서는 안 되겠다고 생각해 봅니다.

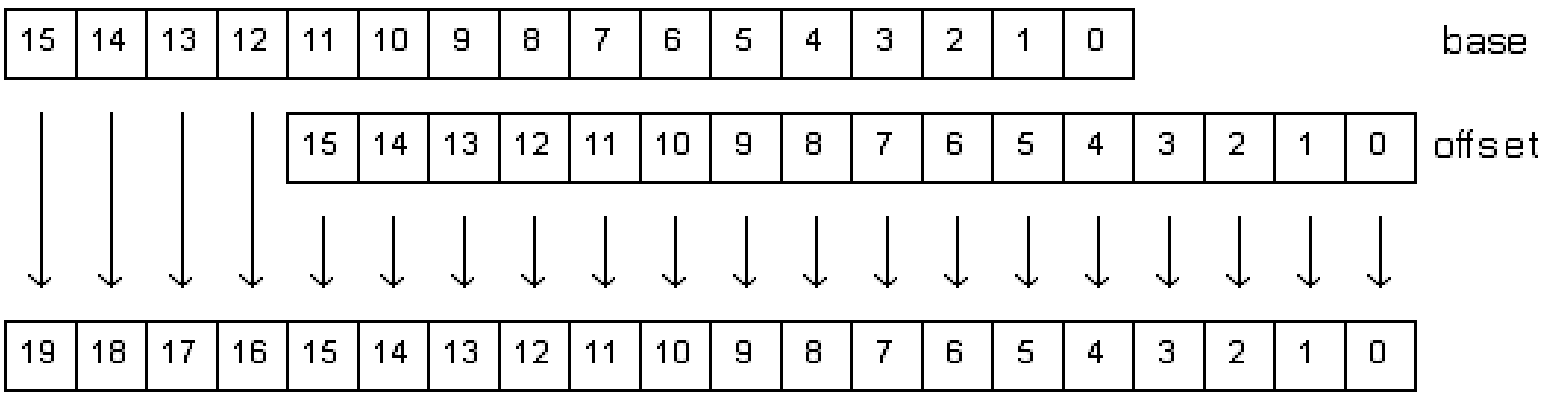
• 인텔은 이미 알려진 **세그멘테이션segmentation**이라는 주소 지정 기법(addressing mode)을 자사의 i8086에 도입했습니다.
• 메모리를 **세그먼트segment**라고 불리는 몇 개의 블록block으로 나눈 다음, 주소를 지정하기 위해, 블록의 처음에서 상대주소를 사용하는 방식이었습니다.

블록 번호(block number)+블록의 처음에서 상대 주소(offset)

• 인텔은 16개의 세그먼트 블록을 사용했고, 상대 주소 지정을 위해 16비트를 사용했습니다.
• 그러므로, 1개 블록의 크기는 2^{16}=64K였습니다.
• 이러한 64K 크기의 블록을 세그먼트라고 합니다.
• 이러한 세그먼트가 모두 16개있으므로, 인텔 i8086이 지원하는 주소 공간의 크기는 다음과 같습니다.

$$2^4 \times 2^{16} = 2^{20} = 1\text{Mega}$$

• 어떻게 20비트의 주소를 생성할 수 있을까요? 이것은 16비트 레지스터를 2개 겹치게 배열함으로써 가능했습니다.



세그멘테이션의 주소 지정(i8086의 경우): 20비트의 주소는 16비트 레지스터에 의해 2곳에 나뉘어 저장되었습니다. 실제로 메모리의 한 곳을 지정하는 주소는 첫 번째 레지스터인 베이스(base) 레지스터가 세그먼트 블록의 시작을 지정하면, 두 번째 레지스터인 오프셋(offset) 레지스터가 세그먼트 블록의 처음에서 오프셋을 지정했습니다.

• 세그먼트 블록의 시작을 가리키는 16비트 레지스터를 **베이스 레지스터(base register)**라고 하고, 블록의 처음에서 상대 주소를 가리키는 16비트 레지스터를 **오프셋 레지스터(offset register)**라고 합시다.

• 베이스의 하위 12비트와 오프셋의 상위 12비트는 겹치게 배열되었습니다. 실제적으로, 겹치는 12비트의 값은 수학적으로 더한(add) 결과가 되었습니다.

• 예를 들면, 베이스 값이 다음과 같다고 가정합니다.

0100 0000 0011 0111

• 그리고, 오프셋의 값이 다음과 같다고 가정합니다.

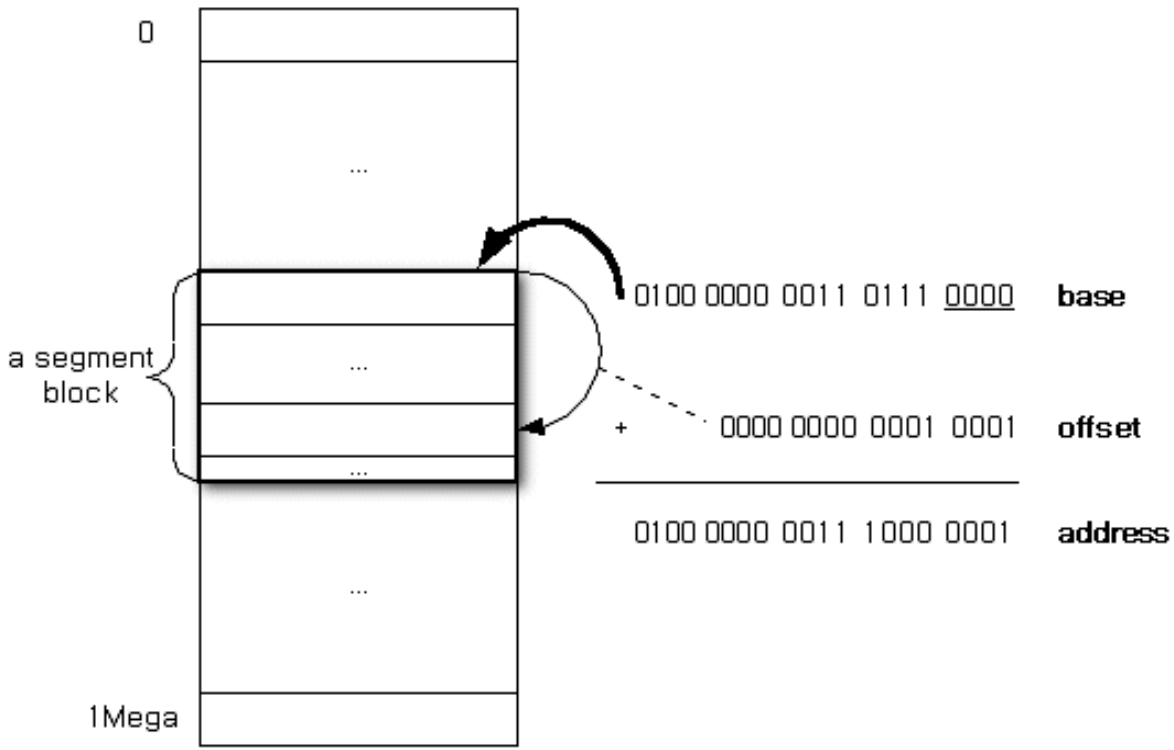
0000 0000 0001 0001

• 이것은 1Mega 메모리에서 마지막에 4bit를 추가한 20bit의 주소로 해석되어, 세그먼트 블록이 아래의 값에서 시작함을 의미합니다.

0100 0000 0011 0111 0000

• 그리고, 이 세그먼트 블록에서 아래의 오프셋 값만큼 떨어진 메모리 셀cell을 의미하였습니다.

0000 0000 0001 0001



20비트의 계산: 20비트 주소를 계산하기 위해, base의 하위 4비트가 0으로 채워진 20비트 세그먼트 블록의 시작 주소와, 16비트 offset의 값을 더합니다. 실제의 주소는 0100 0000 0011 0111:0000 0000 0001 0001(base:offset)임에도 불구하고, 프로그램은 20비트 값인 0100 0000 0011 1000 0001이라고 생각합니다.

• 하위 4비트가 0으로 채워진 20비트의 base값은 1Mega 메모리 상에서 세그먼트 블록의 시작을 가리킵니다.
• 실제로 20비트의 주소는 존재하지 않는다는 것을 주의하세요.
• 하위 4비트는 항상 0이므로, 실제 20비트가 가리키는 값은 상위 16비트만이 사용됩니다.
• 즉, 2^4=16Byte으로 나뉘어진 작은 블록(small block)의 시작만을 가리킬 수 있습니다. 이 때 0에서 시작하는 2^4단위의 작은 블록을 **패러그래프(paragraph)**라고 했습니다.
• 그래서 베이스 값은 패러그래프의 경계만을 가리킬 수 있는 것입니다.
• 이렇게 64K 블록의 시작이 정해지면, 16비트 오프셋 레지스터가 블록의 시작에서 상대 주소(relative address)를 가리킵니다.
• 프로그램은 이러한 2개의 16비트 주소 레지스터가 더해진 20비트 주소를 메모리 주소로 간주하는 것입니다.

64K의 한계

• 세그멘테이션을 사용하는 컴퓨터에서, 배열의 전체 크기가 64K를 넘어가면 어떠한 문제가 발생할까요?

• 배열의 시작 주소에서 64K의 경계를 넘는 순간, 배열의 시작 주소는 바뀌어야 합니다.

• 이 시절, 이러한 구현이 불가능한 것은 아님에도 불구하고, 볼런드(borland) 등 대부분의 컴파일러 벤더(vendor)들은 배열의 크기를 64K로 제한했습니다.

• 이러한 제한을 **64K의 한계**라고 합니다.

```
#include <stdio.h>

char a[65536]={0,};//64K의 한계를 무시한 에러!

void main() {
    printf("%c\n",a[0]);
}
```

• 에러 메시지는 다음과 같이 출력되었습니다.

"Array size too large"

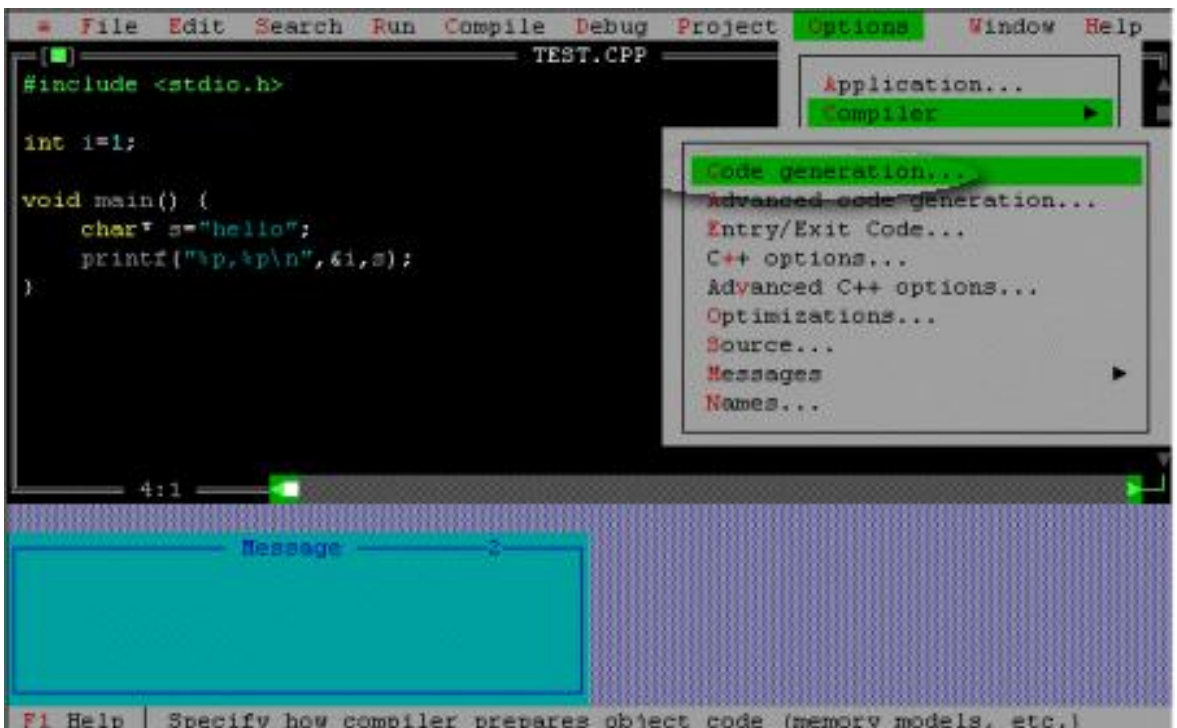
메모리 모델(memory model)

- 코드 및 데이터는 몇 개의 세그먼트 블록을 사용할까요?
- 코드와 데이터 세그먼트가 각각 몇 개의 세그먼트 블록을 사용하는가에 따라서, 포인터를 표현하는 방식은 달라져야 합니다.
- 만약 데이터가 오로지 한곳의 세그먼트에 저장된다면, 데이터를 가리키는 포인터 중 베이스 값은 고정되어 있으므로, 프로그램이 종료할 때까지 바뀌지 않습니다.
- 하지만, 데이터가 2곳의 세그먼트에 저장된다면, 실행 중에, 베이스 값도 바뀌어야 하므로, 포인터는 32비트로 표현되어야 할 것입니다.

- 이처럼 코드와 데이터가 차지하는 세그먼트의 수에 따라, 메모리 모델이 나뉘었습니다.

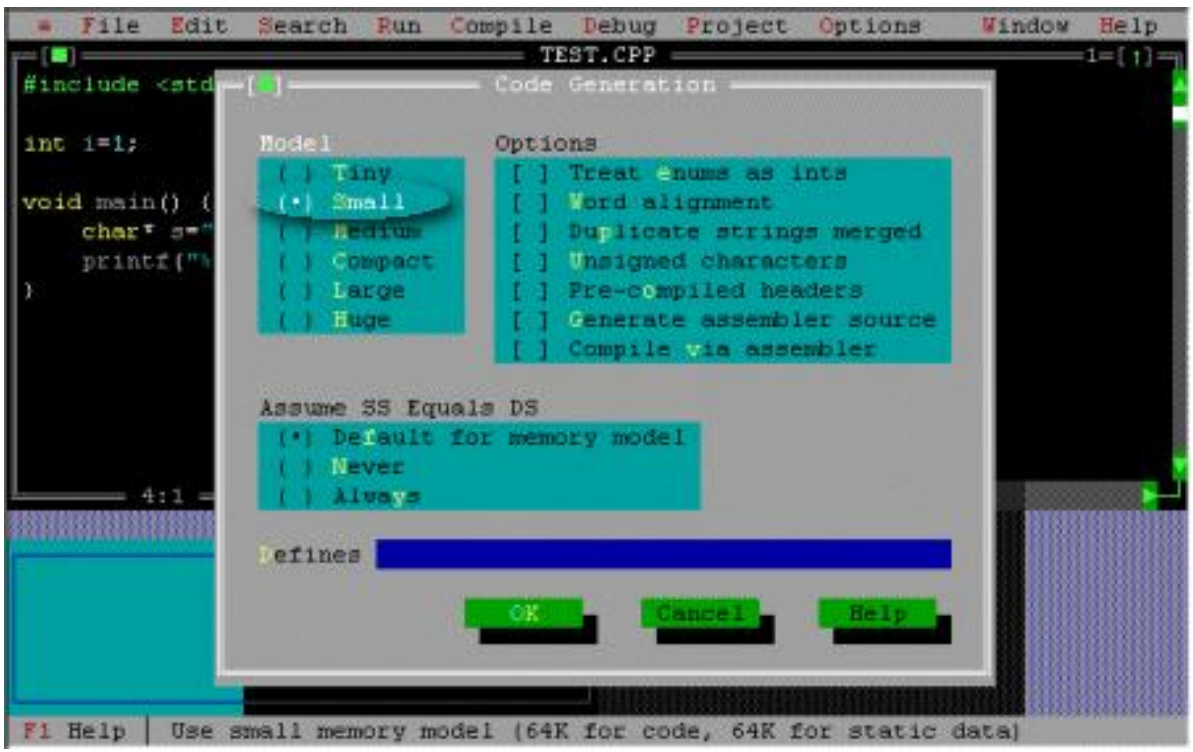
```
#include <stdio.h>

int i=1;

void main() {
    char* s="hello";
    printf("%p,%p\n",&i,s);
}
```

볼런드 C++ 3.1의 경우



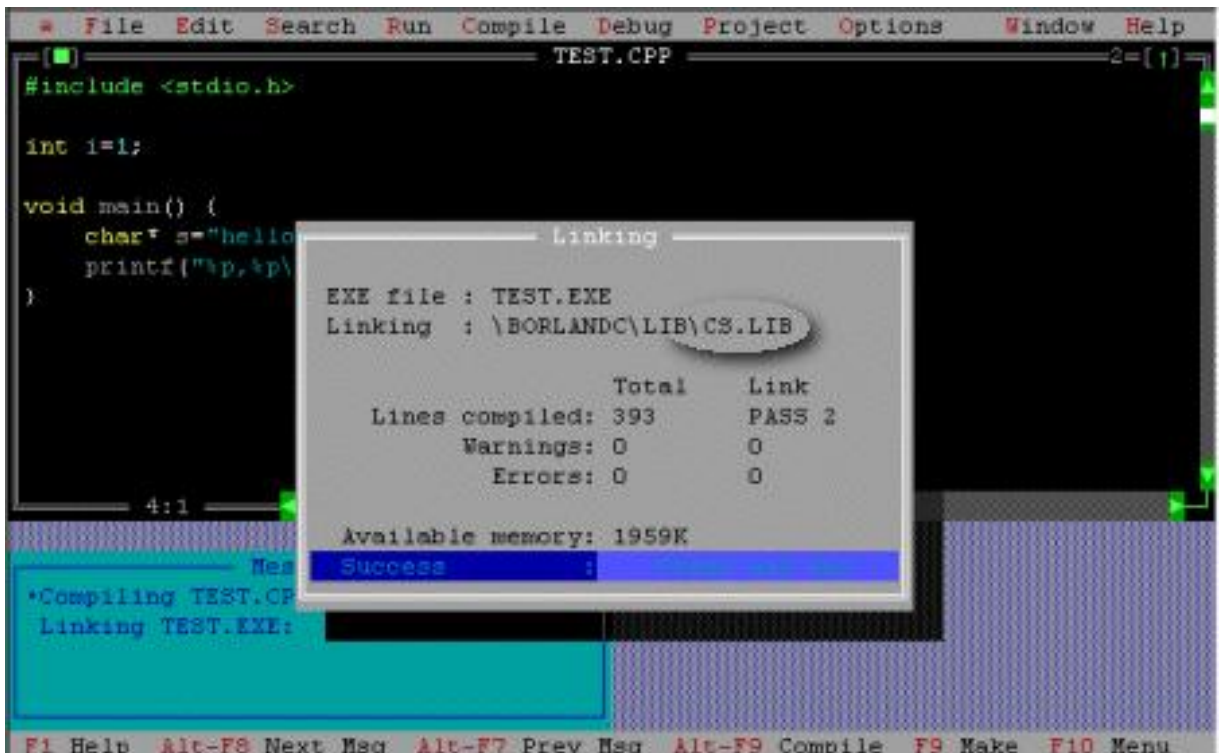
Code generation...: 여기서에 메모리 모델 및 최적화 방식 등을 지정할 수 있습니다.

- 대화상자에서 메모리 모델로 Small을 선택합니다.



Small 모델: 이 모델은 코드와 데이터 각각 1개의 세그먼트를 차지합니다.

- Compiler->Build All을 선택하여 실행 파일을 만들어 봅시다.



CS.LIB 라이브러리의 선택: Small 모델로 컴파일하면 CS.LIB가 선택되어 실행 파일이 만들어집니다.

• 실행 결과는 다음과 같습니다.

00AA,00Ac

• 코드와 데이터 각각이 1개의 세그먼트만을 사용하므로, 주소를 출력하면 베이스 값은 출력되지 않고, 오프셋 값만 출력됩니다.
• 이러한 16비트 포인터를 near 포인터라 하였습니다. 즉, near 포인터는 베이스 값이 고정된(fixed) 포인터입니다.

- 이제 메모리 모델을 Large로 바꾸어 보겠습니다.
- Large 모델은 코드와 데이터 모두 여러 개의 세그먼트를 사용하는 모델입니다. 그러므로, 포인터는 20비트를 표현하기 위해, 32비트로 유지해야 합니다.

231F:0094, 231F:0096

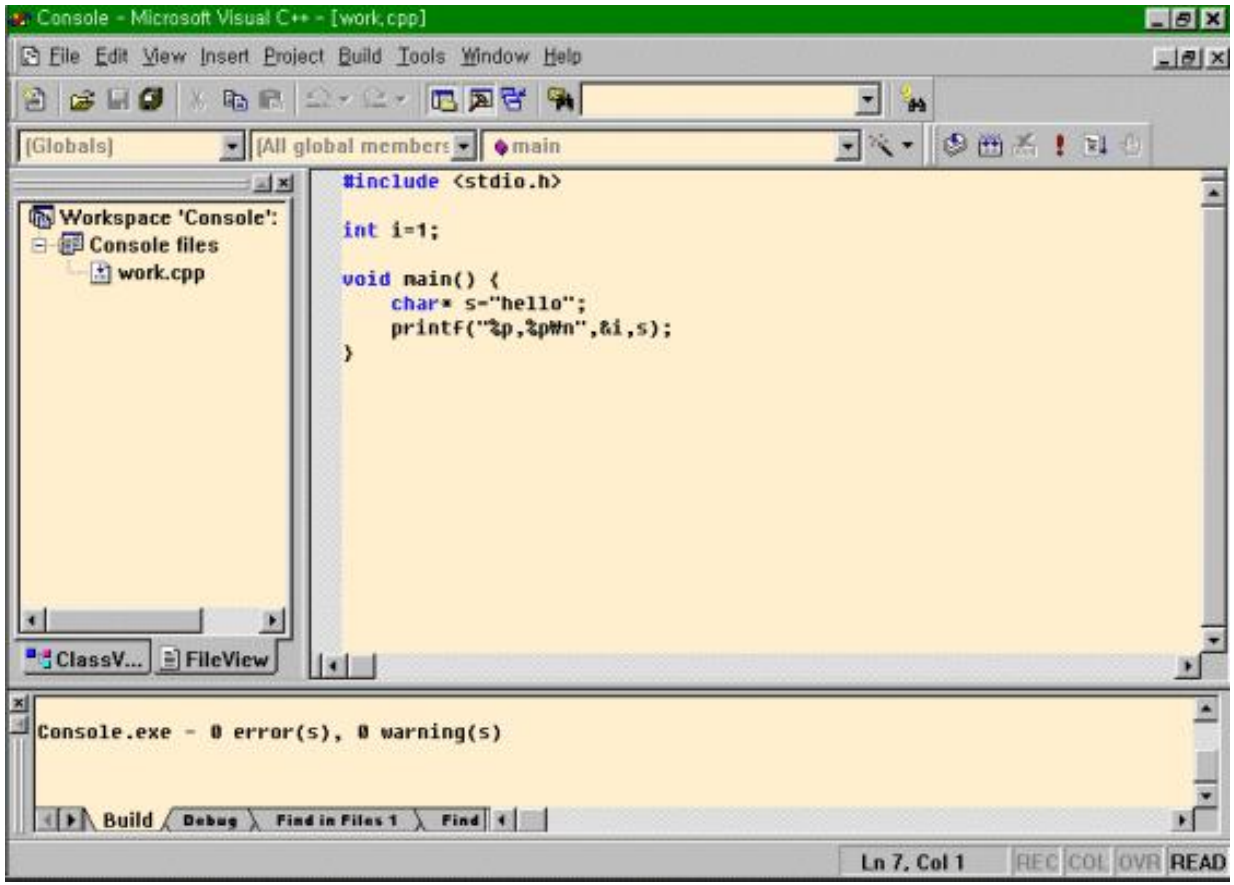
- Large 모델에서 포인터는 '베이스:오프셋' 형태로 표현된 것을 알 수 있습니다.
- 2017년도인 지금 보면, 황당하게 보이지만, 당시 C 프로그램을 하던 개발자들에게는 당연하게 이해해야 하는 부분이었습니다.

선형 주소(linear address)

- Windows™같은 32비트 운영체제에서는 주소를 어떻게 표현할까요?
- Windows에서는 주소를 나타내기 위해, 더 이상 세그멘테이션을 사용하지 않습니다. 주소는 선형적인 32비트로 표현됩니다.
- 이러한 주소를 **선형 주소(linear address)**라고 합니다.

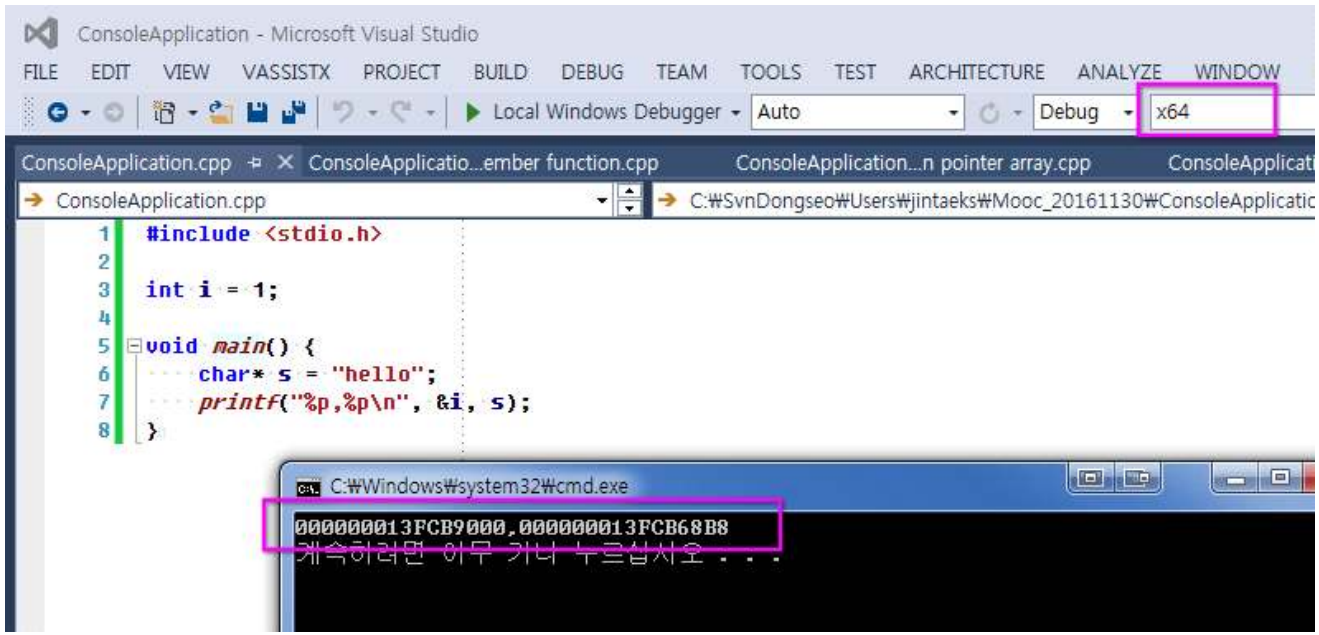
- 위의 프로그램을 Win98의 Visual C++ 6.0에서 컴파일하면, 결과는 다음과 같습니다.

00412A30,00412A3C



Visual C++ 6.0: 이 컴파일러는 32비트 운영체제인 Windows에서 동작하는 32비트 전용 컴파일러입니다. 그러므로, 더 이상 64K의 한계는 존재하지 않습니다.

- 같은 코드를 Visual Studio 2013에서 x64로 빌드해서 실행해 보겠습니다.



Visual Studio 2013 x64 모드: x64모드에서 주소는 64bit로 표현됩니다.

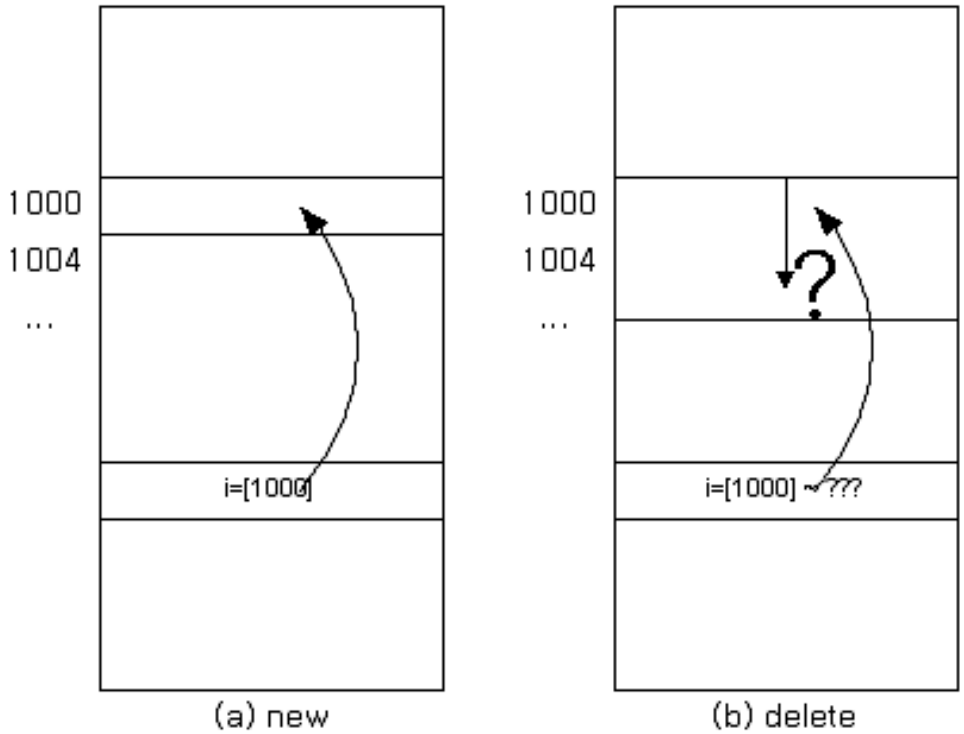
- 결과는 다음과 같습니다.

000000013FF09000,000000013FF068B8

동적 할당, 그 내부(internal)

- 간단하게 보이는 아래의 프로그램에서 이상한 부분을 찾아봅시다.
- 문법 에러(syntax error) 혹은 논리적 에러(semantic error)는 존재하지 않습니다.

```
#include <stdio.h>

void main() {
	int* i;
	i=new int;//i=(int*)malloc(sizeof(int));와 같다.
	*i=3;
	printf("%d\n",*i);
	delete i;//free(i);와 같다.
}
```



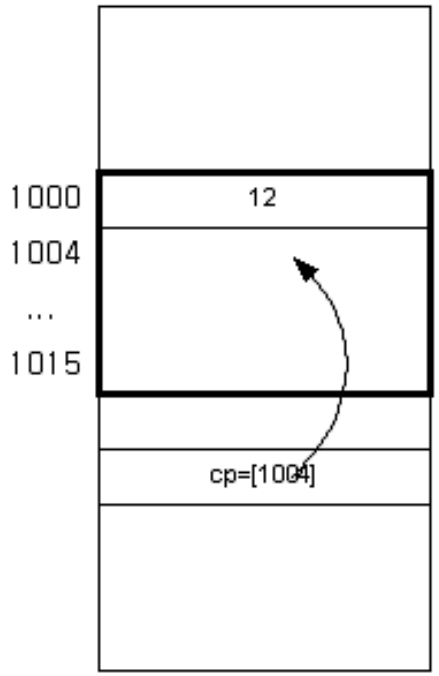
이상한 delete의 동작: (a) new에 의해 4바이트가 할당되고, 이곳을 i가 가리킨다고 합시다. new가 할당한 곳은 [1000]입니다. (b) delete는 i가 가리키는 곳을 해제합니다. i에는 오로지 [1000]밖에 없습니다. 어떻게 i가 가리키는 곳이 [1000]~[1003]까지의 4바이트인지 알 수 있을까요?

- new int; 에 명시한 int에 의해 new는 정확하게 4바이트를 할당할 것입니다.
- 이곳이 [1000]번지에서 [1003]번지까지의 4바이트라고 합시다.
- main()의 마지막 문장은 i가 가리키는 [1000]에서 [1003]까지의 4바이트를 해제할 것입니다.
- 어떻게 이것이 가능한가요? i에는 단지 [1000]만 있을 뿐, 4바이트가 할당되었다는 정보도, 할당된 범위가 [1000]에서 [1003]까지란 정보도 없습니다.
- new는 이렇게 동작합니다.

① new는 사용자가 지정한 메모리의 크기+4바이트를 할당합니다
② 그리고, 할당한 위치+4를 리턴합니다.

- delete는 이렇게 동작합니다.

① delete는 지정된 포인터-4에 접근하여 할당된 메모리의 크기를 얻습니다.
② 그리고 (4 + '지정된 포인터-4'에 지정된 크기) 만큼의 메모리를 해제합니다.



new와 delete의 실제 동작: 볼런드 C++의 경우 동적 메모리 할당은 할당한 메모리의 크기를 유지하기 위해 여분의 4바이트를 더 할당합니다. 이러한 여분의 할당은 컴파일러마다 다를 수 있지만, 여분의 할당은 반드시 존재합니다. 사용자가 요구한 메모리의 크기가 12바이트라면 컴파일러는 16(12+4)바이트를 할당합니다. 그리고, '할당한 위치+4'(1004)를 리턴합니다.

• 반드시 짚고 넘어가야 할 주의 사항이 있습니다.
• 우리는 동적 메모리 할당이 항상 4바이트가 추가로 할당될 것이라고 가정(볼런드 C++에서조차)해서는 안 됩니다.
• 위의 소스에서 아래의 문장을 고려해 봅시다.

```
cp=new char[12];
```

• 위 문장을 아래와 같이 수정하면 어떻게 될까요?

```
cp=new char[13];
```

• 놀랍게도 결과는 다음과 같습니다.

474832
474800
474832

최소 블록(minimum block): 패러그래프(paragraph)

• 볼런드 C++의 도스용 버전의 이러한 동작은 세그멘테이션에 기인합니다. 그것은 다음과 같은 이유 때문입니다.

"**사용자가 동적 할당을 위해 지정한 바이트는, 항상 16의 배수로 할당합니다.**"

• 즉, 13을 지정하면, 17바이트를 할당해야하는데, 17을 포함하는 최소 16의 배수는 32이므로, 32바이트를 할당하는 것입니다.
• 세그멘테이션 주소 지정 방식의 주소 표현 방식 때문입니다.
• 메모리 블록의 베이스 주소(base address)의 하위 4바이트는 사용할 수 없으므로, 항상 메모리의 16바이트의 경계에서 동적 할당이 이루어지는 것입니다.

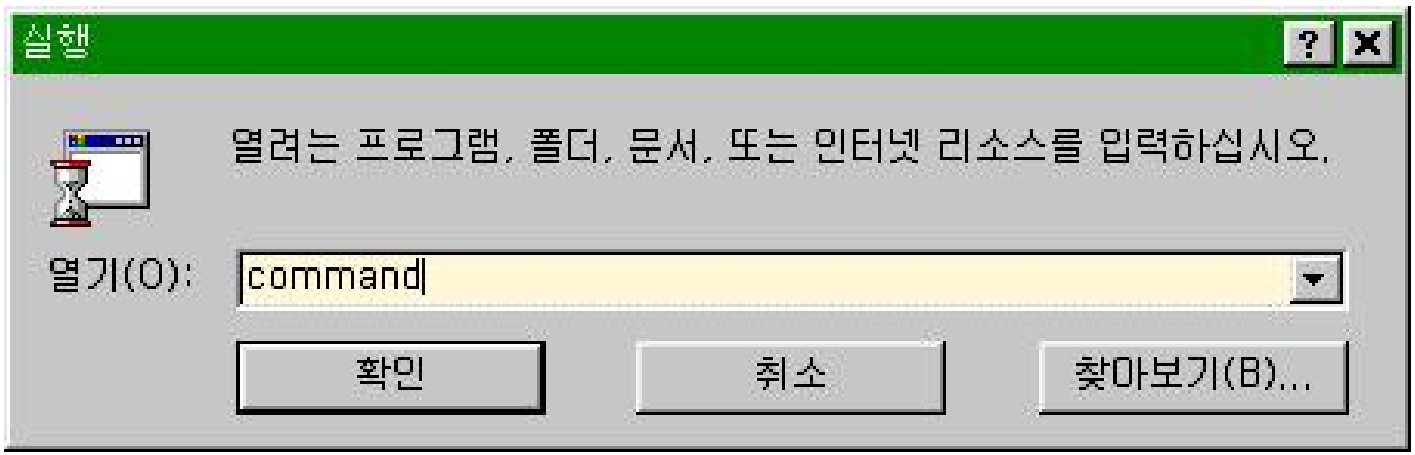
이러한 16바이트의 경계를 '**패러그래프 경계**'라고 합니다. 하지만, 32비트 운영체제에서 더 이상 패러그래프 경계는 존재하지 않습니다. 그렇지만, 효율성을 위해서 메모리 할당은 정해진 최소 단위로 이루어 집니다.

단편화(fragmentation)

• 위의 예에서 사용자가 메모리 블록의 크기로 13을 지정한 경우, 실제로 할당한 메모리는 32바이트이지만, 이 중 17바이트만이 실제로 사용되고, 나머지, 15바이트는 사용하지도 못하게 됩니다.

• 이러한 현상을 **메모리 내부 단편화(memory internal fragmentation)**라고 합니다. 최소 블록의 크기가 클수록 내부 단편화로 사용하지 못하는 메모리의 크기는 늘어 납니다.

• 디스크 블록(disk block)을 할당할 때에도 내부 단편화는 존재합니다. Win+R을 눌러 실행 대화상자(run dialog box)를 실행해 봅시다.



Windows의 실행 대화상자: '열기'에 command 혹은 cmd를 입력하고 확인을 선택하면 도스창(DOS prompt)이 실행됩니다.

• 도스창에서 다음과 같이 명령어를 입력합니다.

CD ₩

• C: 드라이브의 루트(root)로 이동한 다음 다음과 같이 명령어를 입력합니다.

DIR

• 이 명령은 루트 폴더에 존재하는 파일을 보여 줍니다. 아래 그림은 한 실행 결과입니다. 사용 가능한 디스크 공간이 1,434,173,440바이트 인 것을 확인할 수 있습니다.

```
COMMAND  COM        116,836  06-26-98  8:01p COMMAND.COM
DULOSBIB       <DIR>         03-12-99  4:08p DULOSBIB
FRUNLOG  TXT          1,091  03-15-99 11:35a FRUNLOG.TXT
WINDOWS        <DIR>         03-12-99 11:38a WINDOWS
NETLOG   TXT         41,657  03-15-99 11:48a NETLOG.TXT
AUTOEXEC BAT            305  03-17-99  8:38p AUTOEXEC.BAT
SCANDISK LOG          8,828  03-17-99  5:06p SCANDISK.LOG
PROGRA~1       <DIR>         03-12-99 11:42a Program Files
MYDOCU~1       <DIR>         03-12-99 12:08p My Documents
HPRD6          <DIR>         03-12-99  2:09p HPRD6
UTIL           <DIR>         03-12-99  2:20p UTIL
AUTOEXEC SYD            140  03-12-99  3:05p AUTOEXEC.SYD
TEMP           <DIR>         03-12-99  2:25p temp
TMP            <DIR>         03-12-99  2:25p tmp
FORWIN95       <DIR>         03-12-99  2:31p ForWin95
BAT            <DIR>         03-12-99  4:08p Bat
ABORT    LOG              0  03-12-99  8:14p abort.log
WEBSHARE       <DIR>         03-12-99  8:54p WEBSHARE
MMP└╫~1  LNK            271  03-14-99  5:21p Mmp└╫ ╣┘╓╫ ▒i▒Γ.lnk
CONFIG   SYS            132  03-17-99  8:38p CONFIG.SYS
          9 file(s)        169,260 bytes
         11 dir(s)   1,434,173,440 bytes free

C:\>
```

DIR의 결과: 현재 사용 가능한 디스크 공간은 1,434,173,449바이트입니다.

• 이제 COPY 명령을 사용하여 5바이트 크기의 텍스트 파일을 만들어 보겠습니다.

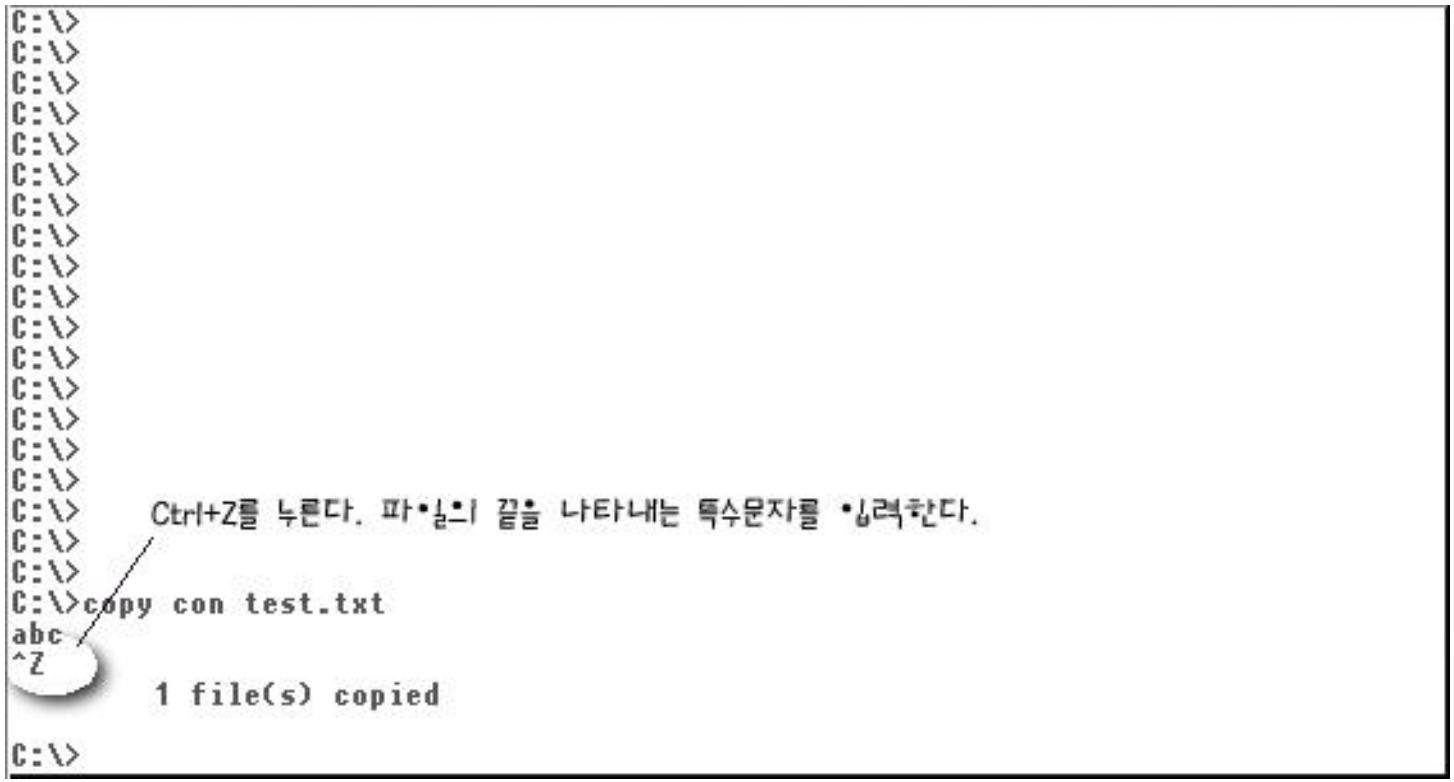
COPY CON TEST.TXT

• 위의 명령을 입력한 다음에, 연속된 줄에 파일의 내용을 아래와 같이 입력합니다.

abc
^Z

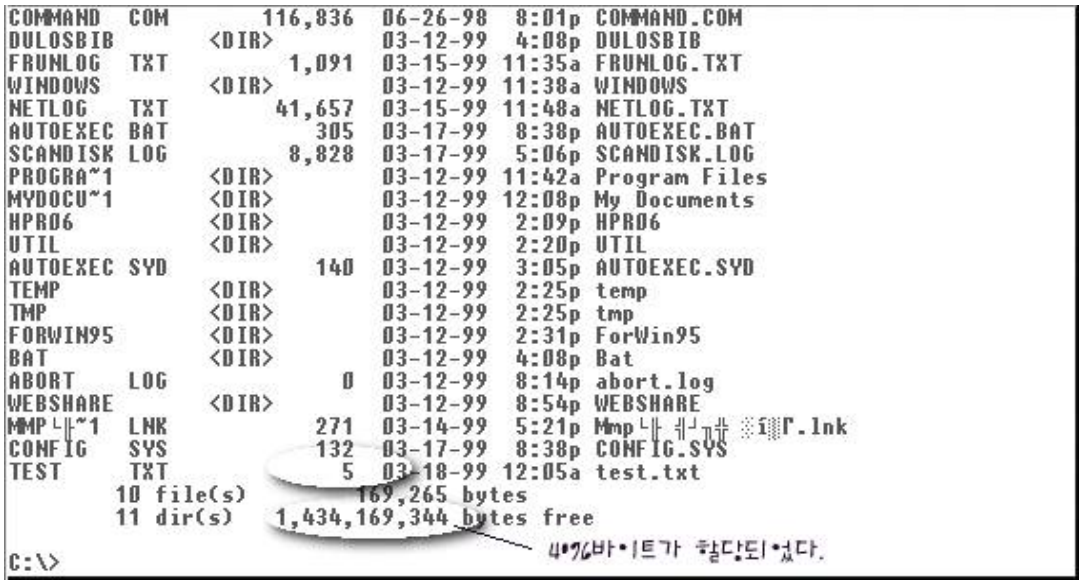
• 여기서 ^Z는 Ctrl+Z를 입력합니다(이것은 파일의 끝(EOF)을 나타내는 특수 문자를 입력합니다). 그러면 크기가 5인 텍스트파일이 만들어집니다.

abc가 3바이트, 줄의 끝에 붙는 '\r\n'이 2바이트입니다.



크기가 5바이트인 test.txt 파일의 생성: 이렇게 파일을 만드는 방법은 DOS 시절 많이 사용되었습니다. Windows 사용자는 notepad.exe를 사용하여 파일을 만들어도 좋습니다.

- 이제 사용된 디스크 공간을 확인해 봅시다.



- 위의 그림에서 보듯이, 5바이트 크기의 test.txt 파일이 만들어진 것을 확인할 수 있습니다.
- 그리고 사용할 수 있는 디스크 공간은 다음과 같이 출력되었습니다.

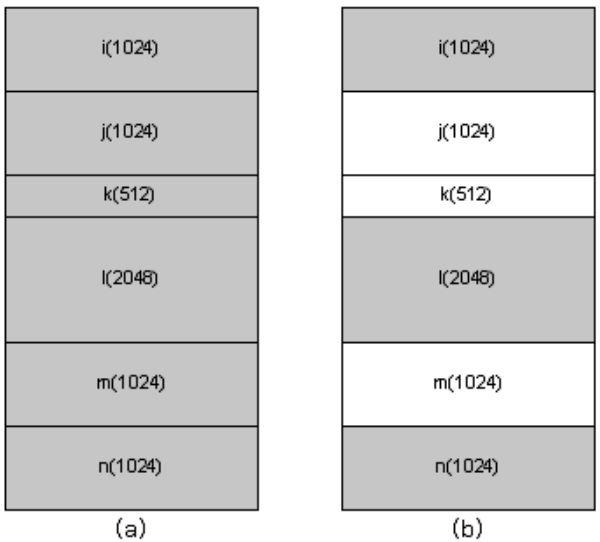
1,434,169,344

- 작업 전의 용량과 비교해 보면, 4096바이트(2^{12})가 줄어든 것을 알 수 있습니다.

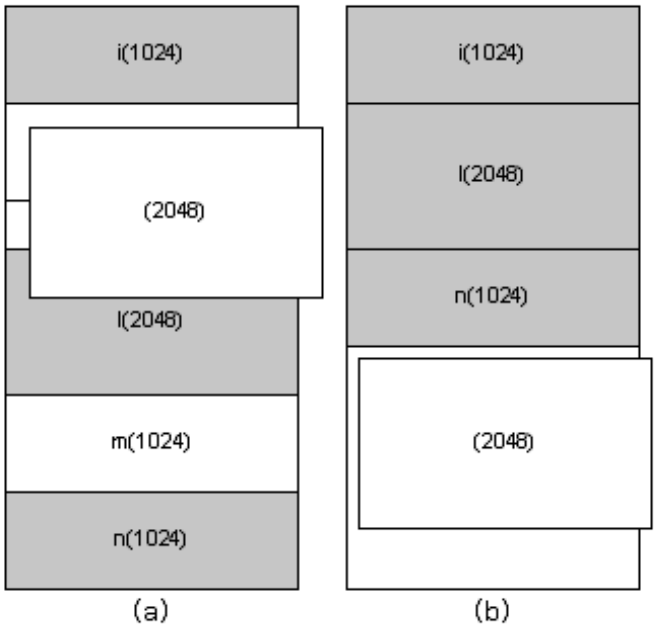
외부 단편화(external fragmentation)

• 사용할 수 있는 메모리 블록의 크기가, 할당하려는 메모리 블록보다 크다면 메모리 할당이 가능해야 합니다.
• 하지만, 메모리 할당이 실패하는 경우가 있습니다. 이것을 이해하기 위해서는 메모리 **외부 단편화(external fragmentation)**를 이해하는 것이 필요합니다.

• 아래 그림과 같은 메모리 할당을 생각해 봅시다. i(1024)의 형태는 i 블록이 1024바이트 할당되었음을 나타냅니다.



메모리 블록의 할당: (a) 6,656바이트의 메모리가 모두 할당된 상태. 사용 가능한 메모리 공간은 0바이트이므로 더 이상 동적 메모리 할당이 불가능합니다. (b) 사용자가 j,k와 m블록을 해제했습니다. 이제 사용 가능한 메모리 공간은 2,560바이트입니다.



2,048바이트의 할당: (a) 사용자가 2048바이트의 새로운 메모리 블록의 할당을 요구했습니다. 하지만, 연속된 메모리 공간을 할당해야 한다면 이것은 불가능합니다. (b) i,j와 n블록의 위치를 옮기면 이제 2048바이트의 연속된 메모리 공간을 할당하는 것이 가능합니다. 이렇게 연속된 메모리 블록의 할당을 위해 메모리 블록을 옮기는 것은 운영체제의 지원이 필요합니다. 이러한 작업을 **컴팩션**(compaction)이라고 합니다.

• 사용자가 다시 2048바이트의 메모리 블록을 요구했다고 합시다.
• 사용 가능한 메모리는 비록 2,560바이트임에도 불구하고, 2048바이트를 할당하는 것은 불가능합니다.
• 왜냐하면, 대부분의 컴파일러는 메모리 블록을 할당할 때, 연속된 메모리 블록을 요구하기 때문입니다.
• 이러한 상황을 **외부 단편화(external fragmentation)**라고 합니다. 즉 메모리 외부 단편화란 다음과 같은 상태입니다.

"할당된 메모리 블록이 연속적이지 않고, 불연속적인 상태"

• 외부 단편화 제거를 '**압축(compaction)**'이라고 합니다.
• DOS는 컴팩션을 지원하지 않았습니다. 그러므로, 도스에서 동적 메모리 할당은 조심스러운 코딩을 요구했습니다.

실습문제

1. 최소 블록의 크기가 32이고, 여분의 메모리로 4바이트가 할당 더 되는 어떤 시스템에서, 사용자가 유지하는 레코드 블록의 크기를 30이라고 합시다. 현재 사용 가능한 메모리가 20000이라면, 몇 개의 레코드를 더 할당할 수 있을까요?

2. 클러스터(cluster)의 크기가 4096일 때, 사용자가 10,000바이트의 파일을 만들면 디스크 내부 단편화로 사용할 수 없는 공간은 몇 바이트입니까?

3. 컴팩션(compaction)은 어느 시점에 행하는 것이 좋을까요? 또한 컴패션 알고리즘에는 어떤 종류가 있는지 설명하세요.

4. Win32의 파일 시스템 같은, 페이징(paging) 시스템에서는 왜 컴팩션이 필요 없을까요?

@