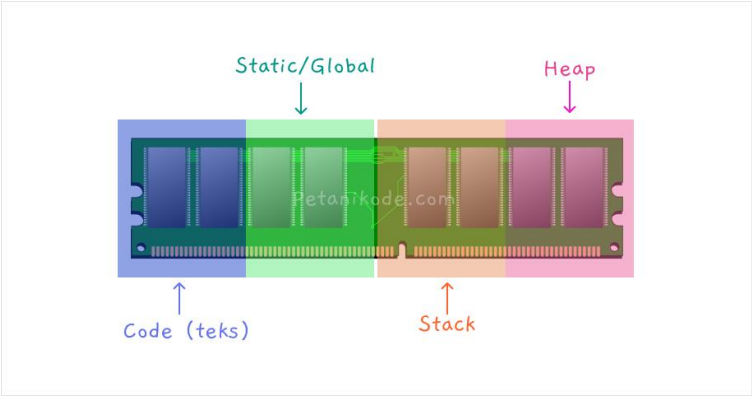
- **Memory Allocation & De-allocation**

  Alokasi memori merupakan upaya program untuk “memesan” ukuran memori di saat sebelum atau saat program dieksekusi, sedangkan de-allocation bertujuan untuk mengembalikan/melepaskan memori yang tidak terpakai sehingga tidak ada memori yang sia-sia. Teknik alokasi memori terbagi atas dua yaitu statis dan dinamis. Sesuai dengan definisinya, alokasi memori statis meminta programmer untuk menentukan ukuran dari suatu data pada saat menulis program sedangkan alokasi memori dinamis, ukuran data tidak ditulis secara eksplisit. Contoh yang sudah kita praktekan selama ini adalah membuat sebuah array dengan ukuran tertentu. Cara yang pertama (statis) adalah dengan langsung mencantumkan ukuran dari array. Jadi saat *compile time* atau sebelum program berjalan, array akan didaftarkan terlebih dahulu ke dalam memori dengan ukuran yang telah ditentukan. Sedangkan cara kedua, sebuah array akan didaftarkan dalam memori saat *run-time* atau saat program sedang berjalan. Walaupun cara kedua merupakan alokasi memori dinamis, hal tersebut tidak menyebabkan array akan tersimpan ke dalam *heap* melainkan array akan tetap tersimpan ke dalam *local stack.* Teknik seperti ini akan menimbulkan masalah lain jika ukuran array sangat besar. Dalam beberapa sistem operasi, ukuran *local stack* akan dibatasi dan jika suatu variabel memiliki ukuran yang melebihi ukuran *local stack* yang terjadi adalah *stack overflow* dimana local stack tidak dapat memuat variabel yang besar tersebut.

  Saat program dijalankan, komputer akan mengalokasikan memori (RAM) menjadi empat bagian :

  1. Stack code, digunakan untuk menyimpan code program.
  2. Static/global, digunakan untuk menyimpan variabel global atau statis.
  3. Local stack, digunakan untuk menyimpan variabel local.
  4. Heap, digunakan untuk menyimpan variabel dengan cara alokasi dinamis.



Perbedaan lainnya adalah, de-allocation pada alokasi memori statis akan bekerja pada saat program selesai dieksekusi sedangkan pada alokasi dinamis proses de-allocation dapat dikerjakan walaupun program sedang berjalan.

- ## Alokasi Memori Dinamis

Alokasi memori dinamis dapat diamanfaatkan untuk menambah dan mengurangi ukuran penggunaan memori ketika program dijalankan. Sehingga memori komputer dapat digunakan dengan optimal. Dalam pemrograman C terdapat empat fungsi utama untuk alokasi memori dinamis yaitu `malloc()`, `calloc()`, `realloc()` dan `free()`. Sebagai catatan, keempat fungsi yang telah disebutkan tersedia dalam library stdlib.h.

- **`Malloc()`**

Fungsi ini digunakan untuk mengalokasikan memori secara dinamis dan datanya akan tersimpan ke dalam *heap*. Berikut adalah syntax dari fungsi `malloc()`.

```
ptr = (cast_type *) malloc (byte_size);
```

Keterangan :

`ptr` adalah variabel pointer

`cast_type` adalah type data dari variabel pointer `ptr`

`Byte_size` adalah argument untuk menentukan ukuran alokasi memori dalam satuan byte

Perhatikan code berikut:

```
#include "stdio.h"
#include "stdlib.h"

int main(){
int N; //variabel menampung nilai panjang array
printf("Panjang array = "); //prompt
scanf("%d",&N); //input user

int *arr; //deklarasi pointer
arr = (int *)malloc(N*sizeof(int)); //membuat array dengan cara
alokasi memori dinamis

for(int i=0; i<N; i++){ //loop hingga N
arr[i] = i; //isi setiap indeks array dengan nilai i
}
printf("Ukuran array = %d (dalam byte)\n",N*sizeof(*arr));
printf("Panjang array = %d ",N*sizeof(*arr)/sizeof(*arr));
      Return 0;
}
```

```
Panjang array = 100
Ukuran array = 400 (dalam byte)
Panjang array = 100
--------------------------------
Process exited after 0.9448 seconds with return value 0
Press any key to continue . . .
```

Code di atas mengilustrasikan alokasi memori dinamis. Program akan membuat sebuah array dengan panjang sebanyak `N` dimana `N` adalah nilai yang diinput oleh user. Setelah itu dilanjutkan dengan membuat sebuah array dengan cara alokasi memori dinamis (menggunakan fungsi `malloc()`). Berikutnya adalah melakukan looping untuk mengisis setiap indeks dari array. Di akhir program, akan menampilkan ukuran array (dalam byte) dan panjang dari array. Fungsi `sizeof()` berfungsi untuk mendapatkan ukuran data dalam satuan byte. Sekilas cara ini hampir sama dengan code berikut

```
#include "stdio.h"

int main(){
int N;
printf("Panjang array = ");
scanf("%d",&N);

int arr[N];

for(int i=0; i<N; i++){
arr[i] = i;
}

printf("Ukuran array = %d (dalam byte)\n",N*sizeof(arr[0]));
printf("Panjang array = %d ",N*sizeof(*arr)/sizeof(arr[0]));
return 0;
}
```

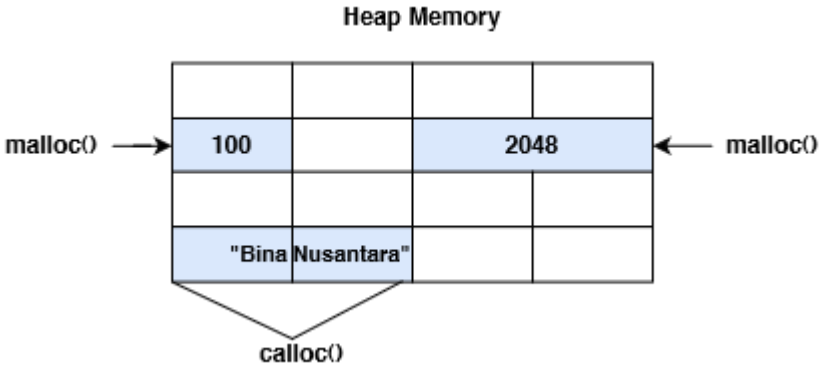
```
Panjang array = 100
Ukuran array = 400 (dalam byte)
Panjang array = 100
--------------------------------
Process exited after 1.557 seconds with return value 0
Press any key to continue . . .
```

Namun, cara kedua akan menyimpan array ke dalam *local stack* sedangkan program pertama akan menyimpan array ke dalam *heap*. Waktu yang tepat menggunakan program pertama adalah ketika data array yang ingin dibuat memiliki ukuran yang besar.

- **`Calloc()`**

Fungsi ini memiliki tujuan yang sama dengan fungsi `malloc()`, namun yang menjadi pembeda diantara keduanya adalah fungsi `malloc()` hanya menggunakan 1 block memori dengan ukuran tertentu sedangkan fungsi `calloc()` akan menggunakan beberapa block memori untuk satu variabel. Berikut adalah ilustrasi alokasi memori dari

kedua fungsi yang telah disebutkan. Nilai 100 dan 123 masing-masing tersimpan dalam satu block



sedangkan string “Bina Nusantara” tersimpan ke dalam dua block memori menggunakan fungsi `calloc()`. Berikut adalah syntax dari fungsi calloc().

```
ptr = (cast_type *) calloc (block_size, byte_size);
```

Keterangan :

`ptr` adalah variabel pointer

`cast_type` adalah type data dari variabel pointer `ptr`

`block_size` adalah ukuran atau jumlah block yang digunakan

`byte_size` adalah argument untuk menentukan ukuran alokasi memori dalam satuan byte

Berikut adalah contoh implementasi fungsi calloc():

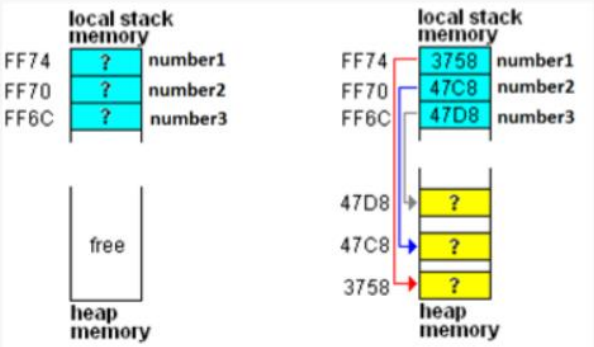
```
#include "stdio.h"
#include "stdlib.h"

int main(){
int N; //variabel menampung nilai panjang array
printf("Panjang array = "); //prompt
scanf("%d",&N); //input user

int *arr; //deklarasi pointer
arr = (int *)calloc(2,N*sizeof(int)); //membuat array dengan cara
alokasi memori dinamis menggunakan fungsi calloc()

for(int i=0; i<N; i++){ //loop hingga N
arr[i] = i; //isi setiap indeks array dengan nilai i
}
printf("Ukuran array = %d (dalam byte)\n",N*sizeof(*arr));
printf("Panjang array = %d ",N*sizeof(*arr)/sizeof(*arr));
      Return 0;
}
```

```
Panjang array = 100
Ukuran array = 400 (dalam byte)
Panjang array = 100
--------------------------------
Process exited after 1.612 seconds with return value 0
Press any key to continue . . .
```

Program di atas memiliki kesamaan dengan program yang diilustrasikan pada fungsi `malloc()` yaitu membuat sebuah array dengan alokasi memori dinamis. Namun fungsi yang digunakan berbeda dan juga terdapat adanya argument lain yang harus didefiniskan yaitu `block_size`. Perbedaan lainnya antara `malloc()` dan `calloc()` adalah performa fungsi `calloc()` lebih lambat bila dibandingkan dengan fungsi `malloc()`.

Kedua fungsi yang telah dijelaskan sebelumnya akan menghasilkan sebuah pointer. Itulah alasan kenapa sebelum menggunakan fungsi tersebut kita perlu mendeklarasi sebuah variable bertype pointer. Pointer ini nantinya akan berisi informasi alamat memori pada *heap*. Berikut adalah ilustrasi alokasi memori yang terjadi :

```
int *number1, *number2, *number3;
number1 = (int *) malloc (sizeof(int));
number2 = (int *) malloc (sizeof(int));
number3 = (int *) malloc (sizeof(int));
```



Gambar di atas menunjukkan terdapat 3 variabel yang akan dialokasikan secara dinamis menggunakan fungsi `malloc()`. Kita ambil contoh variabel pertama yaitu `*number1`. Variabel ini awalnya akan terdaftar di dalam *local stack* pada alamat `FF74`. Kemudian ketika dialokasi menggunakan fungsi `malloc()` maka isi dari variabel `*number1` adalah alamat memori yang digunakan pada *heap* untuk menyimpan data yaitu `3758`.

- `realloc()`

Fungsi ini digunakan untuk mengalokasikan ulang memori dari variabel yang telah dialokasikan dengan fungsi `malloc()` ataupun `calloc()`. Fungsi ini biasanya digunakan untuk mengubah ukuran alokasi memori pada suatu variabel karena

terkadang ketika program sedang berjalan, kita ingin menambah atau mengurangi ukuran dari data misalnya array. Berikut adalah syntax dari fungsi `realloc()` :

```
ptr = (cast_type *) realloc (*allocated_ptr, byte_size);
```

Keterangan :

`ptr` adalah variabel pointer yang digunakan untuk relokasi

`cast_type` adalah type data dari variabel pointer `ptr`

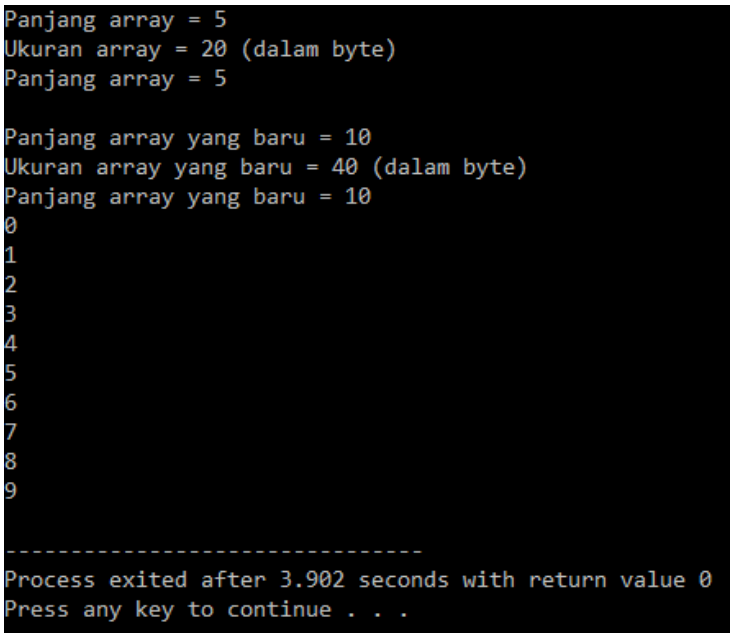
`*allocated_ptr` adalah variabel pointer yang telah dialokasi sebelumnya

`byte_size` adalah argument untuk menentukan ukuran alokasi memori dalam satuan byte

Perhatikan code berikut:

```
#include "stdio.h"
#include "stdlib.h"

int main(){
int N,newN; //variabel ukuran array
printf("Panjang array = ");
scanf("%d",&N); //input ukuran array awal

int *arr; //deklarasi pointer
arr = (int *)malloc(N*sizeof(int)); //membuat array dengan cara
alokasi memori dinamis


for(int i=0; i<N; i++){ //loop sebanyak N
arr[i] = i; //isi data setiap indeks array dengan i
}
printf("Ukuran array = %d (dalam byte)\n",N*sizeof(*arr));
printf("Panjang array = %d \n",N*sizeof(*arr)/sizeof(*arr));

printf("\nPanjang array yang baru = ");
scanf("%d",&newN); //input panjang array yang baru

arr = (int *)realloc(arr,newN*sizeof(int)); //mengalokasikan ulang
array yang sudah dibuat. Panjang array sesuai dengan newN
for(int i=N;i<newN;i++){ //loop dari N hingga newN
arr[i] = i; //mengisi data array pada indeks yang baru      }

printf("Ukuran array yang baru = %d (dalam
byte)\n",newN*sizeof(*arr));
printf("Panjang array yang baru = %d
\n",newN*sizeof(*arr)/sizeof(*arr));

for(int i=0;i<newN;i++) printf("%d\n",arr[i]);


return 0;
}
```

Program di atas bertujuan untuk membuat sebuah array dengan ukuran yang dinamis yang artinya ukuran array dapat diubah ketika program dijalankan. Ilustrasi berikut menunjukkan array awalnya memiliki panjang 5 lalu memanfaatkan proses *looping*, setiap indeks array akan diisi dengan nilai `i`. Kemudian array dialokasikan ulang dengan mengubah panjang array sehingga panjang array saat ini adalah 10.



Selain mengubah ukuran data, menggunakan fungsi `realloc()` juga turut mengubah alamat memori pada *heap*.

- `free()`

Fungsi ini digunakan untuk menghapus alokasi memori yang sebelumnya telah dibuat oleh fungsi `malloc()`, `calloc()` ataupun `realloc()`. Berikut adalah syntax dari fungsi `free()` :

`free(*allocated_ptr)`

Keterangan :

`*allocated_ptr` adalah variabel pointer yang telah dialokasi sebelumnya

Perhatikan code berikut :

```
#include "stdio.h"
#include "stdlib.h"

int main(){
int N;
printf("Panjang array = ");
scanf("%d",&N);
```

```
int *arr;
arr = (int *)malloc(N*sizeof(int));

for(int i=0; i<N; i++){
arr[i] = i;
}
printf("Ukuran array = %d (dalam byte)\n",N*sizeof(*arr));
printf("Panjang array = %d \n",N*sizeof(*arr)/sizeof(*arr));
printf("Alamat = %x\n",arr);

free(arr);
printf("Panjang array setelah fungsi free() = %d, alamat =
%x",sizeof(*arr) / sizeof(arr[0]),arr);
return 0;
}
```

```
Panjang array = 7
Ukuran array = 28 (dalam byte)
Panjang array = 7
Alamat = 9d6d00
Panjang array setelah fungsi free() = 1, alamat = 9d6d00
--------------------------------
Process exited after 0.9475 seconds with return value 0
Press any key to continue . . .
```

Program di atas mengilustrasikan penggunaan fungsi `free()`. Array yang telah dialokasikan menggunakan fungsi `malloc()` akan dihapus pada akhir program sehingga isi dari data pada variabel `arr` turut dihapus. Alasan panjang array setelah dihapus menunjukkan angka 1 karena variabel arr masih tersedia pada heap di alamat 9d6d00 dan jika terdapat variabel yang terdaftar pada memori maka secara default lokasi tersebut akan memuat nilai tertentu (tergantung aplikasi compiler). Hal ini seperti ketika kita mendeklarasi sebuah variabel bertype integr (yang tidak di-assign dengan sebuah nilai) lalu kita lakukan print terhadap variabel tersebut maka akan muncul sebuah nilai tertentu.

- **Macro**

Bahasa pemrograman C merupakan salah satu bahasa tingkat tinggi yang artinya adalah pengguna dapat memberikan instruksi kepada komputer dengan menggunakan bahasa yang dapat dipahami oleh manusia. Komputer yang ada saat ini hanya dapat memahami bahasa mesin yang terdiri dari ratusan atau bahkan jutaan deret bilangan binary. Jadi, ketika komputer mengeksekusi program yang telah ditulis, aplikasi compiler (dev-c++) akan menyusun (compile) setiap langkah instruksi lalu urutan perintah-perintah tersebut diterjemahkan ke dalam bahasa mesin. Seluruh proses yang telah disebutkan akan berjalan pada file program utama.

Bahasa pemrograman C dibuat dalam konsep modul atau dengan kata lain, library yang disediakan terdapat pada file lain. Macro berfungsi untuk memasukkan seluruh baris kode

dari suatu library sebelum kode program utama diterjemahkan ke dalam bahas mesin. Jika tidak menggunakan macro dan ingin menggunakan fungsi dari suatu library maka seluruh baris kode dari fungsi tersebut wajib ditulis dalam program utama. Sebagai contoh, kita ingin menggunakan fungsi `printf()` dari library `stdio.h` pada program utama kita. Jika kita tidak menambahkan `#include <stdio.h>` pada *header* maka kita perlu memindahkan kode dari fungsi `printf()` pada file `stdio.h` ke dalam file program utama. Tentu saja cara ini tidak efisien karena program kita akan menjadi lebih sulit untuk dibaca. `#include <stdio.h>` merupakan salah satu macro yang memberikan instruksi kepada preprocessor untuk memasukkan seluruh kode dari library `stdio.h`. Terdapat beberapa macro yang digunakan dalam pemrograman C, diantaranya adalah :

1. `#define`, digunakan untuk mendefinisikan konstanta dan fungsi
2. `#include`, digunakan untuk menambahkan kode program ke dalam program utama
3. `#undef`, digunakan untuk menghapus macro yang telah didefinisikan
4. `#ifdef`, digunakan untuk mengecek apakah macro sudah didefiniskan
5. `#ifndef`, digunakan untuk mengecek apakah macro belum didefinisikan
6. `#if`, digunakan untuk membuat kondisi if
7. `#else`, digunakan untuk membuat alternatif kondisi if
8. `#elif`, digunakan untuk membuat kondisi else-if
9. `#endif`, digunakan untuk mengakhiri block if
10. `#error`, digunakan untuk mengecek pesan error

Berikut adalah beberapa penggunaan macro :

Define

```
#include "stdio.h"
#define PHI 3.14

int main(){
printf("%f",PHI);
return 0;
}
```

```
#include "stdio.h"
#define sum(a,b) (a+b)

int main(){
printf("%d",sum(2,2));
return 0;
}
```

Predefined

Macro-macro yang sudah didefinisikan di komputer. Beberapa predefined macro yang dapat digunakan pada pemrograman C diantaranya adalah :

1. `__DATE__` , Berisi tanggal saat ini
2. `__TIME__` , Berisi waktu saat ini
3. `__FILE__` , Berisi nama file dari kode program
4. `__LINE__` , Berisi informasi baris program
5. `__STDC__`, Berisi 1 jika program di-compile dengan standar ANSI

```
#include <stdio.h>

int main() {

   printf("File : %s\n", __FILE__ );
   printf("Date : %s\n", __DATE__ );
   printf("Time : %s\n", __TIME__ );
   printf("Line : %d\n", __LINE__ );
   printf("ANSI : %d\n", __STDC__ );

   return 0;
}
```

Direktif kondisi

```
#include <stdio.h>

#ifndef DEBUG
#define DEBUG true
#endif

int main(){
     #if defined(DEBUG)
     printf("Debugging mode is on\n");
     #else
     printf("Debugging mode is off\n");
     #endif
     return 0;
}
```

- **Pointer to Function**

Selain digunakan untuk menyimpan nilai, pointer juga dapat digunakan untuk menyimpan sebuah fungsi. Variabel pointer akan merujuk kepada alamat dari suatu fungsi. Melalui variabel pointer tersebut kita dapat memanggil fungsi dan kita juga bisa meneruskan sebuah fungsi sebagai argument ke fungsi yang lain. Berikut adalah syntax dari *pointer to function* :

```
type_data *pointer_var(type_data_argument)
```

Keterangan :

`type_data` adalah tipe data dari nilai yang dikembalikan oleh fungsi

`*pointer_var` adalah variabel berjenis pointer untuk menampung alamat fungsi

`type_data_argument` adalah tipe data dari argument yang akan dikirimkan kepada fungsi

Berikut adalah contoh program dari *pointer to function* :

Pointer to function	Tanpa pointer to function

`#include "stdio.h"`
`int sum(int a, int b);`
 	
`int main(){`
`int x=10,y=3;`
`int(*s)(int,int); //deklarasi pointer yang memuat fungsi`
`s = sum; //memilih fungsi untuk pointer`
 	
`printf("%d",(*s)(x,y));`
`return 0;`
`}`
 	
`int sum(int a, int b){`
`return a + b;`
`}`
	`#include "stdio.h"`
	`int sum(int a, int b);`
	
	`int main(){`
	`int x=10,y=3;`
	`printf("%d",sum(x,y));`
	`return 0;`
	`}`
	
	`int sum(int a, int b){`
	`return a + b;`
	`}`

Pointer to function	Tanpa pointer to function

`#include "stdio.h"`
`int calc(int x);`
`int mul(int a, int b);`
 	
`int main(){`
`int x = 10, y = 3;`
`int(*m)(int,int);`
`m = mul;`
 	
`int(*c)(int);`
`c = calc;`
 	
`printf("%d",(*c)((*m)(x,y)));`
`return 0;`
`}`
 	
`int calc(int x){`
`return x * 2;`
`}`
 	
`int mul(int a, int b){`
`return a + b;`
`}`
	`#include "stdio.h"`
	
	`int calc(int);`
	`int mul(int, int);`
	
	`int main(){`
	`int a = 10, b = 3;`
	`printf("%d",calc(mul(a,b)));`
	`return 0;`
	`}`
	
	`int calc(int x){`
	`return x * 2;`
	`}`
	
	`int mul(int a, int b){`
	`return a + b;`
	`}`