Bagian 3

3 Pointer

"It's like a finger pointing away to the moon. Don't concentrate on the finger or you will miss all that heavenly glory."

-Bruce Lee

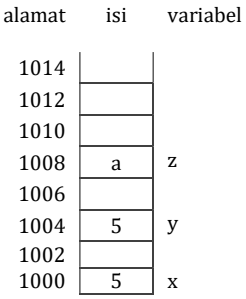
Bab ini membahas tentang pointer, tentang bagaimana data disimpan dan dimanipulasi dalam memori. Pembahasan meliputi operasi – operasi dasar pointer, keterkaitan antara pointer dengan array, dan penggunaan pointer dalam fungsi. Selain itu akan dipaparkan pula mengenai pengalokasian memori secara dinamis. Di akhir bab akan diberikan contoh sebuah studi kasus berupa pembuatan fungsi *random generator* yang dapat menghasilkan sekumpulan bilangan acak.

3.1 Pengertian

Pada dasarnya semua data yang dimanipulasi oleh program tersimpan dalam memori. Data tersebut tersimpan dalam memori di mana masing – masing memiliki alamat unik. Bila terdapat pendeklarasian variabel maka di memori akan dialokasikan tempat untuk variabel tersebut. Sebagai contoh berikut ini adalah deklarasi beberapa variabel.

```
int x{5};
auto y = x;
char z{'a'};
```

Deklarasi di atas dapat diilustrasikan dalam area memori seperti ditunjukkan Gambar 3-1 berikut ini.

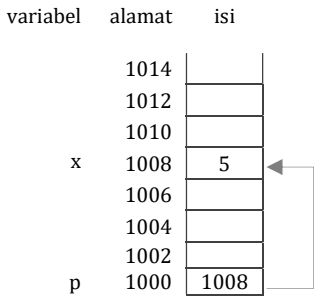


Gambar 3-1 Alokasi Memori Variabel

Dari ilustrasi di atas terlihat bahwa variabel x berada pada alamat 1000, y 1004, dan z 1008. Setiap alamat menyimpan sebuah nilai sesuai tipe datanya. Seperti analogi sebuah loker yang terdiri dari banyak tempat penyimpanan. Tiap tempat penyimpanan memiliki nomor dan tiap nomor menyimpan isinya masing – masing.

Pada dasarnya pointer adalah sebuah variabel seperti umumnya. Bila variabel pada umumnya memiliki tipe data tertentu maka pointer hanya dapat menyimpan alamat memori. Dengan kata lain pointer adalah sebuah objek dalam memori yang menunjuk ke sebuah nilai yang tersimpan di memori berdasarkan alamat nilai tersebut.

Sebuah pointer dapat mereferensi ke sebuah alamat dan mengambil isinya (dereferensi). Gambar 3-2 berikut ini adalah ilustrasi sebuah pointer yang menunjuk ke sebuah alamat.



Gambar 3-2 Pointer

Pointer p terlihat seperti variabel biasa, tetapi nilai yang tersimpan adalah 1008, di mana nilai tersebut adalah alamat dari variabel x yang berisi 5.

3.1.1 Deklarasi

Pendeklarasian pointer dapat dilakukan seperti variabel biasa dengan menambahkan operator unary *. Berikut ini adalah contoh pendeklarasian pointer.

```
int *a;
char *p;
```

a dan p adalah pointer. a merupakan pointer ke tipe int sedangkan p pointer ke tipe char. Sebuah pointer yang dideklarasikan menunjuk ke tipe data tertentu hanya boleh menunjuk alamat memori yang menyimpan nilai dengan tipe data tersebut.

3.1.2 Operator

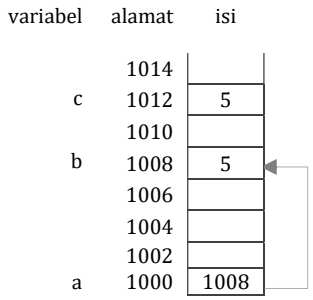
Pointer menggunakan operator unary asterix (*) dan amphersand (&). Kedua operator tersebut bersifat komplemen, artinya satu operator adalah pelengkap operator yang lain. Berikut ini adalah penggunaan kedua operator tersebut:

1. Operator asterix (*) digunakan untuk mendeklarasikan bahwa sebuah variabel adalah pointer.
2. Operator asterix (*) digunakan untuk melakukan pengambilan nilai (dereferensi) suatu alamat memori yang ditunjuk.
3. Operator amphersand (&) digunakan untuk mengambil alamat suatu memori (referensi) melalui nama variabel.

Berikut ini adalah contoh penggunaan operator * dan &.

```
int *a;
int b{4};
int c{b};
a = &b;
```

Deklarasi di atas menerangkan bahwa a sebuah pointer bertipe int, b dan c variabel bertipe int. Kemudian variabel b diisi nilai 4, dan variabel c berisi salinan isi variabel b. Baris berikutnya a menunjuk ke alamat dari variabel b dengan operasi &b. Gambar 3-3 berikut ini adalah ilustrasi pemetaan area di memori untuk deklarasi di atas.



Gambar 3-3 Pemetaan Memori Variabel dan Pointer

Terlihat bahwa pointer a menyimpan (menunjuk, mereferensi) alamat variabel b yaitu 1008. Pointer a dapat mengakses (dereferensi) isi variabel b dengan operasi *a. Bila deklarasi di atas dilanjutkan dengan operasi berikut:

```
cout << “nilai b : ” << b << ‘\n’;
cout << “nilai c : ” << c << ‘\n’;
cout << “alamat b: ” << a << ‘\n’;
cout << “nilai b : ” << *a << ‘\n’;
```

akan menghasilkan tampilan:

```
nilai b : 5
nilai c : 5
alamat b: 1008
nilai b : 5
```

Pointer a menyimpan alamat b, ketika ditampilkan pointer a bernilai 1008 yaitu alamat b. Baris terakhir menampilkan nilai variabel b melalui pointer a. Operasi dilanjutkan seperti berikut:

```
*a = 9;
cout << “nilai b: ” << b << ‘\n’;
cout << “nilai c: ” << c << ‘\n’;
```

Baris pertama berarti “nilai yang ditunjuk oleh a diisi oleh 9”, yang sama artinya dengan mengubah nilai variabel b dengan 9. Berikut ini adalah tampilan yang dihasilkan:

```
nilai b: 9
nilai c: 5
```

Nilai variabel c tidak terpengaruh dengan perubahan di variabel b karena variabel c hanya menyalin nilai variabel b di operasi sebelumnya. Analoginya seperti mengkopi selembar catatan, ketika lembar catatan asli dicoret – coret maka tidak akan mempengaruhi lembar salinannya, demikian sebaliknya.

3.2 Penggunaan

3.2.1 Aritmatika Pointer

Seperti telah dibahas sebelumnya pointer menyimpan sebuah alamat memori, yang berupa nilai numerik. Oleh karena itu sebuah pointer dapat dikenai operasi aritmatik seperti tipe numerik pada umumnya. Sebagai ilustrasi awal potongan kode berikut ini menunjukkan sebuah pointer yang dikenai operasi aritmatik penjumlahan.

```
int x{6};
int* y{&x};
cout << y << ‘\n’;
y++;
cout << y << ‘\n’;
```

Kode di atas berisi sebuah *pointer* `y` ke tipe `int` yang menunjuk ke alamat variabel `x`. Pointer `y` kemudian dikenai operasi *increment* sehingga naik 1 ‘tingkat’. Berikut ini adalah output yang muncul (hasil bisa berbeda).

```
0x28fea8
0x28feac
```

Berdasar output di atas terlihat bahwa alamat yang ditunjuk oleh pointer `y` sebelum dan sesudah operasi *increment* selisih 4 *byte* bukannya 1 *byte*. Hal ini dikarenakan ukuran tipe `int` adalah 4 *byte*. Berikut ini adalah contoh lain operasi aritmatik *pointer* ke tipe `double`.

```
double x{'a'};
double* y{&x};
cout << y << '\n';
cout << (y + 2) << '\n';
```

Tipe `double` memiliki ukuran 8 *byte* sehingga ketika *pointer* `y` ditambahkan dengan 2 maka dapat dipastikan bahwa selisih alamat yang ditunjuk oleh `y` dan `(y + 2)` sebesar (1 + 2 * 8) *byte* yaitu 16 *byte*. Berikut ini adalah output (hasil bisa berbeda) dari potongan kode di atas.

```
0x28fea0
0x28feb0
```

Dari dua ilustrasi di atas dapat disimpulkan bahwa akibat operasi aritmatik pada pointer adalah pergeseran alamat memori yang ditunjuk dengan kelipatan ukuran dari tipe data. Operasi ini dapat digunakan untuk melakukan 'penjelajahan' (*traverse*) elemen – elemen dalam array (§3.2.2).

3.2.2 Array dan Pointer

Aritmatika pointer dapat digunakan untuk melakukan *traversing* elemen – elemen dalam array. Ketika sebuah pointer menunjuk ke sebuah *array* maka yang ditunjuk adalah alamat dari elemen ke-0. Elemen – elemen dalam array tersusun secara berurutan sehingga dengan menggeser pointer ke arah memori sebelahnya berarti sama juga artinya dengan menunjuk ke elemen *array* yang di sebelahnya. Besarnya 1 kali pergeseran adalah sebanyak ukuran tipe data pada *array*. Dengan kata lain bila elemen *array* bertipe `int` maka tiap terjadi 1 kali pergeseran akan melompat sebanyak 4 *byte*.

Potongan kode berikut ini adalah contoh operasi aritmatik pointer terhadab array.

```
int x[]{8, 2, 9, 4, 6, 3, 1};
int* p = x;           // p menunjuk x[0]

p++;                  // p menunjuk x[1]
cout << *p            // 2
cout << *(p + 2)      // 4
```

Berikut ini adalah output ketika kode dijalankan:

```
2
4
```

Berikut ini adalah contoh lain operasi aritmatik pointer untuk *traversing* seluruh elemen array.

```
int x[]{9, 2, 1, 7, 4};
int* p = x;

for(auto i = 0; i < 5; ++i)
        cout << p++ << ' ';
```

Berikut ini adalah output ketika kode dijalankan:

```
9 2 1 7 4
```

Array dalam bahasa C (*C-stye array*) tidak menyimpan informasi mengenai ukuran sehingga mungkin saja terjadi operasi *traversing* yang di luar batasan (*out of bound*). Sebagai contoh bila perulangan pada kode di atas diubah menjadi seperti berikut ini:

```
for(auto i = 0; i < 8; ++i)
```

Maka yang terjadi adalah pengaksesan pointer ke area memori di luar alokasi untuk elemen *array* **x**.

3.2.3 Fungsi dan Pointer

3.2.3.1 Passing Argument by Pointer

Secara default ketika ada sebuah nilai yang dilewatkan (sebagai parameter atau argumen) ke fungsi maka yang terjadi adalah proses penyalinan (copy) dari nilai asli. Dengan kata lain jika terjadi manipulasi terhadap nilai yang masuk tersebut tidak akan mempengaruhi nilai aslinya.

Sebagai ilustrasi berikut ini adalah kode untuk fungsi swap() yang digunakan untuk menukar 2 nilai yang dilewatkan.

```
void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main()
{
        int a{3};
        int b{4};
        swap(a, b);
        // …
}
```

Di main() terdapat 2 variabel yaitu a dengan nilai 3 dan b dengan nilai 4. Saat terjadi pemanggilan terhadap fungsi swap() dengan a dan b sebagai argumennya akan terjadi penyalinan nilai a ke x dan b ke y. Proses penukaran terjadi di dalam fungsi swap() melibatkan variabel lokalnya (x, y, dan tmp). Ketika pemanggilan selesai dan proses dikembalikan ke pemanggil (fungsi main()) maka seluruh variabel lokal di fungsi swap() akan dihapus.

Berdasar kondisi tersebut dapat dikatakan bahwa proses penukaran justru terjadi terhadap x dan y sebagai variabel lokal swap(), bukan pada a dan b sebagai nilai asli yang hendak ditukar.

Pelewatan argumen dengan menyalin nilai asli ke dalam fungsi disebut *passing argument by value* atau sering disingkat *pass by value*. Cara lain untuk melewatkan argumen ke fungsi

adalah dengan *pass by pointer*. Cara ini 'hanya' melewatkan alamat nilai asli ke dalam fungsi. Berikut ini adalah penggunaan pass by value dalam fungsi swap().

```
void swap(int* x, int* y)
{
        int tmp = *x;
        *x = *y;
        *y = tmp;
}

int main()
{
        int a{3};
        int b{4};
        swap(&a, &b);
        // …
}
```

Argumen formal dalam fungsi swap() (x dan y) yang merupakan pointer yang menunjuk ke alamat yang dilewatkan. Saat terjadi pemanggilan fungsi swap() dengan argumen berupa alamat dari variabel a dan b maka x akan menjadi 'handle' untuk a dan y untuk b. Proses penukaran dalam fungsi swap() benar – benar terjadi pada a dan b.

Selain pass by pointer masih terdapat cara lain yaitu pass by reference yang akan di bahas pada bab berikutnya (3.3).

3.2.3.2 Pointer ke Fungsi

Sebagaimana variabel fungsi juga memiliki alamat memori. Alamat memori dari sebuah fungsi dapat ditujuk oleh pointer. Bila pointer ke variabel harus memiliki tipe yang sama maka pointer ke fungsi juga harus memiliki *signature* yang sama. Berikut ini adalah contoh deklarasi pointer ke fungsi:

```
int (*f)(int, int)
```

f adalah sebuah pointer ke fungsi dengan signature berupa return type int dan 2 parameter bertipe int. Semua fungsi yang memiliki signature sama dengan f dapat ditunjuk. Berikut ini adalah lanjutan kode deklarasi di atas:

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

f = &add;
cout << (*f)(4, 5);   // 9

f = &mul;
cout << (*f)(4, 5)    // 45
```

Pemanggilan `f` harus diawali dengan tanda asterik seperti halnya pada pointer ke variabel. Dari kode di atas `f` dapat dipandang seperti layaknya fungsi yang sesungguhnya, tetapi dapat berubah perilaku (*polymorphic*). Pointer ke fungsi seperti ini dalam bahasa C# disebut sebagai tipe *delegate*.

3.3 Reference

Sebuah nilai akan dialokasikan pada alamat memori tertentu. Setiap alamat dapat direferensi oleh pointer. Berbeda dari bahasa C di C++ terdapat cara lain untuk melakukan referensi selain menggunakan pointer yaitu dengan *reference*. Berbeda dengan pointer yang bisa menunjuk ke alamat yang berbeda – beda maka *reference* hanya bisa ke satu alamat saja. Sekali *reference* diinisiasi terhadap alamat tertentu maka selama daur hidup tidak boleh berpindah ke alamat yang lain. Selian itu pengoperasian *reference* lebih natural, seperti halnya variabel biasanya; tidak seperti pointer yang menggunakan notasi asterik untuk menunjuk ke nilai yang direferensi. Potongan kode berikut ini adalah contoh pendeklarasian reference:

```
int x{8};
int& r = x;
```

Deklarasi di atas menjelaskan bahwa r (ditandai dengan *amphersand*) merupakan reference ke x. Perbedaan mendasar antara *pointer* dengan *reference* adalah pada lokasinya. Pada *pointer* alamat antara *pointer* itu sendiri dengan alamat yang direferensi berbeda. *Reference* memiliki alamat yang sama dengan alamat yang direferensinya sehingga dapat dikatakan bahwa reference adalah alias dari sebuah alamat. Untuk membuktikan perhatikan kode berikut yang merupakan lanjutan deklarasi sebelumnya:

```
cout << &x;
cout << &r;
```

Setelah dijalankan dapat dilihat bahwa alamat x dan r adalah sama.

Seperti telah disebutkan pada bahasan sebelumnya (3.2.3.1) tentang *passing argument by pointer* terdapat cara lain untuk melewatkan argumen yaitu *by reference.* Berikut ini adalah contoh fungsi swap() yang menggunakan *passing argument by reference.*

```
void swap(int& x, int& y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main()
{
        int a{3};
        int b{4};
        swap(a, b);
        // …
}
```

Dari kode di atas dapat dilihat bahwa argumen formal pada fungsi swap() adalah *reference* ke argumen aktual ketika terjadi pemanggilan.

3.4 Ukuran Data

Ukuran sebuah variabel dapat dihitung menggunakan operator **sizeof**. Besarnya ukuran tergantung dari besarnya tipe data yang digunakan. Sebagai contoh ukuran tipe int dapat dihitung dengan **sizeof(int)**. Berikut ini adalah contoh potongan kode untuk menampilkan beberapa ukuran tipe data menggunakan operator **sizeof.**

```
cout << "ukuran    int : " << sizeof(int) << endl;
cout << "ukuran   char : " << sizeof(char) << endl;
cout << "ukuran  float : " << sizeof(float) << endl;
cout << "ukuran double : " << sizeof(double) << endl;
```

Tampilan setelah dijalankan adalah sebagai berikut.

```
ukuran    int : 4
ukuran   char : 1
ukuran  float : 4
ukuran double : 8
```

Satuan ukuran data seperti yang tertampil pada contoh output di atas adalah byte. Hal yang sama juga berlaku untuk menghitung variabel bertipe array atau struct. Sebagai contoh sebuah array berukuran 5 bertipe float akan memiliki ukuran sebesar 20 byte (4 *byte* x 5). Potongan kode berikut ini adalah contoh lain untuk perhitungan ukuran array dan struct.

```
struct pixel {
  int x;
  int y;
} p;

float a[5];

cout << "ukuran  array a : " << sizeof(a) << endl;
cout << "ukuran struct p : " << sizeof(p);
```

Contoh output untuk potongan kode di atas adalah sebagai berikut.

```
ukuran    array a : 20
ukuran   struct p : 8
```

Berdasarkan contoh tampilan output di atas ukuran struct p adalah 8 byte karena member dari *struct* adalah x dan y yang masing – masing bertipe int (4 byte x 2).

Namun hal yang berbeda kemungkinan akan terjadi ketika dalam sebuah struct berisi member dengan tipe yang beragam. Berikut ini contoh pendeklarasian struct dengan member beragam sekaligus menampilkan ukurannya.

```
struct pixel {
  int x;
  int y;
  char z;
} p;
...
cout << "ukuran member struct p" << endl;
cout << "  p.x: " << sizeof(p.x) << endl;
cout << "  p.y: " << sizeof(p.y) << endl;
cout << "  p.z: " << sizeof(p.z) << endl;
cout << "ukuran struct p: " << sizeof(p) << endl;
```

Berikut ini adalah contoh tampilan ketika potongan kode di atas dijalankan.

```
ukuran member struct p
  p.x: 4
  p.y: 4
  p.z: 1

ukuran struct p: 12
```

Berdasar contoh tampilan di atas ternyata ukuran struct p berbeda dari jumlah ukuran ketiga membernya. Bila dijumlah total ukuran member struct p adalah: 4 (x: int) + 4 (y: int) + 1 (z: char) = 9. Namun, ketika dihitung secara utuh struct p berukuran 12 sehingga ada selisih 3 byte. Hal ini terjadi berdasar ukuran **word** dalam sistem yang digunakan adalah 4 byte.

Catatan:

> *Word adalah satuan natural untuk data yang digunakan oleh prosesor. Ukuran word berupa bit (binary digit). Seluruh implementasi pada buku ini menggunakan sistem 32 bit, artinya panjang word yang digunakan adalah 32 bit. Bila 1 byte terdiri dari 8 bit, maka dapat dikatakan dalam sistem 32 bit panjang word-nya adalah 4 byte (32 / 8 byte).*

Dalam contoh di atas member x dan y bertipe int di mana ukurannya sama dengan word sehingga tidak ada penambahan. Pada member z yang berukuran 1 byte akan ditambahkan 3 byte agar sesuai dengan jumlah byte dalam word yaitu 4 (1 + 3). Penambahan byte ini biasa disebut sebagai *byte padding*. Seandainya ditambahkan lagi 1 member bertipe char pada struct p maka ukurannya akan tetap 12 byte. Berikut ini adalah contoh modifikasinya.

```
struct pixel {
  int x;
  int y;
  char z;
  char c;
} p;
...
cout << "ukuran member struct" << endl;
cout << "  p.x: " << sizeof(p.x) << endl;
cout << "  p.y: " << sizeof(p.y) << endl;
cout << "  p.z: " << sizeeof(p.z) << endl;
cout << "  p.c: " << sizeof(p.c) << endl;
cout << "ukuran struct p: " << sizeof(p) << endl;
```

Bila dijalankan akan menghasilkan output sebagai berikut.

```
ukuran member struct p
  p.x: 4
  p.y: 4
  p.z: 1
  p.c: 1

ukuran struct p: 12
```

Berdasar tampilan output di atas bila ukuran tiap member dijumlah akan menghasilkan 10 byte, tetapi ukuran struct tetap 12 byte seperti sebelum ada penambahan member c yang bertipe char.

Selain itu urutan pendeklarasian member dalam struct juga berpengaruh terhadap ukuran total sebuah struct. Sebuah struct dengan ukuran tertentu bisa berubah ukurannya karena perubahan urutan pendeklarasian member.

3.5 Alokasi Dinamis

3.5.1 Wild Pointer

Seperti telah dibahas sebelumnya pendeklarasian sebuah pointer dapat dilakukan seperti variabel pada umumnya. Ketika sebuah variabel dideklarasikan maka akan dialokasikan sebuah tempat di memori dan akan diisikan dengan nilai default sesuai dengan tipe data variabel yang bersangkutan.

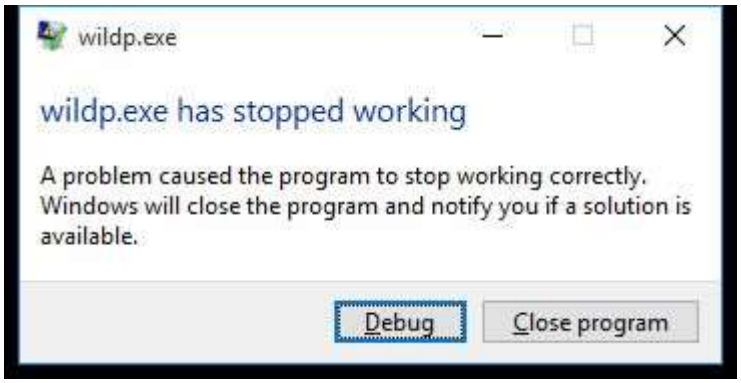
Sebuah pointer ketika dideklarasikan akan secara otomatis menunjuk ke sebuah alamat memori secara acak. Setelah pendeklarasian pointer dapat melakukan pengaksesan maupun pengubahan nilai pada alamat yang ditunjuk. Berikut contoh deklarasi 2 buah pointer ke tipe int dan char.

```
int *a;
char *b;

*a = 9;
*b = 'a';

cout << "*a: " << *a << ", " << *b: " << *b << endl;
```

Bila potongan kode di atas dijalankan maka kemungkinan besar akan terjadi crash. Berikut ini adalah contoh pesan kesalahan yang muncul ketika program dijalankan.



Gambar di atas adalah tampilan pesan kesalahan yang muncul di lingkungan sistem Windows 10 menggunakan compiler GCC/g++ versi 4.9.2

Pada saat pointer dideklarasikan maka secara otomatis akan menunjuk ke sebuah alamat memori secara acak. Apa yang terjadi ketika pointer menunjuk ke alamat yang sedang digunakan oleh proses lain di luar, kemudian mengubah nilai pada alamat yang ditunjuk tersebut? Maka sistem operasi akan menganggap itu sebagai operasi yang ilegal sehingga akan terblokir.

Pendeklarasian pointer seperti di atas sering di sebut sebagai 'wild pointer'. Untuk mengantisipasi hal tersebut biasanya pada saat deklarasi pointer akan diinisialisasi dengan nilai *nullptr* sebelum akhirnya menunjuk ke sebuah alamat yang memang sudah ditentukan. Berikut ini modifikasi deklarasi pada potongan kode sebelumnya.

```
int *a = nullptr;
char *b = nullptr;
```

Penunjukan ke nilai *nullptr* berarti *pointer* tidak menunjuk ke alamat manapun.

3.5.2 Alokasi dan Dealokasi

Pembahasan 'wild pointer' di atas menunjukkan bahwa sebuah pointer harus menunjuk ke alamat yang memang sudah ditentukan untuk menghindari resiko pelanggaran operasi. Selain itu pendeklarasian pointer seperti bahasan sebelumnya bersifat 'statis', artinya sekali pointer dideklarasikan maka akan mengambil sebuah alamat (seperti variabel pada umumnya) untuk menyimpan nilai yang juga berupa alamat dan akan hidup sepanjang program (atau fungsi) berjalan, walaupun sudah tidak terpakai.

Selain dialokasikan dengan cara 'statis' seperti sebelumnya, pointer dapat dialokasikan juga secara dinamis yang artinya dapat digunakan hanya pada saat diperlukan saja dan dapat dihapus (dealokasi) setelah selesai. Berikut ini adalah contoh potongan kode yang menggunakan alokasi memori secara dinamis.

```
int *a = new int;
char *b = new char;

*a = 9;
*b = 'A';

cout << "a menunjuk ke: “ << a << “, nilai: " << *a << endl;
cout << "b menunjuk ke: “ << b << “, nilai: " << *b << endl;
```

Pada saat deklarasi, pointer a dan b akan dialokasikan memori untuk ditunjuk menggunakan operator new. Operator new akan mengalokasikan memori seukuran tipe data yang dialokasikan. Berdasar deklarasi di atas alokasi untuk pointer a adalah seukuran tipe data int, dan b seukuran tipe char. Ukuran sebuah tipe data didapat menggunakan operator **sizeof.**

Berikut ini adalah contoh tampilan bila potongan kode di atas dijalankan (alamat bisa berbeda).

```
a menunjuk ke: 008A0F38, nilai: 9
b menunjuk ke: 008A0FB8, nilai: A
```

Potongan kode di atas tidak menyebabkan crash walaupun alamat yang ditunjuk oleh pointer a dan b adalah acak dan diubah nilainya. Hal tersebut karena ketika terjadi pengalokasian memori secara dinamis, compiler akan memastikan hanya memori yang bebas saja yang akan dialokasikan sehingga tidak menyebabkan pelanggaran akses. Selain itu juga alokasi memori yang sudah tidak digunakan dapat dibebaskan (dealokasi) sehingga penggunaan memori lebih efisien. Berikut ini adalah modifikasi dari potongan kode sebelumnya.

```
int *a = new int;
```

```
    char *b = new char;

    *a = 9;
    *b = 'A';

    cout << "a menunjuk ke: “ << a << “, nilai: " << *a << endl;
    cout << "b menunjuk ke: “ << b << “, nilai: " << *b << endl;

    delete a;
    delete b;
    cout << "setelah dealokasi\n";

    cout << "a menunjuk ke: “ << “, nilai: ", a, *a);
    printf("b menunjuk ke: %p, nilai: %c\n", b, *b);
```

Berikut ini adalah contoh tampilan bila potongan kode di atas dijalankan (alamat bisa berbeda).

```
a menunjuk ke: 008A0F38, nilai: 9
b menunjuk ke: 008A0FB8, nilai: A
setelah dealokasi
a menunjuk ke: 008A0F38, nilai: 7868344
b menunjuk ke: 008A0FB8, nilai: P
```

Dari contoh tampilan di atas setelah didealokasikan pointer tetap menunjuk alamat yang sama. Walaupun alamat yang ditunjuk masih sama tetapi sudah bukan ‘milik’ pointer yang bersangkutan sehingga bila dipaksa untuk diisi nilai kemungkinan besar akan terjadi crash, kecuali sebelumnya di re-alokasi.

Selain keuntungan berupa efisiensi penggunaan memori, pengalokasian memori dinamis ini juga mengakibatkan ‘sampah memori’ ketika pemrogram lupa untuk men-dealokasikan memori yang sudah tidak digunakan (berbeda dengan bahasa Java yang memiliki *garbage collector* untuk membersihkan sampah memori secara otomatis setelah memori tidak digunakan). Oleh karena itu penanganan alokasi memori secara dinamis harus dilakukan dengan sangat cermat.

3.6 Contoh Kasus

Sebuah program membutuhkan fungsi yang dapat menghasilkan n bilangan acak antara 1 – n, misal fungsi tersebut diberi nama *getrandom()*.

Penyelesaian

Fungsi *getrandom()* membutuhkan sebuah parameter, misal bernama n, bertipe int untuk menentukan batas bilangan acak yang dihasilkan. Berikut ini adalah implementasi fungsi *getrandom().*

```
int *getrand(int n)
{
    int* r{new int[10]};

    srand((unsigned)time(NULL));
    for(auto i = 0; i < n; ++i)
        r[i] = rand() % n + 1;
```

```
    return r;
}
```

Dalam fungsi *getrandom()* dideklarasikan pointer **r** yang menunjuk ke array berukuran **n** yang dialokasikan secara dinamis**.**

Berikut ini adalah contoh penggunaannya.

```
int main() {
    int* rands{getrand(5)};

    for(auto i = 0; i < n; ++i)
        cout << rands[i] << endl;

    // dealokasi
    delete [] rands;

    return 0;
}
```

Contoh tampilan setelah dijalankan adalah sebagai berikut.

```
4
3
5
2
2
```

3.7 Latihan

1. Pada fungsi main() di contoh penggunaan fungsi getrand() di atas,
    a. Mengapa harus melakukan dealokasi?
    b. Mengapa dealokasi dilakukan di fungsi main(), bukan di getrand()?

2. Diketahui sebuah deklarasi sebagai berikut:

    ```
    int a[]{2, 3, 4, 5, 6};
    int *x, *y, z;
    ```

    Kemudian dikenai operasi – operasi berikut secara berurutan:
    (1) x = a;
    z = *x;
    Berapa nilai z?
    (2) *(x + 1) = z;
    Berapa nilai array **a** sekarang?
    (3) y = &a[2];
    *y = (*x)++ + z;
    Berapa nilai array **a** sekarang?

Buatlah fungsi untuk mengacak sebuah string yang dimasukkan. Fungsi memiliki sebuah parameter bertipe string sebagai masukan dan mengembalikan nilai string yang sudah teracak. (nama fungsi: *scramble()*)