

Nama : Lorenzo Liunardo

NIM : 211401061

## Pengenalan Pointer dalam Bahasa C/C++

Pointer adalah variabel yang menyimpan alamat dari variabel lainnya.

Bagaimana cara kita mendeklarasikan pointer dalam c/c++?

Misal:

```
int a; // kita mendeklarasikan variabel a bertipe integer
```

```
int *p; // dengan memberikan lambang asterisk sebelum nama variabel maka kita sudah membuat suatu variabel pointer yaitu *p
```

```
p = &a; // pendeklarasian seperti ini akan membuat alamat dari variabel integer a disimpan ke dalam variabel pointer p sehingga variabel p sekarang menyimpan alamat dari variabel a.
```

Kemudian kita misalkan variabel a memiliki alamat "204" dan variabel p memiliki alamat "64", maka:

Jika kita mencetak p akan menghasilkan output "204" (dimana ini merupakan alamat dari variabel a).

Jika kita mencetak &a akan menghasilkan output "204" pula karena &a berarti kita ingin memunculkan alamat dari variabel a.

Lalu jika kita mencetak &p maka akan memunculkan output berupa alamat dari variabel pointer p itu sendiri yaitu "64".

Kemudian anggap int a bernilai 5, bagaimana cara kita membuat variabel p memunculkan angka 5 tsb? Cara nya adalah dengan mencetak \*p maka akan memunculkan nilai variabel yang dimiliki oleh variabel a.

Konsep ini dinamakan referencing dimana kita menyimpan alamat variabel a ke p dan kita bisa mendapatkan nilai dari variabel a dengan mencetak \*p, kita juga bisa memodifikasi nilai dari a dengan mendeklarasikan \*p = (angka yang kita inginkan misalnya 8), maka ketika kita mencetak a, akan menghasilkan output berupa angka 8 dan bukan lagi angka 5 karena kita sudah mengubah nilai dari a melalui p.

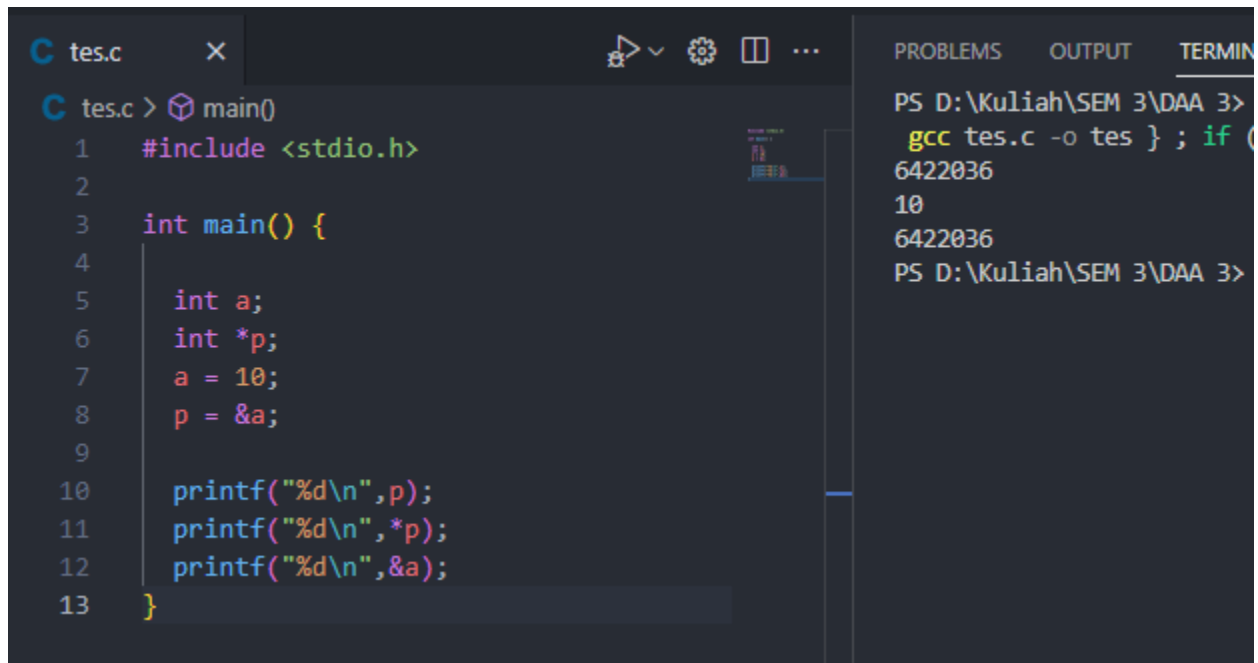
Kesimpulannya:

p → alamat

\*p → nilai dari alamat yang disimpan

## Bekerja dengan Pointer

Misalkan kita mempunyai kode dalam bahasa C sbb:

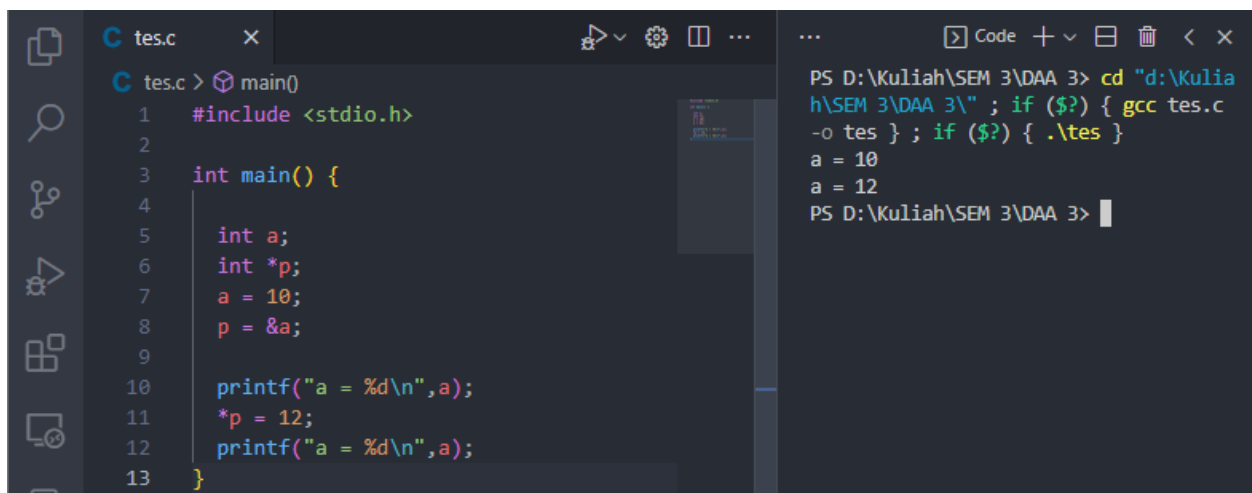


```
tes.c tes.c > main()
1  #include <stdio.h>
2
3  int main() {
4
5      int a;
6      int *p;
7      a = 10;
8      p = &a;
9
10     printf("%d\n",p);
11     printf("%d\n",*p);
12     printf("%d\n",&a);
13 }
```

PROBLEMS OUTPUT TERMINAL

```
PS D:\Kuliah\SEM 3\DAA 3> gcc tes.c -o tes } ; if (
6422036
10
6422036
PS D:\Kuliah\SEM 3\DAA 3>
```

Lalu kita mempunyai kode 2 sbb:

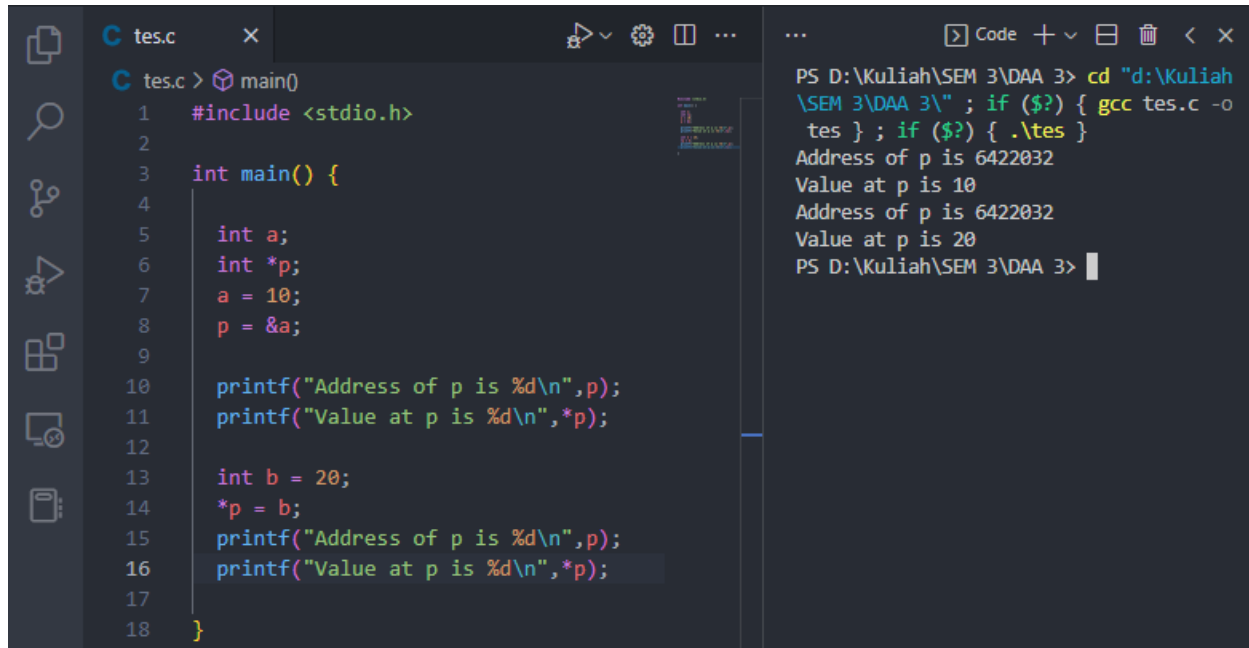


```
tes.c tes.c > main()
1  #include <stdio.h>
2
3  int main() {
4
5      int a;
6      int *p;
7      a = 10;
8      p = &a;
9
10     printf("a = %d\n",a);
11     *p = 12;
12     printf("a = %d\n",a);
13 }
```

Code + - < x

```
PS D:\Kuliah\SEM 3\DAA 3> cd "d:\Kuliah\SEM 3\DAA 3" ; if ($?) { gcc tes.c
-o tes } ; if ($?) { .\tes }
a = 10
a = 12
PS D:\Kuliah\SEM 3\DAA 3>
```

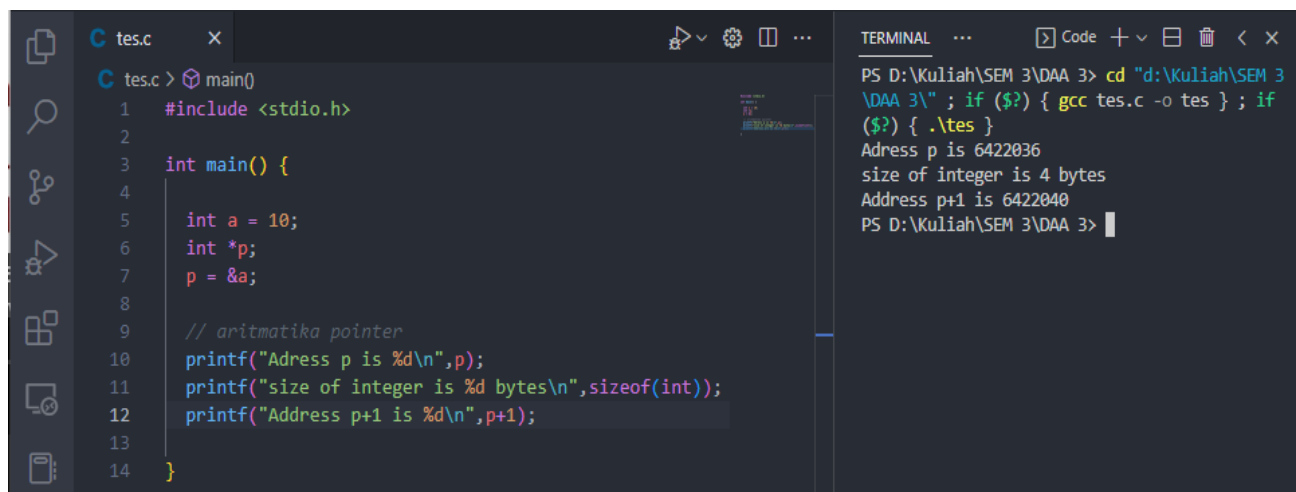
Lalu kita mempunyai kode 3 sbb:



```
tes.c tes.c > main()
1  #include <stdio.h>
2
3  int main() {
4
5      int a;
6      int *p;
7      a = 10;
8      p = &a;
9
10     printf("Address of p is %d\n",p);
11     printf("Value at p is %d\n",*p);
12
13     int b = 20;
14     *p = b;
15     printf("Address of p is %d\n",p);
16     printf("Value at p is %d\n",*p);
17
18 }
```

```
PS D:\Kuliah\SEM 3\DAA 3> cd "d:\Kuliah\SEM 3\DAA 3\" ; if ($?) { gcc tes.c -o tes } ; if ($?) { .\tes }
Address of p is 6422032
Value at p is 10
Address of p is 6422032
Value at p is 20
PS D:\Kuliah\SEM 3\DAA 3>
```

Lalu kita mempunyai kode 4 sbb:

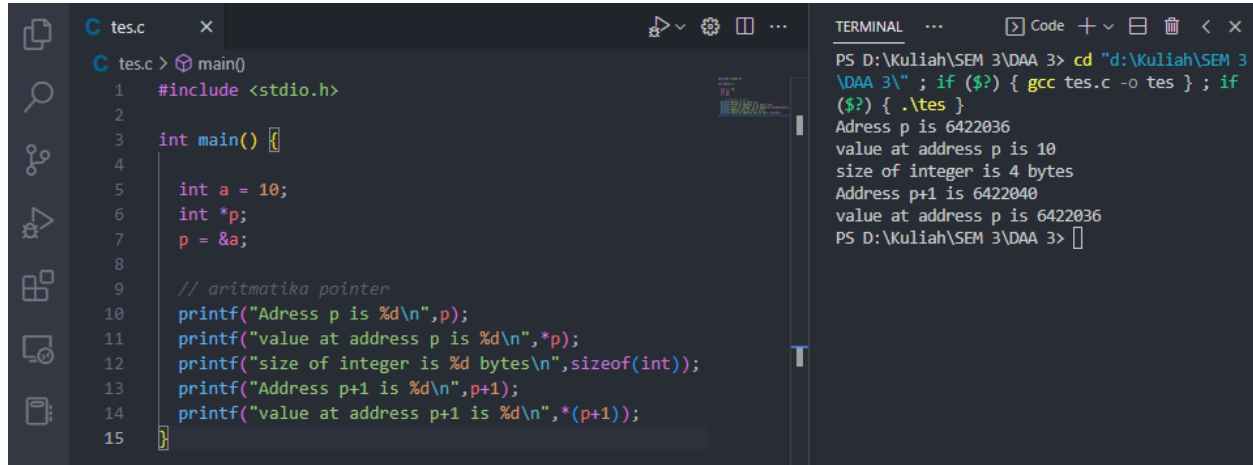


```
tes.c tes.c > main()
1  #include <stdio.h>
2
3  int main() {
4
5      int a = 10;
6      int *p;
7      p = &a;
8
9      // aritmatika pointer
10     printf("Adress p is %d\n",p);
11     printf("size of integer is %d bytes\n",sizeof(int));
12     printf("Address p+1 is %d\n",p+1);
13
14 }
```

```
PS D:\Kuliah\SEM 3\DAA 3> cd "d:\Kuliah\SEM 3\DAA 3\" ; if ($?) { gcc tes.c -o tes } ; if ($?) { .\tes }
Adress p is 6422036
size of integer is 4 bytes
Address p+1 is 6422040
PS D:\Kuliah\SEM 3\DAA 3>
```

Kita bisa menambahkan nilai dari alamat pointer, jika kita menambahkan p menjadi p+1, kita bisa lihat bahwa nilai alamat p berubah dari 6422036 menjadi 6422040. Kenapa bertambah 4? Karena variabel p kita bertipe integer, dan integer mempunyai ukuran 4 bytes. Jadi ketika kita menambahkan 1 maka dia harus berpindah sebesar 4 bytes untuk bisa ke integer berikutnya.

Lalu kita mempunyai kode 5 sbb:

The image shows a code editor with a C program named 'tes.c' and a terminal window. The C program defines a pointer 'p' to an integer 'a' and prints the address of 'p', the value at that address, the size of an integer, the address of 'p+1', and the value at that address. The terminal output shows the program's execution, where 'p' is at address 6422036 and contains the value 10. 'p+1' is at address 6422040 and contains a garbage value (6422036).

```
tes.c > main()
1  #include <stdio.h>
2
3  int main() {
4
5      int a = 10;
6      int *p;
7      p = &a;
8
9      // aritmatika pointer
10     printf("Adress p is %d\n",p);
11     printf("value at address p is %d\n",*p);
12     printf("size of integer is %d bytes\n",sizeof(int));
13     printf("Address p+1 is %d\n",p+1);
14     printf("value at address p+1 is %d\n",*(p+1));
15 }
```

```
PS D:\Kuliah\SEM 3\DAA 3> cd "d:\Kuliah\SEM 3
\DAA 3\" ; if ($?) { gcc tes.c -o tes } ; if
($?) { .\tes }
Adress p is 6422036
value at address p is 10
size of integer is 4 bytes
Address p+1 is 6422040
value at address p is 6422036
PS D:\Kuliah\SEM 3\DAA 3>
```

Ketika kita memunculkan nilai dari alamat p+1 kita mendapatkan nilai integer yang kita tidak ketahui. Ini merupakan nilai garbage karena kita sebenarnya tidak mempunyai sebuah nilai integer dialokasikan ke memori tersebut, dan ini menjadi salah satu hal yang berbahaya dari menggunakan aritmatika pointer/manipulasi pointer pada C karena kita bisa mencapai alamat mana saja dan pada satu titik operasi seperti ini dapat memberikan hal yang tidak kita inginkan di program kita.

## Type-tipe pointer, void pointer, aritmatika pointer

Dalam mendeklarasikan pointer kita perlu menyamakan tipe data pointer kita dengan tipe data variabel yang ingin disimpan alamatnya.

Misal:

Int \* → int

Char \* → char

Apa tujuan dari dilakukannya hal ini? Kenapa kita mempunyai 1 saja tipe pointer utk semua tipe data? Hal ini karena pointer memiliki konsep yang dinamakan dereferensi dimana dengan pointer kita tidak hanya menyimpan alamat dari data tsb namun kita juga bisa mengakses dan memodifikasi data yang ada pada alamat tersebut.

Misalkan kita mempunyai program dlm C sbb:

```
#include <stdio.h>
```

```
int main() {
```

```
    int a = 1025;
```

```

int *p;
p = &a;
printf("size of integer is %d bytes\n", sizeof(int));
printf("Address %d, value = %d\n", p,*p);

char *p0;
p0 = (char*)p;
printf("size of integer is %d bytes\n", sizeof(char));
printf("Address %d, value = %d\n", p0,*p0);
}

```

Akan menghasilkan output sbb:

Size of integer is 4 bytes

Address = 2948664, value = 1025

Size of char is 1 bytes

Address = 2948664, value = 1

Kemudian muncul pertanyaan kenapa nilai dari p0 adalah 1 bukan 1025 meskipun dia sudah memiliki alamat yang sama dengan p.

Untuk itu mari kita lihat kembali jika kita menuliskan 1025 ke dalam angka biner menggunakan 32 bits

1025 = 00000000 00000000 00000100 00000001

Karena p0 merupakan char dan hanya bisa menyimpan 1 byte maka ketika kita memanggil nilai dari p0 dia hanya akan menunjukkan nilai dari alamat paling kanan dari 1025 yakni 00000001 yang ketika kita ubah ke nilai sebenarnya adalah 1.

Lalu kita mempunyai kode program sbb:

```
#include <stdio.h>
```

```
int main() {
```

```

    int a = 1025;
    int *p;
    p = &a;
    printf("size of integer is %d bytes\n", sizeof(int));
    printf("Address %d, value = %d\n", p,*p);
    printf("Address %d, value = %d\n", p+1,*(p+1));

```

```

    char *p0;
    p0 = (char*)p;
    printf("size of integer is %d bytes\n", sizeof(char));
    printf("Address %d, value = %d\n", p0,*p0);
    printf("Address %d, value = %d\n", p0+1,*(p0+1));

```

```
}
```

Akan menghasilkan output sbb:

Size of integer is 4 bytes

Address = 4456036, value = 1025

Address = 4456040, value = -858993460

Size of char is 1 bytes

Address = 4456036, value = 1

Address = 4456037, value = 4

Sekarang ketika kita mencetak alamat dari p0+1 akan memunculkan 4456037 yang mana sesuai dengan pertambahan 1 byte dari alamat p0 yaitu 4456037, namun nilai dari p0+1 berubah menjadi 4. Kenapa hal ini bisa terjadi?

1025 = 00000000 00000000 00000100 00000001

Dari hasil angka biner 1025 yang telah kita punya ketika kita menambah 1 ke alamat p0 maka sekarang nilai dari p0 + 1 akan memakai 00000100 yang mana jika kita terjemahkan ke desimal akan menghasilkan angka 4.

Setelah itu kita mempunyai kode program sbb:

```
#include <stdio.h>
```

```
int main() {
```

```
    int a = 1025;
```

```
    int *p;
```

```
    p = &a;
```

```
    printf("size of integer is %d bytes\n", sizeof(int));
```

```
    printf("Address %d, value = %d\n", p,*p);
```

```
    // void pointer - generic pointer
```

```
    void *p0;
```

```
    p0 = p;
```

```
    printf("Address = %d",p0 );
```

```
}
```

Akan menghasilkan output:

Size of integer is 4 bytes

Address = 3341104, value = 1025

Address = 3341104

Disini kita dapat mendeklarasikan sebuah pointer tanpa tipe data tertentu dengan menggunakan void. Keuntungannya kita tidak perlu menyamakan tipe data dengan variabel utk

dapat menyimpan alamat dari variabel tersebut, namun kelemahannya kita tidak dapat mencetak nilai dari alamat yg dituju karena void hanya menyimpan alamat saja dan bukan nilai. Kita juga tidak dapat melakukan operasi aritmatika seperti misalnya `p0+1`. Program akan error jika kita memaksakan hal tersebut.

## Pointer to pointer dalam c/c++

Kita bisa membuat sebuah pointer yang menunjuk ke pointer lainnya.

Misal kita mempunyai kode sebagai berikut:

```
#include <stdio.h>
int main() {

    int x = 5;
    int *p = &x;
    *p = 6
    int **q = &p;
    int ***r = &q;

    printf("%d\n", *p);
    printf("%d\n", *q);
    printf("%d\n", **q);
    printf("%d\n", **r);
    printf("%d\n", ***r);
    ***r = 10;
    printf("x = %d\n", x);
}
```

Akan menghasilkan output berupa:

```
6
4193620
6
4193620
6
X = 10
```

Kita bisa lihat jika kita mendeklarasikan pointer `**q` yang merujuk ke pointer `*p` maka ketika kita memanggil `*p` akan menghasilkan alamat dari `x` yang sama saja ketika kita memanggil `p`, kemudian ketika kita memanggil `**p` maka akan menghasilkan nilai dari `x` yaitu 5 sama seperti jika kita memanggil `*p` begitu pula akan sama terhadap pointer `***r` yang merujuk ke `**q`. Kita juga bisa merubah nilai melalui pointer yang merujuk ke pointer lainnya. Pada program kita

mendeklarasikan `***r = 10` hal ini akan mengubah nilai x menjadi 10 karna pointer r merujuk ke pointer q merujuk ke pointer p dan merujuk ke x.

## Pointer as function arguments-call by reference

Misalkan kita mempunyai program sbb:

```
#include <stdio.h>
```

```
Void increment(int *p) {  
    *p = (*p) + 1  
}
```

```
int main() {  
  
    int a;  
    a = 10;  
    increment(&a);  
    printf("a = %d\n",a);  
  
}
```

Akan menghasilkan output berupa `a = 11`

Hal ini terjadi karena ketika kita memanggil fungsi `increment(&a)`, kita mempassing alamat dari variabel `a` ke pointer `*p` dan kemudian operasi `*p = (*p) + 1` akan menambahkan nilai dari alamat variabel yang disimpan dengan 1 sehingga nilai `a` yang semula adalah 10 telah bertambah menjadi 11. Cara ini biasa dinamakan `call by reference`

## Pointer dan arrays

Misalkan kita mempunyai array `int A[5]`

`A[0] = 2` dengan alamat 200

`A[1] = 2` dengan alamat 204

`A[2] = 2` dengan alamat 208

`A[3] = 2` dengan alamat 212

`A[4] = 2` dengan alamat 216

Jika kita mempunyai code sbb:

```
Int A[5]
```

```
Int *p
```



```

P = &A[0]
Print p akan menghasilkan nilai alamat elemen pertama array yaitu 200
Print *p akan menghasilkan nilai elemen pertama array yaitu 2
Atau jika
Int A[5]
Int *p
P = A
Print A akan menghasilkan nilai alamat elemen pertama array yaitu 200
Print *A akan menghasilkan nilai elemen pertama array yaitu 2
Print A + 1 // 204
Print *(A+1) // 4

```

Misalkan kita mempunyai kode sbb:

```
#include <stdio.h>
```

```

int main() {

    int a[] = {2,4,5,8,1};
    printf("%d\n",a); // akan mencetak alamat dari elemen array yg pertama
    printf("%d\n",&a[0]); // akan mencetak alamat dari elemen array yg pertama
    printf("%d\n",a[0]); // akan mencetak nilai dari elemen array yg pertama
    printf("%d\n",*a); // akan mencetak nilai dari elemen array yg pertama
}

```

Kesimpulannya adalah:

Element pada index ke i-

Memunculkan alamat - &[i] atau (A + i)

Memunculkan nilai - A[i] atau \*(A + i)

## Arrays sebagai function arguments

Sebagai contoh disini kita akan membuat program untuk menjumlahkan semua elemen pada array dan mengeluarkan hasilnya sebagai output

```
#include <stdio.h>
```

```

Int sumOfElements(int *a, int size) { // baik int *a atau int a[] akan ttp sama
    Int i, sum = 0;

    for(i = 0; i < size; i++) {
        sum += a[i]; // a[i] sama dengan *(a + i)
    }
}

```

```

        return sum;
    }

int main() {

    int a[] = {1,2,3,4,5};
    Int size = sizeof(a) / sizeof(a[0]);
    Int total = sumOfElements(a, size);
    printf("Sum of elements = %d\n",total);

}

```

Maka pada output program akan menghasilkan keluaran yakni 20 yang merupakan hasil penjumlahan dari semua elemen array. Perlu diingat bahwa karena array ini di passing by reference maka kita perlu membuat 1 variabel yg menyimpan ukuran dari array kita, jika kita tidak melakukan hal tersebut maka yang akan di passing ke fungsi penjumlahan array hanyalah elemen array yang pertama.

## Character arrays and pointers

Beberapa karakter array dan pointer adalah sbb:

- Jika kita ingin menyimpan sebuah string dalam sebuah array maka kita harus memenuhi syarat ukuran array  $\geq$  jumlah karakter pada string + 1
- Array dan pointer berbeda tipe namun digunakan dalam cara yang serupa
- Array selalu di passing ke sebuah function by reference. Artinya adalah selalu nilai alamat array lah yang dipassing dan bukan value dari array itu sendiri

Misalkan kita ingin membuat program mencetak "hello"

```

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

void print(char *c)
{
    while(*c != '\0');
    {
        printf("%c",*c);
        c++;
    }
    printf("\n");
}

```

```
}
```

```
int main()
{
    char c[20] = "hello";
    print(c);
}
```

Program ini akan memunculkan output berupa hello, bisa kita lihat pada looping while kita mengecek apakah `*c != '\0'`. Apa maksud dari `'\0'` ini? Dari penjelasan karakteristik di atas kita sudah tau bahwa jika kita ingin menyimpan sebuah string dalam sebuah array maka kita harus memenuhi syarat ukuran array  $\geq$  jumlah karakter pada string + 1, maka ketika kita menyimpan hello di `char c[20]` maka program secara otomatis memberikan nilai null `'\0'` setelah akhir kata hello agar program tau kata yang kita berikan itu ada sampai index ke berapa.

## Pointer dan array multidimensional

Misalkan kita punya sebuah array 2d

`int b[2][3]`, maka:

`b[0]`

`b[1]`

Kedua nya ini merupakan array 1 dimensi yang mempunyai 3 integer

Kemudian jika kita mendeklarasikan

`int *p = b`, maka ini akan error karena `b` akan mengembalikan sebuah pointer ke array 1 dimensi yang mempunyai 3 integer dan bukan sebuah pointer ke integer

Maka cara yang benar adalah

`int (*p)[3] = b`

Misalkan:

`B[0][0] = 2` dgn alamat 400

`B[0][1] = 3` dgn alamat 404

`B[0][2] = 6` dgn alamat 408

`B[1][0] = 4` dgn alamat 412

`B[1][1] = 5` dgn alamat 416

`B[1][2] = 8` dgn alamat 420

Maka jika:

Print `b` or `&b[0]` // mencetak 400

Print `*b` or `b[0]` or `&b[0][0]` // mencetak 400

Print `b+1` or `&b[1]` // mencetak 412

Print `*(b+1)` or `b[1]` or `&b[1][0]` // mencetak 412  
Print `*(b+1)+2` or `b[1]+2` or `&b[1][2]` // mencetak 420  
Print `*(b+1) = *b[0][1]` // mencetak 3

Kesimpulannya:

Untuk array 2 dimensi

$B[i][j] = *(b[i] + j) = (*(b+i) + j)$

Kemudian kita punya array 3 dimensi:

`int c[3][2][2]`, dgn ket:

`C[0][0][0] = 2` dgn alamat 800

`C[0][0][1] = 5` dgn alamat 804

`C[0][1][0] = 7` dgn alamat 808

`C[0][1][1] = 9` dgn alamat 812

`C[1][0][0] = 3` dgn alamat 816

`C[1][0][1] = 4` dgn alamat 820

`C[1][1][0] = 6` dgn alamat 824

`C[1][1][1] = 1` dgn alamat 828

`C[2][0][0] = 0` dgn alamat 832

`C[2][0][1] = 8` dgn alamat 836

`C[2][1][0] = 11` dgn alamat 840

`C[2][1][1] = 13` dgn alamat 844

Kita dapat membuat pointernya sbb:

`Int (*p)[2][2] = c`

Maka:

$c[i][j][k] = *(c[i][j] + k) = (*(c[i] + j) + k) = (*(c+i+j)+k)$

Print `c` // 800

Print `*c` or `c[0]` or `&c[0][0]` // 800

Print `*(c[0][1] + 1)` or `c[0][1][1]` // 9

Print `*(c[1] + 1)` or `c[1][1]` or `&c[1][1][0]` // 824

## Pointer dan dynamic memory

Misalkan kita mempunyai program dengan beberapa fungsi, maka ketika kita menjalankan program tersebut, program tersebut akan dieksekusi di dalam stack. Ketika suatu fungsi memanggil fungsi lainnya maka fungsi tersebut harus menunggu sampai fungsi yang ia panggil selesai terlebih dahulu sebelum dapat menjalankan sisa fungsi yang ia miliki. Ketika suatu fungsi sudah selesai dijalankan, maka ia akan dihapus dari stack karena sudah tidak ada lagi yang perlu dijalankan dan agar tidak memakan memori. Stack juga mempunyai fixed size memori, ketika kita menjalankan suatu program, OS akan menyediakan sekian memori pada

stack (kita misalkan 1 MB) ketika program terus menerus memanggil fungsi melebihi kecepatan eksekusi dan semakin banyak fungsi yg menunggu di dalam stack maka ketika memori yg di set tadi tidak dapat lagi menampung fungsi yang ada maka program kita akan crash atau sering disebut stack overflow. Oleh karena keterbatasan ini, maka kita memiliki yang namanya heap. Tidak seperti stack, ukurannya dapat menyesuaikan ketika aplikasi dijalankan dan tidak ada batasan alokasi memori. Program dapat sepenuhnya mengontrol berapa banyak memori di heap sampai berapa waktu yg dibutuhkan untuk menyimpan data di dalam memori sepanjang aplikasi dijalankan dan dapat terus bertambah selagi kita tidak kehabisan memori.

Ketika kita ingin menggunakan dynamic memory dalam:

C, kita menggunakan fungsi sbb:

1. Malloc
2. Calloc
3. Realloc
4. free

C++, kita menggunakan operator

- New
- delete

Contoh program dalam C:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int a; // akan berada di stack
    int p;

    p = (int*)malloc(sizeof(int)); // mengalokasikan 4 bytes di heap
    *p = 10;

    free(p); // membebaskan memori p di heap

    p = (int*)malloc(20*sizeof(int)); // mengalokasikan array 20 elemen ke heap
}
```

Dalam c++ akan berbentuk seperti ini:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int a; // akan berada di stack
    int p;

    p = new int;
    *p = 10;
```

```

        delete p;
        p = new int[20];
        delete[] p;
    }

```

## malloc, calloc, realloc, free

malloc = void \* malloc(size\_t size)

Ex: void \*p = malloc(sizeof(int));

calloc = void \* calloc(size\_t num, size\_t size)

Ex: int \*p = (int\*)calloc(3,sizeof(int))

realloc = void \* realloc (void \* ptr, size\_t size)

Berikut merupakan contoh kodenya:

```

#include <stdio.h>
#include <stdlib.h>
int main() {
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);

    int *a = (int*)malloc(n*sizeof(int)); // dynamically allocated array

    for(int i = 0; i < n; i++) {
        a[i] = i + 1;
    }

    for(int i = 0; i < n; i++) {
        printf("%d ",a[i]);
    }
}

```

Program ini akan membuat kita menentukan besar dari array kemudian akan memasukan nilai pada setiap elemen array yang akan bertambah 1 di setiap iterasinya. Jika kita memasukkan besar array adalah 10 maka output yang akan dihasilkan adalah 1 2 3 4 5 6 7 8 9 10.

Jika pada program pertama kita ingin menggunakan calloc dan bukan malloc maka kode program akan seperti ini:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);

    int *a = (int*)calloc(n,sizeof(int)); // dynamically allocated array

    for(int i = 0; i < n; i++) {
        a[i] = i + 1;
    }

    for(int i = 0; i < n; i++) {
        printf("%d ",a[i]);
    }
}
```

Perbedaan antara calloc dan malloc adalah jika kita tidak membuat for loop utk memasukkan nilai pada setiap elemen array atau setelah for loop memasukkan nilai ke setiap elemen array kita kemudian membebaskan memori tersebut dan mencetak setiap elemen pada array, maka pada malloc akan diisi dengan nilai sampah sedangkan pada calloc akan diisi dengan nilai 0

Berikut adalah contoh kode jika kita ingin menggunakan realloc:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n;
    printf("Enter size of array\n");
    scanf("%d",&n);

    int *a = (int*)calloc(n,sizeof(int)); // dynamically allocated array

    for(int i = 0; i < n; i++) {
        a[i] = i + 1;
    }

    int *b = (int*)realloc(a, 2*n*sizeof(int));
    printf("prev block address = %d, new address = %d\n",a,b);
}
```

```

        for(int i = 0; i < n; i++) {
            printf("%d ",a[i]);
        }
    }
}

```

Jika kita memasukkan besar array adalah 5 maka int \*b akan mengubah ukuran array menjadi sepuluh dan alamat dari array akan tetap sama namun utk karena penambahan ukuran array yang baru tidak diinisialisasi nilai, maka setengah dari nilai array akan bernilai nilai sampah.

Kita juga bisa mengurangi ukuran awal dari sebuah array menggunakan realloc

## Pointer as function returns

Berikut adalah contoh program jika kita ingin menggunakan pointer sbg function returns

```

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

int add(int* a, int* b) { // called function
    // a and b are pointer to integers local to add
    printf("Address of a in add = %d\n",&a); // 3537372
    printf("value in a of add (address of a in main) = %d\n",a); // 3537612
    printf("value at address stored in a of add = %d\n",*a); // 2
    int c = (*a) + (*b);
    return c;
}

int main() { // calling function
    int a = 2, b = 4;
    printf("Address of a in main = %d\n",&a); // 3537612
    // call by reference
    int c = add(&a,&b); // a and b are integers local to main
    printf("sum = %d\n",c); // 6
}

```

Kita dapat memodifikasi kode diatas sbb:

```

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

int* add(int* a, int* b) { // called function - return pointer to integer

```



```

        int c = (*a) + (*b);
        return &c;
    }

int main() { // calling function
    int a = 2, b = 4;
    int* ptr = add(&a,&b); // a and b are integers local to main
    printf("sum = %d\n",*ptr);
}

```

Apa yang kita ubah disini adalah kita merubah function add untuk mengembalikan sebuah pointer to integer

Kemudian kita punya kode sbb:

```

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

void printHelloWorld() {
    printf("Hello World\n");
}

int* add(int* a, int* b) {
    int c = (*a) + (*b);
    return &c;
}

int main() { // calling function
    int a = 2, b = 4;
    int* ptr = add(&a,&b);
    printHelloWorld();
    printf("sum = %d\n",*ptr);
}

```

Ketika kita menjalankan program diatas akan muncul output hello world dan 434211. Kenapa bukan angka 6 yang muncul seperti yang kita inginkan. Ketika kita menjalankan program maka fungsi akan menempati memori di stack dimulai dari yang paling bawah adalah main function. Kemudian main function memanggil fungsi add. Fungsi add akan berada di atas main function dan pointer ptr di main function akan menunjuk ke alamat memori variabel c di fungsi add. Ketika fungsi add selesai menjalankan tugasnya maka fungsi tersebut akan dibebaskan dari memori di dalam stack. Kasusnya disini adalah pointer ptr masih menunjuk ke alamat yang tadi kita misalkan 144. Lalu main function memanggil fungsi printHelloWorld, fungsi ini kemudian menempati posisi alamat yang telah ditinggalkan oleh fungsi add tadi. Jadi fungsi printHelloWorld mengoverwrite isi dari alamat yang ditunjuk oleh ptr tadi sehingga hasil yang

kita inginkan pun tidak terpenuhi. Kesimpulannya adalah itu aman jika kita mempassing alamat dari sebuah local variable secara bottom to top tapi tidak aman jika itu dipassing secara top to bottom.

Maka untuk menyelesaikan permasalahan dari kode diatas kita dapat menggunakan malloc utk menyimpan nya di heap berikut adalah kodenya:

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

void printHelloWorld() {
    printf("Hello World\n");
}

int* add(int* a, int* b) {
    int* c = (int*)malloc(sizeof(int));
    *c = (*a) + (*b);
    return c;
}

int main() { // calling function
    int a = 2, b = 4;
    int* ptr = add(&a,&b);
    printHelloWorld();
    printf("sum = %d\n",*ptr);
}
```

## Function pointers

Function pointer adalah pointer yang menyimpan alamat dari function yang artinya adalah pointer ini akan menyimpan alamat pertama/entry point dari sebuah blok memori yang mengandung semua instruksi di dalam sebuah function.

Berikut adalah contoh kode programnya:

```
#include <stdio.h>

int add(int a, int b) {
    return a+b;
}

int main() {
    int c;
```

```

int (*p)(int,int);
p = add; // function name will return us pointer
c = p(2,3); // de-referencing and executing the function
printf("%d",c); // output = 5
}

```

## Function pointers and callbacks

Function pointers dapat di passing sbg argumen ke function dan sebuah function dapat menerima sebuah function pointer sebagai argumen dapat memanggil kembali function yang pointer ini akan tunjuk.

Misalkan kita ingin membuat kode program mengurutkan suatu array namun kita ingin terkadang itu mengurutkan dari terkecil ke terbesar atau kadang dari terbesar ke terkecil, dengan menggunakan callback function kita dapat menghindari penulisan kode berulang yang akan membuat kode kita tidak efisien dan memakan banyak memori.

Berikut adalah kodenya:

```

#include <stdio.h>

int compare(int a,int b) {
    if(a > b) return 1;
    else return -1;
}

void bubbleSort(int *a, int n, int (*compare)(int,int)) {
    int i,j,temp;
    for(i = 0; i < n; i++) {
        for(j = 0; j < n-1; j++) {
            if(compare(a[j],a[j+1]) > 0) { // compare a[j] with a[j+1] and swap if needed
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}

int main() {
    int i, a[] = {3,2,1,5,6,4};
    bubbleSort(a,6,compare);
}

```

```

        for(i = 0; i < 6; i++) printf("%d ",a[i]);
    }

```

Pada program ini di function compare jika  $a > b$  return 1 maka array akan di list dari yang terkecil ke terbesar namun jika kita mengganti  $a > b$  return -1 maka array akan di list dari yang terbesar ke terkecil.

Berikut contoh lain program callback function

Program ini akan memutlukkan semua elemen dalam array dan kemudian mengurutkannya dalam urutan terkecil hingga terbesar.

```

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

int compare(const void* a, const void* b) {
    int a = *((int*)a); // typecasting to int* and getting the value
    int b = *((int *)b);
    return abs(a) - abs(b);
}

int main() {
    int i, a[] = {-32,22,-1,50,-6,4};
    qsort(a,6,sizeof(int),compare);
    for(i = 0; i < 6; i++) printf("%d ",a[i]);
}

```

Outputnya adalah {-1,4,-6,22,-31,50}

## Memory leak

Memory leak adalah situasi dimana kita mempunyai sejumlah memori di dalam heap yang tidak dibebaskan ketika kita telah selesai menggunakannya. Jadi aplikasi kita menyimpan memori yang tidak digunakan di dalam heap.

Contoh dari memory leak dapat kita lihat dalam program berikut

Program ini adalah sebuah game sederhana dimana kita mempunyai 3 kartu yaitu jack queen king masing-masing berada di posisi 1 2 3. Kemudian komputer akan mengacak ketiga kartu ini. Pemain akan menebak posisi queen setelah diacak. Jika pemain benar dia akan memenangkan 3 kali dari taruhannya, jika salah maka pemain akan kehilangan jumlah yang ia pertaruhkan. Pemain mempunyai uang awal sebesar \$100.

```

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

int cash = 100;

void play(int bet) {
    char c[3] = {'J','Q','K'};
    printf("Shuffling...\n");
    srand(time(NULL)); // seeding random number generator
    int i;
    for(i = 0; i < 5; i++) {
        int x = rand() % 3;
        int y = rand() % 3;
        int temp = c[x];
        c[x] = c[y];
        c[y] = temp; // swaps characters at position x and y
    }

    int playerGuess;
    printf("what's the position of queen - 1,2 or 3? ");
    scanf("%d",&playerGuess);
    if(c[playerGuess - 1] == 'Q') {
        cash += bet;
        printf("you win! Result = \"%c %c %c\" total cash = %d\n",c[0],c[1],c[2],cash);
    }
    else {
        cash -= bet;
        printf("you lose! Result = \"%c %c %c\" total cash = %d\n",c[0],c[1],c[2],cash);
    }
}

int main() {
    int bet;
    printf("Welcome to the virtual casino\n");
    printf("total cash = $%d\n",cash);
    while(cash > 0) {
        printf("what's your bet? $");
        scanf("%d",&bet);
        if(bet == 0 || bet > cash) break;
        play(bet);
    }
}

```

Ketika kita menjalankan program ini maka konsumsi memori nya akan tidak berubah dan tetap sama misalkan 348 kb.

Namun ketika kita mengganti baris: `char c[3] = {'J','Q','K'};` menjadi `char *c = {char*}malloc(3*sizeof(char));`

`c[0] = 'J';`

`c[1] = 'Q';`

`c[2] = 'K';`

Disini kita menyimpannya dalam heap dan bukan dalam stack. Ketika kita menjalankan programnya maka semakin lama kita bermain konsumsi memori program ini akan terus-menerus meningkat. Mengapa hal ini bisa terjadi?

Ini terjadi karena ketika program dijalankan, main function akan di taruh ke dalam memori dalam stack dan kemudian dia memanggil fungsi play, pada fungsi play, variable c akan menunjuk ke array 3 character di dalam heap dan ketika fungsi play sudah selesai maka dia akan dihapus dari heap, namun array 3 character di dalam heap tidak dihapus, kemudian fungsi main terus memanggil kembali fungsi play dan akan membuat array 3 character yang baru lagi di dalam heap sehingga menyebabkan memori dalam heap bertambah terus-menerus karena kita menggunakan perintah free utk membebaskan memori yang tidak digunakan dalam heap.

Oleh karena itu kita harus menambahkan perintah `free(c)` setelah

```
    else {
```

```
        cash -= bet;
```

```
        printf("you lose! Result = \"%c %c %c\" total cash = %d\n",c[0],c[1],c[],cash);
```

```
    }
```

di fungsi play.