

ALOKASI MEMORI DINAMIS

Tujuan :

1. Menjelaskan cara mencari ukuran sebuah obyek
2. Menjelaskan cara mengalokasikan memori
3. Menjelaskan cara membebaskan memori
4. Menjelaskan cara merealokasikan memori

Problem yang terjadi ketika bekerja dengan tipe data array adalah bahwa array cenderung menggunakan ukuran yang tetap, yang telah dipesan di awal deklarasi program. Ukuran ini tidak dapat diperbesar atau diperkecil ketika ternyata kebutuhannya menghendaki demikian.

Pada beberapa kasus, ukuran dari sebuah obyek tidak bisa dipastikan sampai dengan waktu dieksekusi (*run time*). Sebagai contoh, panjang dari string yang dimasukkan oleh user tidak bisa diketahui sebelum eksekusi, sehingga ukuran array bisa jadi bergantung pada parameter yang tidak bisa diketahui sebelum eksekusi program.

Alokasi memori (*memory allocation*) menyediakan fasilitas untuk membuat ukuran buffer dan array secara dinamik. Dinamik artinya bahwa ruang dalam memori akan dialokasikan ketika program dieksekusi. Fasilitas ini memungkinkan user untuk membuat tipe data dan struktur dengan ukuran dan panjang berapapun yang disesuaikan dengan kebutuhan di dalam program.

12.1 Mengetahui kebutuhan alokasi memori dengan fungsi `sizeof()`

Ukuran dari obyek yang akan dialokasikan lokasinya sangat bervariasi bergantung dengan tipe obyeknya. Ukuran ini berkenaan dengan jumlah byte memori yang akan disediakan, misalnya data bertipe `char` akan disimpan dalam 1 byte sedangkan `float` 4 byte. Pada sebagian besar mesin 32 bit `int` disimpan dalam 4 byte.

Cara paling mudah untuk menentukan ukuran obyek yang akan disimpan adalah dengan menggunakan fungsi `sizeof()`. Fungsi ini dapat digunakan untuk mendapatkan ukuran dari berbagai tipe data, variabel ataupun struktur. *Return value* dari fungsi ini adalah ukuran dari obyek yang bersangkutan dalam byte. Fungsi `sizeof()` dapat dipanggil dengan menggunakan sebuah obyek atau sebuah tipe data sebagai parameternya.

```
/* File program : size.c
Menggunakan fungsi sizeof() untuk menentukan ukuran obyek */

#include <stdio.h>

typedef struct employee_st {
    char name[40];
    int id;
} Employee;

main()
{
    int myInt;
    Employee john;

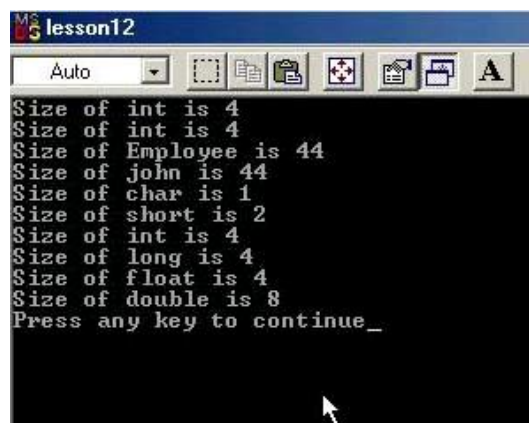
    printf("Size of int is %d\n",sizeof(myInt));
```

```

    /* The argument of sizeof is an object */
    printf("Size of int is %d\n",sizeof(int));
    /* The argument of sizeof is a data type */
    printf("Size of Employee is %d\n",sizeof(Employee));
    /* The argument of sizeof is an object */
    printf("Size of john is %d\n",sizeof(john));
    /* The argument of sizeof is a data type */
    printf("Size of char is %d\n",sizeof(char));
    printf("Size of short is %d\n",sizeof(short));
    printf("Size of int is %d\n",sizeof(int));
    printf("Size of long is %d\n",sizeof(long));
    printf("Size of float is %d\n",sizeof(float));
    printf("Size of double is %d\n",sizeof(double));
    return 0;
}

```

Bila dijalankan akan didapatkan keluaran sebagai berikut :



```

MS-DOS [C:\] lesson12
Auto
Size of int is 4
Size of int is 4
Size of Employee is 44
Size of john is 44
Size of char is 1
Size of short is 2
Size of int is 4
Size of long is 4
Size of float is 4
Size of double is 8
Press any key to continue_

```

Gambar 12.1 Keluaran program size.c

12.2 Mengalokasikan memori dengan fungsi malloc()

Fungsi standar yang digunakan untuk mengalokasikan memori adalah malloc(). Bentuk prototypenya adalah

```
void *malloc(int jml_byte);
```

Banyaknya byte yang akan dipesan dinyatakan sebagai parameter fungsi. Return value dari fungsi ini adalah sebuah pointer yang tak bertipe (*pointer to void*) yang menunjuk ke buffer yang dialokasikan. Pointer tersebut haruslah dikonversi kepada tipe yang sesuai (dengan menggunakan *type cast*) agar bisa mengakses data yang disimpan dalam buffer. Jika proses alokasi gagal dilakukan, fungsi ini akan memberikan return value berupa sebuah pointer NULL. Sebelum dilakukan proses lebih lanjut, perlu terlebih dahulu dipastikan keberhasilan proses pemesanan memori, sebagaimana contoh berikut :

```

char *cpt;
...
if ((cpt = (char *) malloc(25)) == NULL)
{
    printf("Error on malloc\n");
    exit(0);
}

```

```

    }

/* File program : alokasi1.c
Menggunakan fungsi malloc() untuk mengalokasikan memori */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main()
{
    char s1[] = "This is a sentence";
    char *pblok;

    pblok = (char *) malloc(strlen(s1) + 1);
    /* Remember that strings are terminated by the null
       terminator, '\0', and the strlen returns the length
       of a string not including the terminator */
    if (pblok == NULL)
        printf("Error on malloc\n");
    else {
        strcpy(pblok, s1);
        printf("s1: %s\n", s1);
        printf("pblok: %s\n", pblok);
        return 0;
    }
}

```

12.3 Membebaskan kembali memori dengan fungsi `free()`

Jika bekerja dengan menggunakan memori yang dialokasikan secara dinamis, maka seorang programmer haruslah membebaskan kembali memori yang telah selesai digunakan untuk dikembalikan kepada sistem. Setelah suatu ruang memori dibebaskan, ruang tersebut bisa dipakai lagi untuk alokasi variabel dinamis lainnya. Untuk itu digunakan fungsi `free()` dengan prototype sebagai berikut :

```
void free(void *pblok);
```

dengan `pblok` adalah pointer yang menunjuk ke memori yang akan dibebaskan. Dengan demikian memori yang telah dibebaskan tersebut akan dapat digunakan oleh bagian lain dari program. Dalam hal ini, pointer `pblok` tidak perlu di-cast kembali ke void terlebih dahulu. Compiler otomatis telah menangani proses ini.

Sebuah contoh sederhana mengenai pengalokasian, pengaksesan terhadap variabel dinamis, dan pembebasan kembali ruang memori yang telah dipesan ditunjukkan pada contoh program `alokasi2.c` di bawah ini.

```

/* File program : alokasi2.c
Menggunakan fungsi free() untuk membebaskan kembali memori */

#include <stdio.h>
#include <stdlib.h>

```

```

main()
{
    char *pblok;

    pblok = (char *) malloc(500 * sizeof(char));

    if (pblok == NULL)
        puts("Error on malloc");
    else {
        puts("OK, alokasi memori sudah dilakukan");
        puts("-----");
        free(pblok);
        puts("Blok memori telah dibebaskan kembali");
    }
}

```

Contoh eksekusi :

```

OK, alokasi memori sudah dilakukan
-----
Blok memori telah dibebaskan kembali

```

Catatan : Prototype fungsi `malloc()` dan `free()` terdapat pada file `stdlib.h`

Pada pernyataan

```
pblok = (char *) malloc(500 * sizeof(char));
```

`(char *)` merupakan upaya untuk mengkonversikan pointer hasil `malloc()` agar menunjuk ke tipe `char` (sebab `pblok` dideklarasikan sebagai pointer yang menunjuk obyek bertipe `char` / *pointer to char*). Sedangkan

```
malloc(500 * sizeof(char))
```

digunakan untuk memesan ruang dalam memori sebanyak : 500 x ukuran `char` atau : 500 x 1 byte.

12.4 Mengalokasikan ulang memori dengan fungsi `realloc()`

Bisa jadi terjadi ketika hendak mengalokasikan memori, user tidak yakin berapa besar lokasi yang dibutuhkannya. Misalnya user tersebut memesan 500 lokasi, ternyata setelah proses pemasukan data kebutuhannya melebihi 500 lokasi menjadi 600. Maka user tersebut dapat mengalokasikan ulang memori yang dipesannya dengan menggunakan fungsi `realloc()`. Fungsi ini akan mengalokasikan kembali pointer yang sebelumnya telah diatur untuk menunjuk sejumlah lokasi, memberinya ukuran yang baru (bisa jadi lebih kecil atau lebih besar). Sebagai contoh, adalah `pblok` adalah pointer yang menunjuk kepada 500 lokasi `char`, maka user bisa mengalokasikan ulang agar pointer `pblok` menunjuk kepada 600 lokasi `char` sebagai berikut :

```

...
pblok = (char *) malloc(500 * sizeof(char));
...
pblok = realloc(pblok, 600 * sizeof(char));

```

Fungsi `realloc()` akan merelokasi blok memori lama menjadi sebesar kapasitas baru yang dipesan. Jika `realloc()` berhasil melakukan relokasi baru, fungsi ini memberikan return value berupa pointer yang sama, meng-copy data lama ke lokasi baru dan mengarahkan pointer ke sejumlah lokasi baru tersebut (dalam hal ini setelah berhasil meng-copy data lama, fungsi ini juga otomatis membebaskan blok memori yang lama). Namun jika fungsi ini tidak berhasil menemukan lokasi baru yang mampu menampung kapasitas baru yang dipesan oleh user, maka fungsi ini memberikan return value berupa `NULL`. Itulah sebabnya, cara terbaik adalah terlebih dahulu

mengecek hasil `realloc()` untuk memastikan hasilnya bukan NULL pointer, sebelum kemudian menumpuki pointer lama dengan return value dari `realloc()`. Untuk itu bisa dibuat potongan program sebagai berikut :

```
char *newp;

newp = realloc(pblok, 600 * sizeof(char));
if(newp != NULL)
    pblok = newp;
else {
    printf("out of memori\n");
    /* exit or return */
    /* but pblok still points at 500 chars */
}
```

Pada potongan program tersebut, jika `realloc()` memberikan return value selain NULL pointer, berarti proses relokasi sukses, maka pointer `pblok` diset sama dengan `newp`. Namun, jika proses `realloc()` gagal (return valuenya NULL), maka pointer lama (yaitu `pblok`) masih menunjuk kepada 500 data asal.

12.5 Beberapa kesalahan umum

Ada beberapa kesalahan yang seringkali terjadi ketika bekerja dengan alokasi memori secara dinamis. Beberapa di antaranya adalah :

❑ Mengabaikan pengecekan hasil operasi `malloc()`

Ketika gagal mengalokasikan memori, `malloc()` akan memberikan return value berupa NULL pointer. Jika hal ini terjadi, maka ketika pointer tersebut selanjutnya diakses dalam program akan mengakibatkan program mengalami ‘crash’. Sebaliknya, jika hasil operasi `malloc()` dicek terlebih dahulu, akan memungkinkan untuk bisa keluar dari program atau mengabaikan bagian-bagian dari program yang menggunakan buffer yang dialokasikan tadi.

❑ Mengabaikan pembebasan memori setelah digunakan

Ada batasan besarnya kapasitas memori yang tersedia untuk dipakai oleh sebuah program. Jika proses menjalankan program yang panjang mengabaikan pembebasan memori, bahkan terus menerus mengalokasikan yang tersedia, maka program tersebut akan kehabisan memori (*run out*). Kondisi ini disebut dengan *memory leak*, yang merupakan sebuah *bug* yang sangat serius. Hal ini bisa berimplikasi pada unjuk kerja keseluruhan sistem.

❑ Pointer yang mengambang (*Dangling Pointer*)

Setelah pembebasan sekumpulan lokasi memori, pointer yang menunjuk kepada lokasi yang telah dibebaskan tersebut masih berisi alamat memori dari titik awal lokasi tersebut. Secara alami, setelah proses pembebasan, maka pointer penunjuk lokasi tersebut tidak bisa digunakan lagi. Jika pointer ini kemudian digunakan lagi dalam program, maka apapun nilai yang ada pada lokasi tersebut akan bisa digunakan (yang bisa jadi merupakan nilai dari variabel lain). Cara terbaik untuk menghindari hal ini adalah dengan secara langsung mereset pointer menjadi NULL setelah lokasi memori yang ditunjuknya dibebaskan.

```
free(pt);
pt = NULL;
```

Dengan demikian, pointer yang menunjuk kepada lokasi yang telah dibebaskan juga telah dinon-aktifkan.