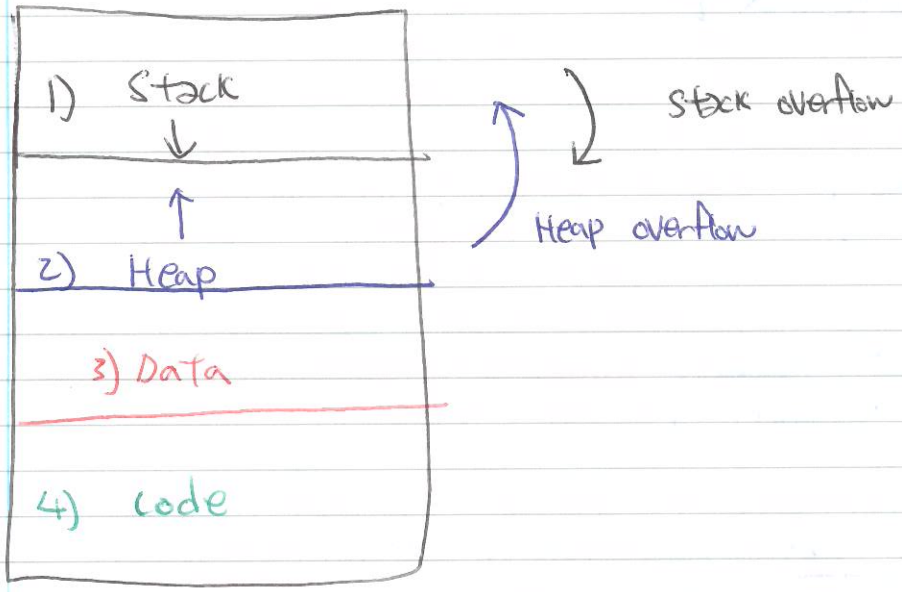


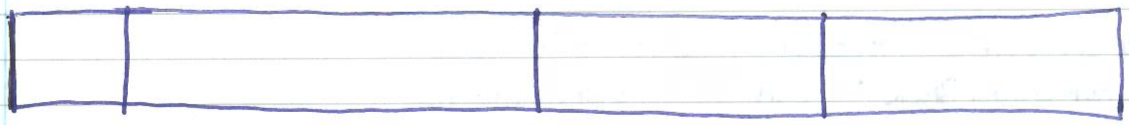
1) Stack — push / pop

- local variable
- return address
- parameters

- one method()
  → 이 기준으로 차례 차례 쌓인다.

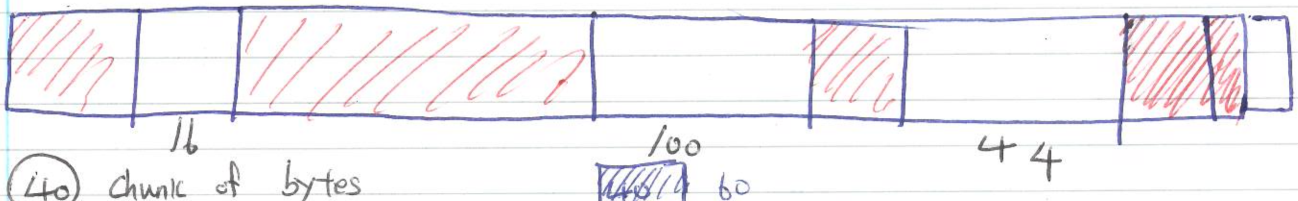
differences :
Stack → 차례차례 쌓인다.
Heap → 가능한 chunk를 찾아감

2) Heap — any thing with new chunk of memory



- So challenge is where should we put ?
  - First fit → $m \geq n$ 아무거나 중에서 가장 큰 것.
  - Best fit → $m \geq n$ 그 중에서 가장 smallest
  - worst fit → $m - n$

ex)



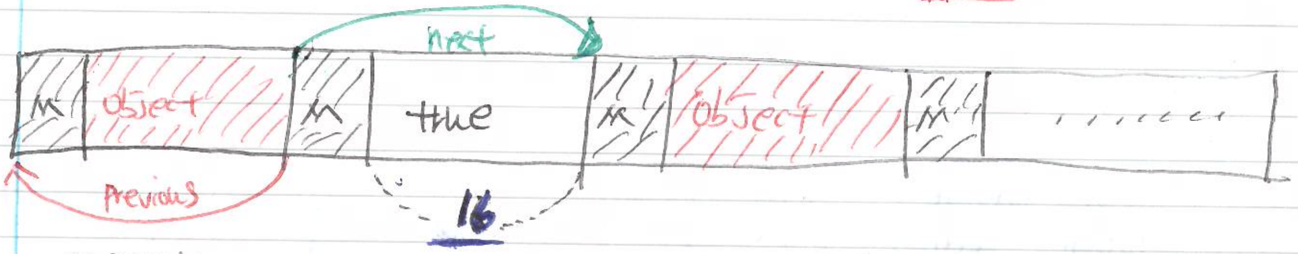
$\boxed{40}$ chunk of bytes $\boxed{40}$ 60

- Best fit → choices ($m \geq n$) ⟨ 100, $\boxed{44}$ ← Best fit
- first fit → 100 — $\boxed{100}$ ← worst fit
- worst fit → choices ($m - n$) ⟨

[illegible] Memory control block. → Double linked List.

```
Class    Mem Control block {

  Public :

  bool    available ;   // is it possible to use or not?
  int     size      ;   // size of memory


  Mem Control Block *   previous ;  // where is the previous?
  Mem Control Block *   next     ;  // what is the next?
};
```

ex) * This <u>memory block</u> stored between each <u>object.</u>



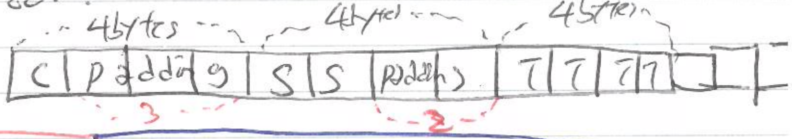
```
available  = true ;   // 메모리 적재 가능
size       = 16 ;     // 16 bytes 가능
next       ;          // 다음 항상 Head를 가리킨다.
previous   ;          // 전
```

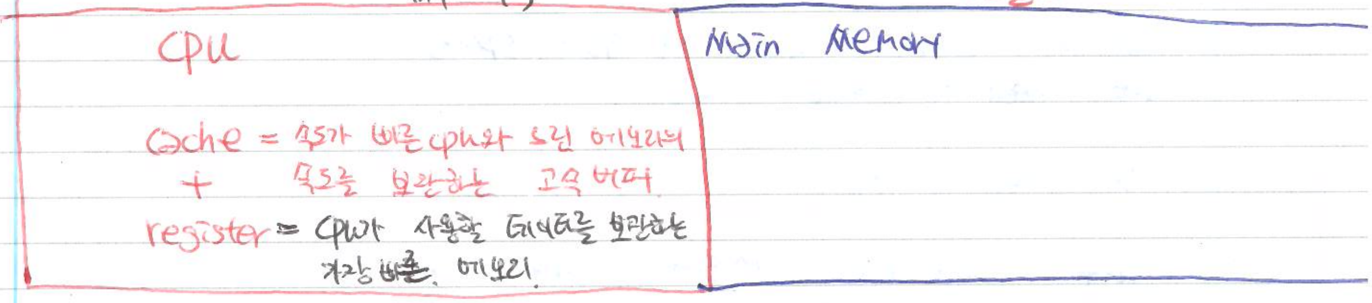
First fit

```
Memory Address   getMemory (int size) {
  MemControl Block *  curr  =  start of Heap;

   While ( curr  != null ptr &&
           ( !curr -> available || curr -> size (size)) {
           Curr -> curr -> next;
   }


};
```

★ Padding = memory block을 2칸씩 읽는 것을 방지하기 위해 메모리 안에 넣는 것.

ex) char c;
short s;
int i;





cache = 속도가 빠른 cpu와 느린 메모리의 속도를 보완하는 고속 버퍼
+
register = cpu가 사용할 데이터를 보관하는 가장 빠른 메모리

class byte padding = 메모리에서 cpu register로 한번에 읽을 수 있도록 cpu 레지스터의 읽기 블록에 맞춰 정렬하는 컴파일러의 최적화 방

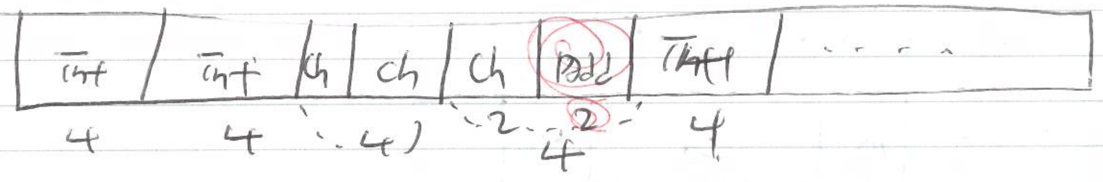
$$\begin{tikzcd} & \text{CPU} \arrow[dl] \arrow[dr] & \\ \text{32 bit} & & \text{64 bit} \end{tikzcd}$$

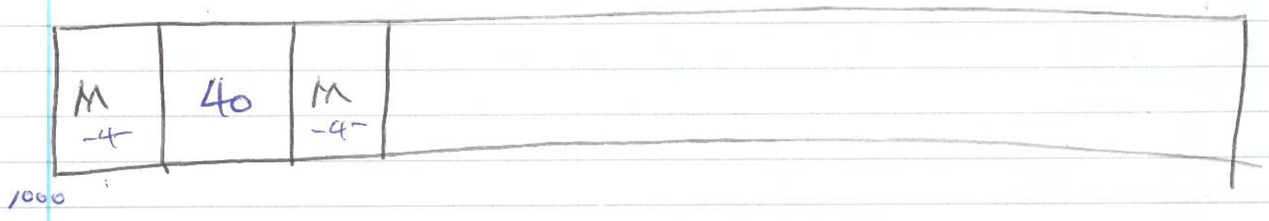
• 32 bit = 4byte must be multiple of 4

32 bit 에서는 4bytes씩 읽는다.
만약 padding을 대지 않는다면
memory block을 2칸씩 읽게 된다.

• 64 bit = 8 byte
must be multiple of 8
64 bit 에서는 8byte씩 읽는다.
//

ex) int int char char char int





4a

Int size = currsize - size - memory block.

[illegible]

Let's say 50 of spare spaces
we need to store 40

50 - 40 = 10

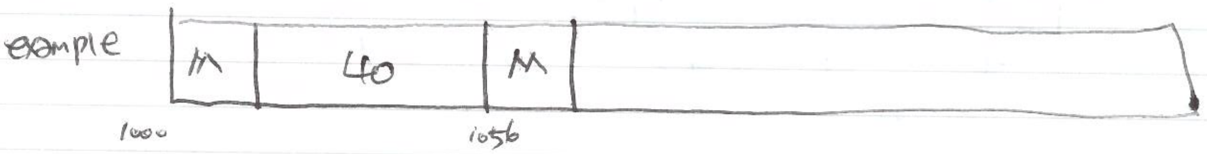
but we need memory block which is 16
10 - 16 = - 6
⇒ internal fragment

Addressing Memory
Address Binding → 프로그램의 명령어와 데이터를 Main memory에 적재할때 그것들의 주소를 결정
Memory block을 아끼려면 $\underbrace{\text{casting}}_{+16 \text{ bytes}}$을 하면 된다.

```
char * y = reinterpret_cast (char*) (x);
y + = 16 ;   // is now 16 bytes after x
```

Heap은 메모리를 Random한 장소에 저장 하지만
원하는 장소를 지정하고 싶다면 placement new를 사용.
원하는 주소를 직접 명시

ex) MemControlBlock* newMCB = new (z) MemControlBlock(...)

example


① 40을 저장 하려면 40 + 16 = 56이 총 필요함
40 = memory used.
16 = Memory Control block.

② 내가 원하는 곳에 저장 하려면.

```
char x = 1000 ;
x += 16 ;
x += 40 ;
```

new (x) MemControlBlock();
↳ Takes a memory address e.g char.

int size = currSize - size - Memory block.
40
10

Let's say 50 of spare spaces
we need to store 40

50 - 40 = 10

but we need memory block which is 16
10 - 16 = - 6
⇒ internal fragment

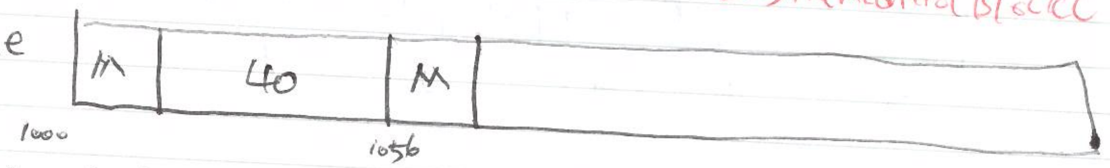
Addressing Memory
Address binding → 프로그램의 명령어와 데이터를 써서 memory에 적재할때 그것들의 주소를
Memory block을 아끼려면 casting을 하면 된다.
+16 bytes

```
char * y = reinterpret_cast (char*) (x);
y + = 16 ;   // is now 16 bytes after x
```

Heap은 메모리를 Random한 장소에 저장 하지만
만약 장소를 지정하고 싶다면 placement new를 사용.
만들 주소를 직접 명시

ex) MemControlBlock* newMCB = new (z) MemControlBlock(...)

example



① 40을 저장 하려면 40 + 16 = 56이 총 필요함
40 = memory used.
16 = Memory Control block.

② 내가 원하는 곳에 저장 하려면.

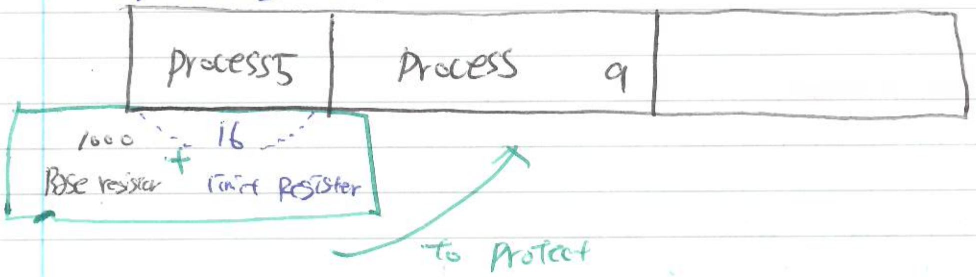
```
char  x = 1000 ;
      x*= 16 ;
      x += 40 ;
```

new (z) MemControlBlock();
↳ Takes a memory address e.g. char.

-4-

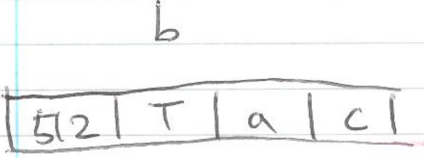
For a process

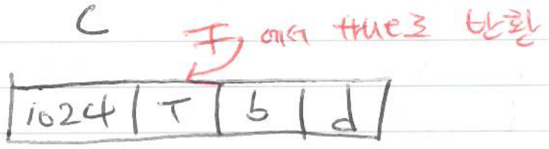
Base resister : Records where it's memory starts.
ㄴ 어디서 시작?
limit resister : Record how much memory is available
ㄴ 어느만큼 가능?



to protect
one process go into another process memory
만약 넘어가면 운영체제가 자동으로 종료 시킨다.

Address binding = logical Address → Physical address.



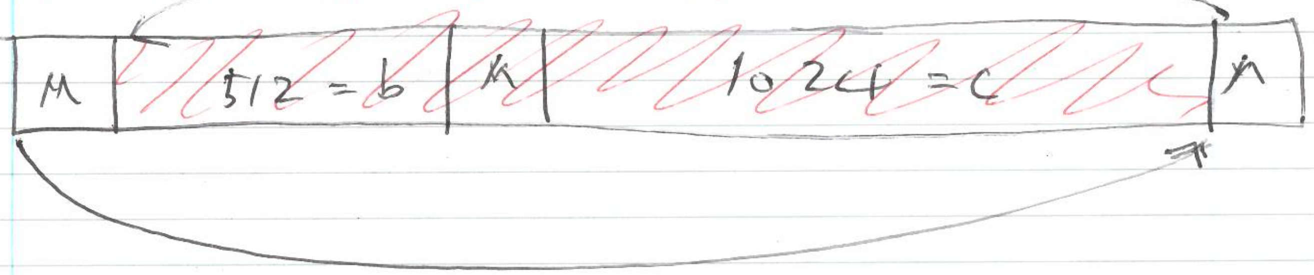


512 size
True 사용 가능
~~Next~~ Previous = a ;
~~previous~~ Next = c ;

1024 size
True 사용 가능
previous b
Next ~~d~~

만약 b는 사용 가능했고 c가 사용 되다가 반환 된다면
굳이 2개를 나눌 필요가 없다. 그럼으로 b+c를 시킨다.

* Memory Control block 제거법. b+c 합치면 Memory Control block 2개



512 + M C(16) + 1024 = 1552 생성.
512
16
1552

-5-

왜 Memory Control block은 Double linked list가 필요한가?

→ 만약 Next Previous 가 없다면 어느 공간이 비었는지 가르킬 수가 없다. 만약 Next 만 있다면 반환된 메모리 스페이스를 알 수가 없다.

Heap fragmentation

4 바이트의 메모리 블럭 4개가 있다고 16 바이트의 메모리를 적재 할 수 있는 것은 아니다 왜냐하면 4개가 연결되어 있는지 아닐수도 있으니까.

↳ 메모리 블럭이 Double linked list 이고 Best-fit concept 을 준수하는 이유.

Normal new() 과 placement new() 의 차이

	new()		Placement new()
new()	// 어딘가에 자동생성 Heap에서	/	new() // 내가 주소지정해서 생성
delete()	// 자동으로 제거 Heap에서	/	foo() // 내가 destructor 로 작성해서

① Y → foo() ; // foo() 로 destructor 부는 곳 직접해야

Memory Pool = keep small objects together to use memory efficiently.

Dave example

```
Class Dave {
    int a ;
    int b ;
}

int main {
}
```

Dave를 생성할때 8 bytes
but memory control 고려하면 16
= 엄청 비효율적
그래서 60명의 Dave를 위한
메모리 풀을 그림 그려봤다.