Modul 2. Fungsi dan Pointer

Jump to bottom

Ivan Sholana edited this page yesterday · 3 revisions

1. Pointer

1.1 Definisi Pointer

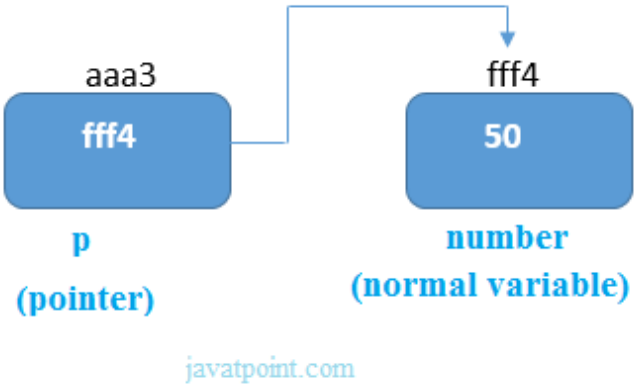
Setiap variabel yang dideklarasikan dalam bahasa C atau pemgrograman lainnya memiliki alamat memori, hal tersebut dikarenakan setiap variabel yang didekalrasikan akan dilakukan alokasi memori pada RAM untuk menyimpan nilai dari variabel tersebut. Untuk membuktikan bahwa setiap variabel memiliki alamat memori maka kita dapat menampilkannya dengan menggunakan tanda `&` pada sebuah variabel.

Contoh:

```
int a = 0;
printf("nilai  a = %d\n",a);
printf("alamat a = %d\n",&a);
```

Kemudian, jika kita dapat mengambil alamat memori coba bayangkan apa yang terjadi jika kita menggunakan alamat memori sebagai sebuah value di dalam variabel lain? dalam kasus tersebut terdapat sebuah variabel yang unik yang disebut `variabel pointer` .

Pointer adalah sebuah variabel yang berisikan `alamat memori` dari variabel tertentu. Dengan berisikan alamat memori sebuah variabel lain pointer memiliki kemampuan untuk **mengakses dan memanipulasi** value yang tersimpan di dalam alamat memori yang variabel pointer simpan. Kemampuan tersebut seakan akan variabel pointer menunjuk atau melakukan `pointing` ke sebuah variabel lain untuk dilakukan operasi dan diakses, oleh karena itu variabel ini disebut `pointer` . Berikut ilustrasi dari pointer:



1.2 Menggunakan Pointer

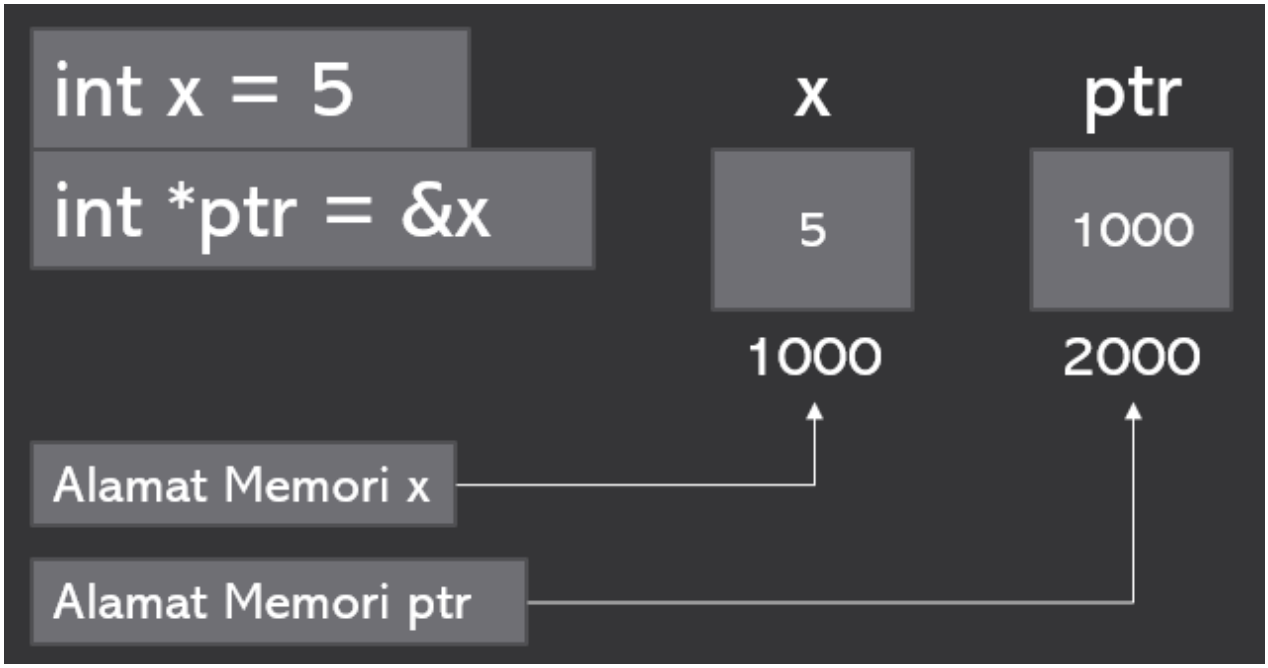
Untuk menggunakan pointer dapat dilakukan dengan mendeklarasikan variabel pointer tersebut. Namun, terdapat perbedaan dalam pendeklarasian variabel biasa dengan variabel pointer. Dalam variabel pointer dikenal operator asterisk `(*)` dan ampersand `(&)` . Operator asterisk memberikan kemampuan variabel pointer untuk mengakses alamat memori yang disimpan sehingga didapatkan nilai yang disimpan dalam alamat memori tersebut.

Adapun penggunaan dari 2 operator tersebut dapat dilihat dalam contoh pendeklarasian variabel pointer berikut:

```
int x = 100;
int *ptr = &x; // --> variabel ptr

printf("ptr = %d\n",ptr);
printf("ptr = %d\n",*ptr);
```

Pada kode program di atas variabel ptr akan menyimpan nilai dari alamat memori variabel x yang kita coba akses menggunakan `&x` . Kemudian coba liat ouput dari program di atas. ptr akan mengoutputkan nilai berupa alamat memori variabel x dan *ptr akan mengouputkan 100 sesuai dengan nilai yang disimpan oleh variabel x yang dipointerkan. Program di atas dapat diilustrasikan seperti gambar berikut:



Pointer sendiri terlebih pada operator asterisk (*) tidak terbatas hanya satu operator saja, kita dapat menggunakan operator asterisk lebih dari satu. Penggunaan paling umum dari operator asterisk yang lebih dari satu adalah `double pointer` . Untuk lebih jelas perhatikan contoh berikut:

```c
#include <stdio.h>

int main()
{
    int x = 100;
    int *ptr = &x;
    int **ptr2 = &ptr;

    printf("x      = %d\n", x);
    printf("&x     = %d\n", &x);
    printf("\n");
    printf("ptr    = %d\n", ptr);
    printf("*ptr   = %d\n", *ptr);
    printf("&ptr   = %d\n", &ptr);
    printf("\n");
    printf("ptr2   = %d\n", ptr2);
    printf("*ptr2  = %d\n", *ptr2);
    printf("**ptr2 = %d\n", **ptr2);
}
```

Coba program di atas di device masing - masing dan analisis hasilnya. Jelaskan hasil analisis di depan kelas!!

1.3 Jenis - Jenis Pointer

1.3.1 Void Pointer

`Void Pointer` merupakan pointer yang tidak memiliki tipe data. Sebelumnya dijelaskan bahwa pointer merupakan variabel dan seperti yang diketahui bahwa perlu dideklarasikan tipe data suatu nilai pada variabel. Namun, pada void pointer hal tersebut tidak perlu dilakukan. Dengan kondisi demikian maka void pointer dapat merujuk alamat memori dari seluruh jenis tipe data. Perhatikan contoh berikut:

```
int main(){
    int a = 10;
    void *ptr = &a;
    printf("%d",*(int*)ptr);
}
```

Untuk mendeklarasikan void ptr maka hanya perlu gunakan void sebagai jenis variabel ptr. Kemudian, ketika ingin mengakses nilai di dalam memori yang dirujuk maka ptr perlu dilakukan casting atau dirubah tipe datanya terlebih dahulu ke tipe data nilai yang dirujuk.

1.3.2 Null Pointer

Null pointer merupakan pointer yang tidak merujuk alamat memori dan memiliki nilai NULL. Tujuan null pointer adalah menyiapkan pointer sebelum dilakukan assigment value terhadap pointer tersebut. Contoh penggunaan dari null pointer adalah ketika ingin melakukan pengaksesan terhadap ekternal file. Ketika eksternal file ada, maka pointer akan berisi alamat memori dari eksternal file tersebut tetapi jika tidak maka pointer akan bernilai null. Kondisi ketika pointer bernilai null dapat dijadikan sebagai handling error dari suatu kejadian sehingga program tidak crash.

```
char str[50];
        FILE *fptr = NULL;

        fptr = fopen("file.txt", "w");

        if(fptr == NULL){
                printf("Error");
                exit(1);
        }
```

Untuk melakukan deklarasi null pointer maka cukup mudah

```
int *ptr = NULL;
```

1.3.3 Dangling Pointer

Dangling pointer adalah pointer yang merujuk terhadap sebuah alamat memori tidak ada. Kejadian ini memungkinkan ketika alamat memori yang dialokasikan secara dinamis dikenai fungsi `free()` atau ketika merujuk alamat memori dari lokal variabel pada fungsi. Dengan begitu memori dengan alamat memori tersebut tidak ada lagi tetapi alamat memori tersebut masih tersimpan di dalam variabel ptr. Oleh karena itu, pointer yang merujuk sautu alamat memori yang tidak ada tersebut disebut dengan `Dangling Pointer` . Untuk memahami pointer ini maka akan dijelaskan lebih lanjut pada mekanisme berjalannya fungsi di dalam memori.

Contoh dari Dangling Pointer:

```
#include <stdio.h>

int* func(){
    int num = 10;
    return &num;
}

int main(){
  int *ptr = NULL;
  ptr = func();
  printf("%d",*ptr);
  return 0;
}
```

Pada program di atas akan terjadi `segmentation fault` atau kesalahan yang terjadi ketika program mencoba mengakses alamat memori yang ilegal.

1.3.4 Wild Pointer

Wild pointer merupakan pointer yang merujuk pada alamat memori yang tidak ada. Namun berbeda dengan dangling pointer, alamat memori yang dirujuk oleh wild pointer memang tidak ada sejak awal bukan tidak ada karena dilakukan free function. Wild pointer tentu akan menyebabkan crash pada memori dikarenakan program tidak mengetahui alamat memori apa yang ingin dirujuk oleh pointer.

Contoh dari wild pointer adalah sebagai berikut:

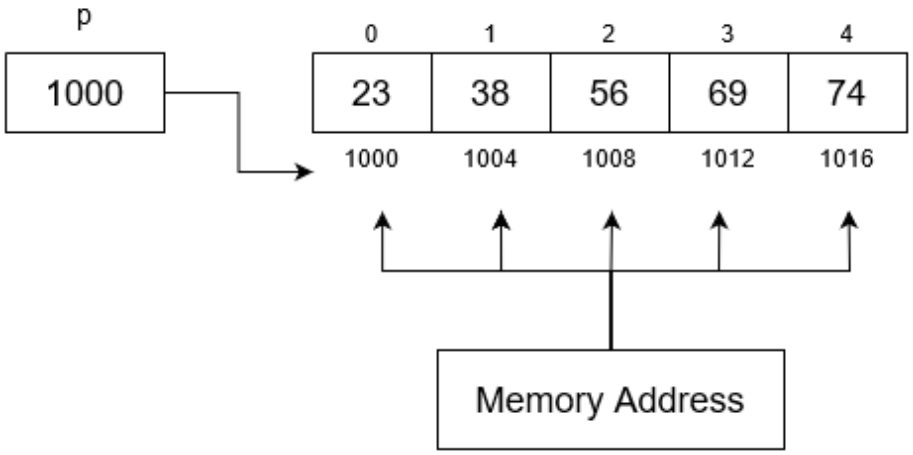
```
int *ptr = 10;
/*  secara sengaja memberikan sembarang value, jika tidak ada alamat memori
dengan alamat 10 maka pointer tersebut aka menjadi wild pointer. */
```

1.4 Pointer dalam Array

Sebelumnya dijelaskan bahwa array merupakan sebuah blok memori yang dialokasikan dengan panjang n dan dapat diakses menggunakan indeks. Namun, pernahkah terpikir kenapa kita dapat mengakses array melalui indeksnya? Didalam pengaksesan indeks array sebenarnya terjadi operasi penambahan alamat memori dengan indeks yang dituju. Alamat memori ini disimpan oleh variabel array. Untuk lebih jelas maka perhatikan contoh berikut:

```
int angka[10];
printf("%d",angka);
```

Output dari program di atas adalah sebuah alamat memori awal dari sebuah baris blok memori yang dialokasikan ketika compile time. Hal ini juga yang menjawab kenapa indeks array berawalan 0 dan kenapa pengaksesan array membutuhkan O(1) atau constant time. Lebih jelasnya perhatikan ilustrasi berikut:



Pada contoh di atas setiap indeks memiliki alamat memori dengan selisih 4 byte. Oleh karena itu ketika mengakses indeks[4], yang sebenarnya terjadi adalah mengakses menggunakan pointer `alamat memori awal + 4 * 4 byte` sehingga indeks[4] sama saja dengan *ptr + 4. Oleh karena itu indeks dimulai dari 0, karena untuk mengakses memori awal atau indeks 0 sama saja dengan `alamat memori awal + 0` sehingga tidak terjadi perpindahan memori. Oleh karena itu hasil dari kode program di bawah ini akan sama:

```
#include <stdio.h>

int main()
{
    int angka[5] = {1, 2, 3, 4, 5};
    printf("*angka   = %d\n", *angka);
    printf("angka[0] = %d\n", angka[0]);

    printf("*angka   = %d\n", *(angka + 3));
```

```
    printf("angka[3] = %d\n", angka[3]);
}
```

Lihat dan analisis hasilnya dan presentasikan di depan kelas!!!

2. Fungsi

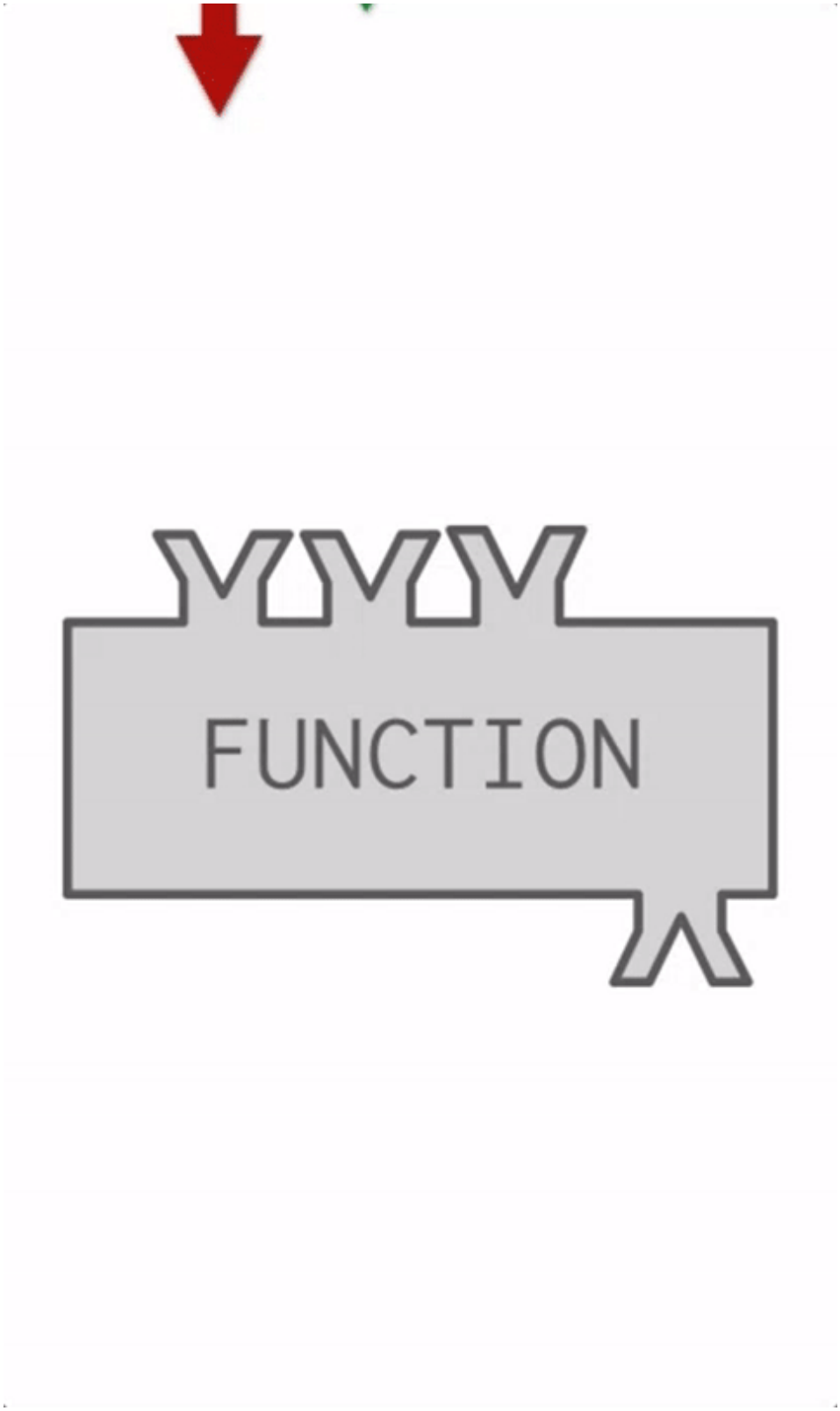
2.1 Definisi Fungsi dan Prosedur

Pada semester sebelumnya kita telah membahas mengenai apa itu fungsi dan prosedur. Pada modul ini kita akan mencoba lebih dalam lagi mengenai fungsi dan prosedur terutama dalam kacamata struktur data. Fungsi dan prosedur keduanya merupakan sub program yang biasanya menjalankan tugas tertentu. Perbedaan di antara fungsi dan prosedur terletak dari ada tidaknya nilai kembalian dan ada tidaknya tipe data dari pendeklarasiannya. Untuk lebih jelaskan contoh dari fungsi dan prosedur berikut ini:

```
// Fungsi
int penjumlahan(int a, int b){
/* a dan b merupakan parameter yang menampung
nilai kiriman dari argumen di mana fungsi tersebut dipanggil
*/
    int c = a + b; // operasi tertentu/khusus yang terjadi di dalam fungsi
    return c; // --> nilai kembalian
}

// Prosedur
void tampil(int a, int b){
    int c = a + b;
    printf("%d",c);
}
```

Dapat dilihat dari contoh kode program di atas sudah cukup jelas menunjukkan perbedaan antara fungsi dan prosedur. Pada fungsi diberikan tipe data dari return value yang akan dikembalikan sedangkan pada prosedur dituliskan `void` yang menunjukkan bahwa prosedur tidak menghasilkan dan mengembalikan nilai apapun. Kemudian pada fungsi karena dia menghasilkan nilai, maka nilai kembalian tersebut perlulah ditampung dan disimpan. Untuk lebih jelasnya perhatikan ilustrasi berikut:



Return value haruslah disimpan di sebuah variabel agar dapat diproses kembali. Untuk lebih jelas perhatikan kode program di bawah ini:

```
#include <stdio.h>

int tambah(int a, int b)
{
   int hasil = a + b;
   return hasil;
}
```

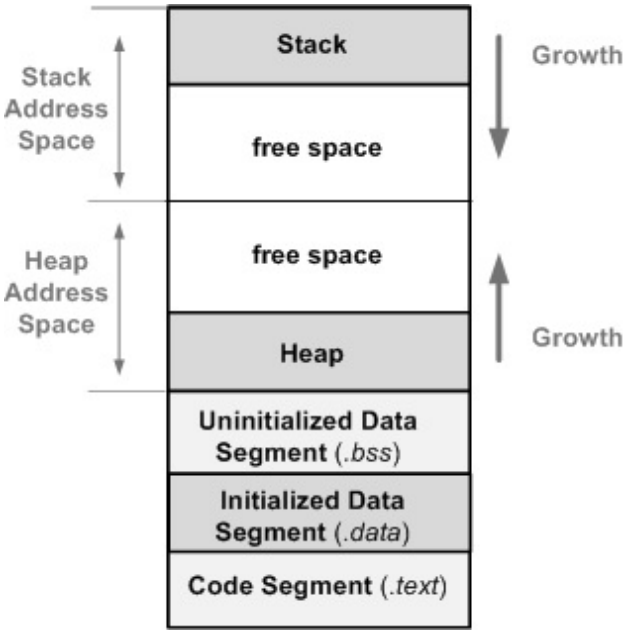
```
int main()
{
    int hasil = tambah(1, 2);
    printf("hasil = %d", hasil);
}
```

Pada kode di atas nilai kembalian dari fungsi tambah berupa variabel hasil akan dikembalikan dan disimpan di dalam variabel hasil pada `int main()` . Dengan pengembalian tersebut dimungkinkan untuk nilai hasil dari penjumlahan a dan b di fungsi tambah ditampilkan pada `int main()` .

Namun, pernahkah berpikir kenapa menggunakan variabel hasil di fungsi tambah saja? kenapa nilai tersebut harus dikembalikan? Untuk menjawab hal tersebut maka mari kita bahas mengenai mekanisme fungsi dan prosedur dalam kacamata struktur data.

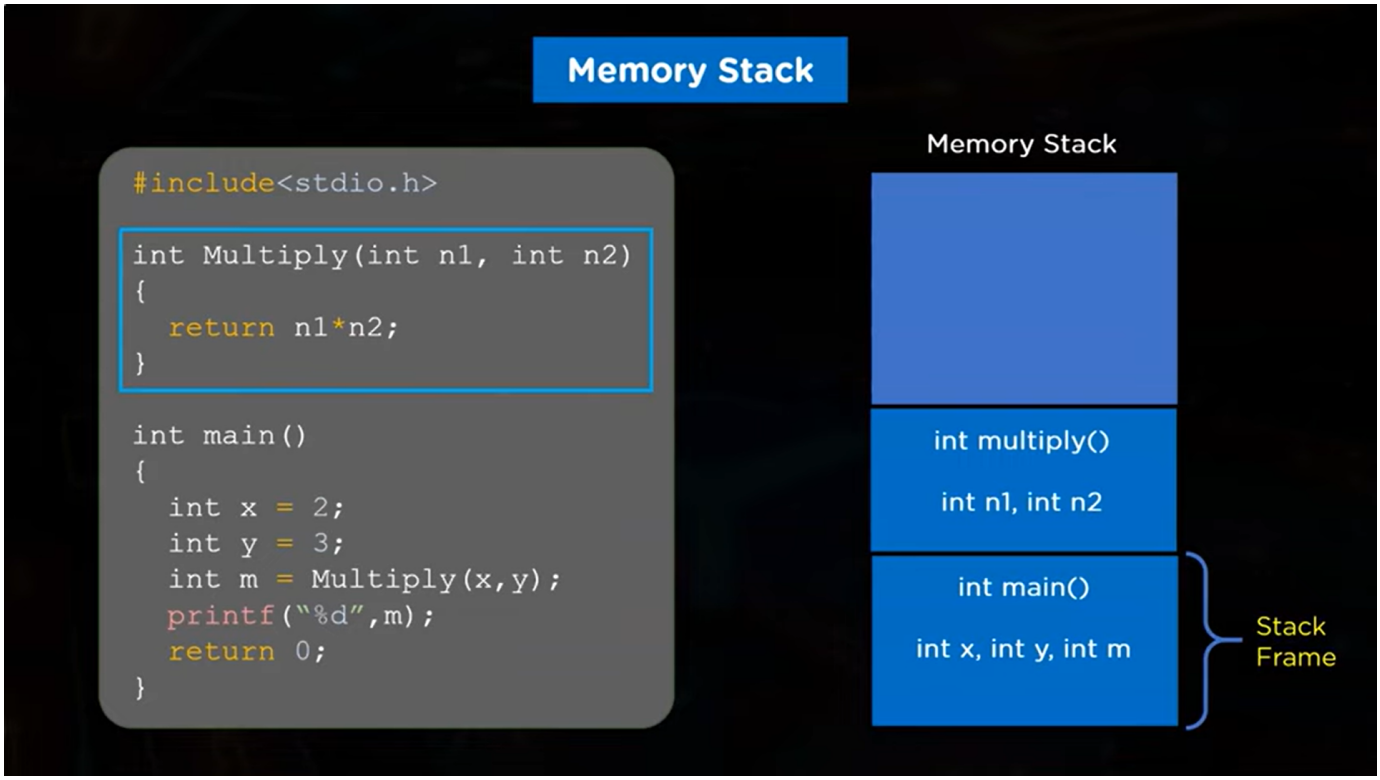
2.2 Segmentasi Memori

Telah dibahas sebelum sebelumnya bahwa ketika sebuah program berjalan maka akan diaolokasikan sejumlah memori untuk menyimpan nilai - nilai yang muncul selama program tersebut berjalan. Lebih dalam lagi alokasi memori tersebut ternyata mempunyai pembagian dalam mengalokasikannya. Untuk lebih jelasnya perhatikan gambar berikut:



Gambar berikut merupakan visualisasi bagaimana sebuah memori dialokasikan ketika sebuah program berjalan. Pada mater kali ini kita akan berfokus pada `stack segmentation` . Tanpa disadari kita telah menggunakan struktur data stack selama ini, bahkan dari pertemuan pertama matkul alpro semester 1. Penggunaan stack ini karena stack segmentation ini sendiri merupakan segmen yang digunakan untuk menyimpan data sementara yang dibutuhkan oleh fungsi-fungsi dalam program. Data di dalam segmen ini ditempatkan secara dinamis saat program berjalan, seperti nilai-nilai dari variabel lokal dan alamat pengembalian fungsi. Segmen ini bersifat read-write, artinya data di dalam segmen ini dapat dibaca dan diubah oleh program.

Secara sederhana ketika program berjalan maka akan dilakukan pengalokasian `stack frame` yaitu area pengalokasian memori dari suatu fungsi yang pada hal ini adalah fungsi utama `int main()` , di dalam stack frame ini akan dialokasikan memori untuk variabel variabel lokal di dalam fungsi tersebut. Kemudian, stack frame akan terbentuk setiap pemanggilan fungsi dilakukan dan akan dihapus ketika fungsi tersebut selesai dijalankan. Untuk lebih jelasnya perhatikan gambar berikut:



Pada gamabar di atas terdapat 2 buah stack frame yaitu int main yang berisi variabel lokal di int main dan multiply yang berisi variabel lokal di fungsi multiply. Ketika kita melakukan return value maka nilai dari stack frame multiply akan dikirimkan terlebih dahulu ke stack frame int main sehingga nilai tersebut tidak hilang. Kenapa hilang? karena ketika fungsi tersebut selesai maka akan dilakukan operasi pop pada stack segment yaitu menghapus stack frame beserta isi dari stack frame tersebut. Hal ini juga yang akan menciptakan error segmentation error pada Dangling pointer. Oleh karena alasan tersebut variabel pada fungsi tidak dapat digunakan di int main yaitu karena operasi pop yang dilakukan oleh stack segmentation memory.

Namun, kita juga dapat mengubah nilai pada fungsi utama tanpa melakukan return value yaitu menggunakan konsep `pass by refference` .

2.3 Pass by Reference

Pemanggilan dengan referensi (Pass By Reference) merupakan penggunaan pointer sebagai parameter fungsi atau prosedur. Dengan menggunakan pointer sebagai paramter dan alamat memori suatu variabel sebagai argumen maka dimungkinkan untuk melakukan manipulasi dan mengakses nilai variabel di fungsi utama pada fungsi atau prosedur tanpa melakukan pengembalian nilai. Hal tersebut karena yang dikirimkan sebagai argumen bukan lagi salinan dari value variabel tetapi alamat dari variabel yang dapat diakses dan mendapatkan nilai asli dari variabel bukan salinannya.

```
// membuat prosedur dengan parameter berupa pointer
void tambah_5 (int *angka){
    *angka = *angka + 5;
}

int main (){
    int nilai = 90;

    printf("Nilai awal adalah %d", nilai);

    /* memanggil prosedur tambah_5 dengan argumen disertai simbol ampersand `&` yang
    digunakan untuk meneruskan alamat variabel nilai ke dalam prosedur tambah_5 */
    tambah_5(&nilai);
    printf("\nNilai setelah ditambah adalah %d",nilai);
}
```

Konsep yang biasa digunakan sebelumnya adalah `Pass by Value` atau mengirimkan salinan nilai suatu variabel untuk diolah, dengan mengirimkan valuenya saja maka tidak akan mempengaruhi variabel aslinya sehingga diperlukan return value.