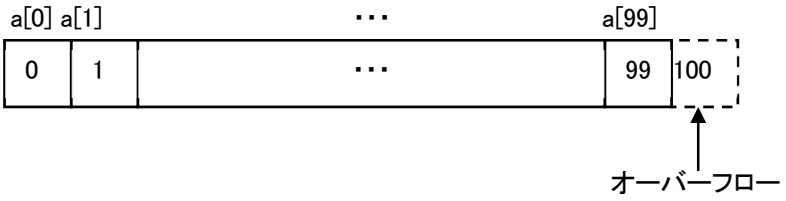
3.1 動的なメモリ確保

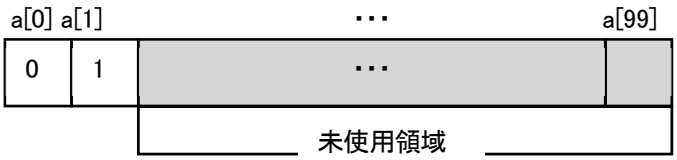
3.1.1 配列の問題点

多数の同じ型のデータを扱う手法として配列があるが、配列はコーディングの段階で要素数を決定する必要があるため、以下の問題が発生する。

- 定義した数以上のデータを格納できない。「 int a[100]; 」と定義した配列 a に 101 個以上のデータを格納できない。



- 実際には使用されない無駄な領域が発生する可能性がある。上記の配列 a において、格納したデータ数が 2 個であった場合、残りの 98 個分の領域は無駄となる。



本節では、この問題を解決するために、動的にメモリ領域を確保する手法を紹介する。

3.1.2　動的なメモリ確保

動的なメモリ確保とは、プログラム実行中に必要に応じてメモリ領域を確保することである。

C 言語では、malloc 関数によりメモリ領域を動的に確保する。

■ **malloc 関数**

```
#include<stdlib.h>
void  *malloc(size_t  size);
```

size バイトの領域を確保し、その領域の先頭アドレスを返す。確保に失敗した場合、NULL を返す。

- size_t 型はサイズを表す型で、一般に unsigned int 型である。
- 戻り値の (void *)型はポインタの一般形である。戻り値は利用する型にキャストする。

<例>　int 型の要素を格納するメモリ領域を動的に確保する。

```
int   *p;
p = (int *)malloc(4);          /* int 型は通常 4Byte のメモリ領域を必要とする */
```

malloc 関数は、メモリの空き容量が足りなくなった場合には領域の確保に失敗することがある。malloc 関数は領域確保に失敗すると NULL を返す。malloc 関数を使用する時には、戻り値を確認し、NULL であったときにはエラー処理をする必要がある。

【 例 】

```
int *p;
p = (int *)malloc(4);
/* malloc 関数の戻り値が NULL ならばエラー処理を行う */
if( p == NULL ){
    fprintf(stderr,"メモリの確保に失敗しました。¥n");
    exit(1);
}
```

本テキストでの malloc 関数のエラー処理

実際にプログラムを作成する時は必ずエラー処理を行う。しかし本テキストのプログラムでは、ソースプログラムが複雑になり、説明が不明確になることを避けるため、メモリ確保に失敗したときのエラー処理を省略する場合がある。

malloc 関数の引数は、確保する領域のサイズであるが、直接数値で指定すると以下の理由によりプログラムの汎用性を損なうことになる。

- データ型のサイズはアーキテクチャに依存する。
- 構造体のサイズは単純にメンバ変数の合計値とはならない

この問題に対処するために、sizeof 演算子を使用する。

■ **sizeof 演算子**

sizeof(データ型) または sizeof(変数名)

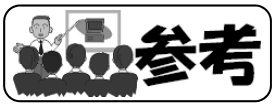
指定した変数や型のサイズ(バイト単位)を求める。

<例>

```
sizeof(int);        / * int 型のサイズを求める* /
sizeof(a);      / *変数 a のサイズを求める* /
```

<例> int 型の要素を格納するメモリ領域を動的に確保し、その領域に 10 を代入する

```
int   *p;
p = (int *)malloc(sizeof(int));
*p = 10;
```



データ型の別名

typedef 宣言を用いることにより、size_t 型のように既存のデータ型や構造体に明確な意味を持った別名を与えることができる。これによりソースプログラムの可読性や移植性が向上する。

■ **typedef 宣言**

typedef　データ型　別名;

既存のデータ型、構造体に別名を与える。

<例>

```
typedef   unsigned  int   size_t;  /* unsigned int と size_t を同一のものとして定義 */
```

<例>

```
typedef   struct   data   DATA;  /* struct   data と DATA を同一のものとして定義 */
```

malloc 関数で動的に確保した領域は、free 関数によって解放する。free 関数によるメモリの解放を行わないと、メモリを確保したままプログラムが終了してしまうことがある。

■ free 関数

```
#include<stdlib.h>
void free(void *p);
```

引数で指定された領域を解放する。

※ 解放するポインタを (void *)型にキャストする必要はない。

<例> int 型の要素 10 個分の領域を動的に確保し、その領域を解放する

```
int   *p;
p = (int *)malloc(sizeof(int) * 10);
free(p);          /*   pが指す領域を解放する  */
```

メモリリーク

メモリリークとは、アプリケーションが動的に確保したメモリ領域が解放されないままメモリ空間を占有し続けるために、使用可能なメモリ領域が減少していく現象である。メモリリークは、システムのパフォーマンスを低下させ、場合によってはシステム停止につながる。従って、メモリ領域を動的に確保した場合は、明示的に解放する必要がある。

【例題1】メモリを動的に確保するプログラム(dmemory)

【実行結果】

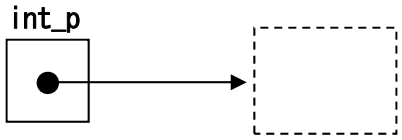
```
C:¥Cpro2>dmemory ↵
int->[10] : double->[4.560000]

C:¥Cpro2>
```

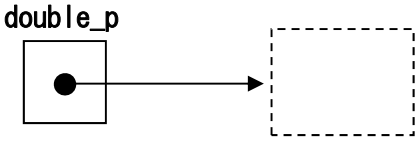
dmemory.c

```
 1 : #include <stdio.h>
 2 : #include <stdlib.h>   /* malloc 関数、free 関数を利用する時には必要 */
 3 :
 4 : int main(void)
 5 : {
 6 :    int *int_p;                /* int 型へのポインタを定義 */
 7 :    double *double_p;          /* double 型へのポインタを定義 */
 8 :
 9 :    int_p = (int *) malloc(sizeof(int));     /* メモリの動的確保 */
10 :    if(int_p == NULL)                        /* 確保に失敗した時の処理 */
11 :    {
12 :       fprintf(stderr,"Out of Memory¥n");
13 :       exit(1);
14 :    }
15 :
16 :    double_p = (double *) malloc(sizeof(double));/* メモリの動的確保 */
17 :    if(double_p == NULL)                    /* 確保に失敗した時の処理 */
18 :    {
19 :       fprintf(stderr,"Out of Memory¥n");
20 :       exit(1);
21 :    }
22 :
23 :    *int_p = 10;
24 :    *double_p = 4.56;
25 :
26 :    printf("int->[%d] : double->[%f]¥n", *int_p, *double_p);
27 :
28 :    free(int_p);               /* 動的に確保した領域を解放 */
29 :    free(double_p);
30 :
31 :    return 0;
32 : }
```

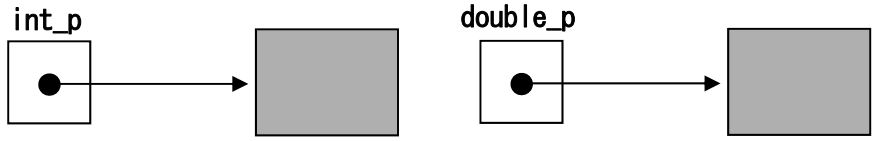
6行目： int 型領域へのポインタ変数 int_p を定義



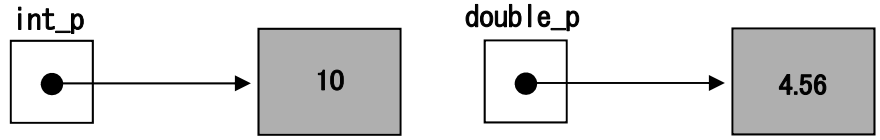
7行目： double 型領域へのポインタ変数 double_p を定義



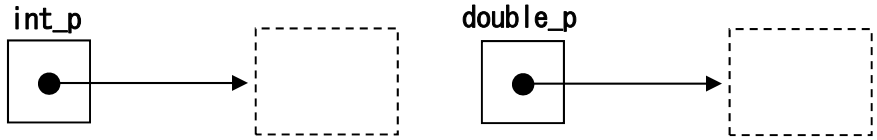
9,16行目： 領域を確保する



23～24行目： 値を代入する



28～29 行目： 確保した領域を解放する



次の例は int 型の要素 10 個分の領域を動的に確保し、その領域を解放するプログラムの一部である。空欄にあてはまる語を記述せよ。

```
int _____ ;   /＊ ポインタ p の定義 ＊/
p = (_____)malloc(sizeof(_____)＊10);   /＊ メモリの動的確保 ＊/


free(_____);   /＊ 動的確保したメモリの解放 ＊/
```