Pointer dalam Bahasa C

IF2110/IF2111 – Algoritma dan Struktur Data
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

Tujuan

Mahasiswa memahami sintaks dan pengertian pointer (dalam bahasa C)

Mahasiswa mengerti penggunaan pointer dengan benar

Mahasiswa memahami mekanisme kerja pointer dalam memory

Referensi

Materi diadopsi dari: Pointers and Memory, Nick Parlante ©1998-2000.

http://cslibrary.stanford.edu/102/PointersAndMemory.pdf

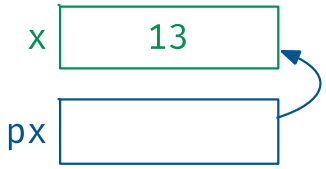
Prinsip Dasar Pointer

pointer tidak menyimpan sebuah nilai
pointer menyimpan alamat dari sebuah variabel

Apakah **pointer** itu?

- Adalah variabel yang menyimpan *reference* dari nilai lain.
- Berbeda dengan variabel "biasa" yang menyimpan nilainya sendiri.

```
int x;
int *px;
x  = 13;
px = &x;
  /* reference to */
```



px menyimpan alamat dari x

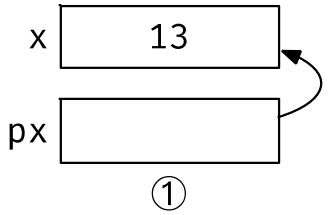
Mengapa pointer?

- Memungkinkan dua bagian/*section* dalam program berbagi akses informasi dengan mudah
- Memungkinkan struktur data berkait/*linked* yang rumit (seperti *linked list*, *tree* berbasis *node*)

Pointer Dereference

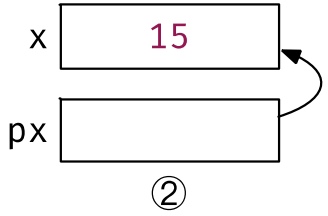
Operasi "*dereference*" adalah operasi untuk mendapatkan nilai yang diacu oleh sebuah pointer.

```
int x;
int *px;

① x  = 13;
  px = &x;
    /* reference to */
```



bentuk dereference, nanti bakal ngacu ke nilai variabelnya

```
② (*px) += 2;
    /* dereference */
```



Null Pointer, Pointer Assignment

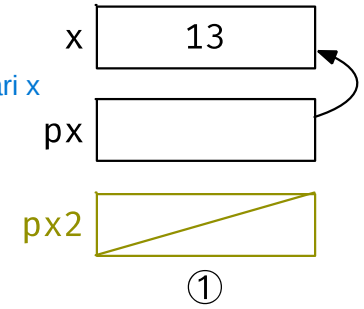
Null pointer: nilai khusus untuk menyatakan bahwa sebuah pointer tidak menunjuk ke mana-mana.

Operator assignment "="

Sebuah pointer di-assign **dengan pointer lain** untuk mengacu nilai yang sama

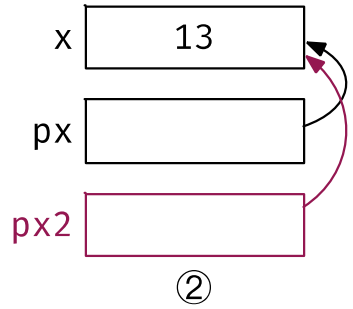
```
int x;
int *px, *px2;

① x  = 13;
  px = &x;
    /* reference to */
  px2 = NULL;
```

px diisi alamat dari x




```
② px2 = px;
```

jadi sama-sama mengacu ke x



Bad Pointer

Pointer yang belum diinisialisasi.

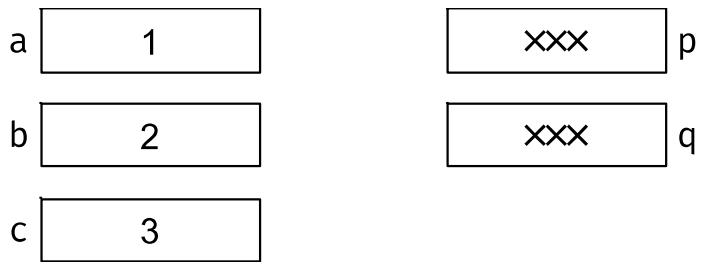
Dereference terhadap bad pointer menyebabkan *runtime error*.

```
int *px;
```

px | xxx

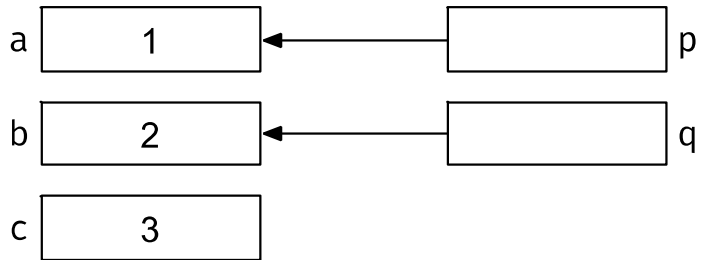
Contoh (1)

```
// allocate three integers and two pointers
int a = 1;
int b = 2;
int c = 3;
int* p;
int* q;

// Here is the state of memory at this point.
// T1 -- Notice that the pointers start out bad...
```



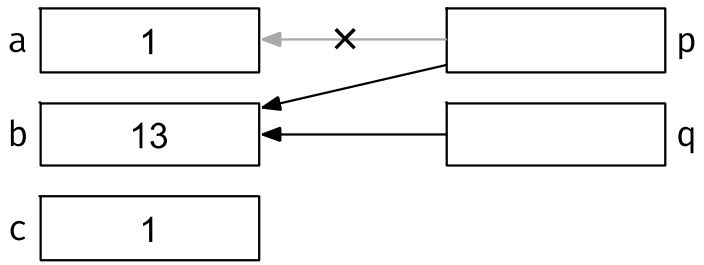
Contoh (2)

```
p = &a; // set p to refer to a
q = &b; // set q to refer to b

// T2 -- The pointers now have pointees
```



Contoh (3)

```
// Now we mix things up a bit...
c = *p;  // retrieve p's pointee value (1) and put it in c
p = q;   // change p to share with q (p's pointee is now b)
*p = 13; // dereference p to set its pointee (b) to 13
         // (*q is now 13)

// T3 -- Dereferences and assignments mix things up
```



Memori Lokal

Alokasi dan **Dealokasi**

Ketika variabel diberikan tempat pada memori untuk menyimpan nilai: ***allocated***.

Ketika variable tidak lagi memiliki tempat pada memori untuk menyimpan nilai: ***deallocated***.

Periode antara allocated-deallocated: ***lifetime***.

Alokasi, Dealokasi, Lifetime

```c
void foo(int a) { // (1) Locals (a, i, scores) allocated when foo runs
    int i;
    float scores[100]; // This array of 100 floats is allocated locally.

    a = a + 1;     // (2) Local storage is used by the computation
    for (i=0; i<a; i++) {
        bar(i + a);    // (3) Locals continue to exist undisturbed,
    }                  // even during calls to other functions.

} // (4) The locals are all deallocated when the function exits.
```

Contoh

```
void X() {
  int a = 1;
  int b = 2;
  // T1

  Y(a);
  // T3
  Y(b);
  // T5
}

void Y(int p) {
  int q;
  q = p + 2;
  // T2 (first time through),
  // T4 (second time through)
}
```

T1 - X()'s locals have been allocated and given values..	**T2** - Y() is called with p=1, and its locals are allocated. X()'s locals continue to be allocated.	**T3** - Y() exits and its locals are deallocated. We are left only with X()'s locals.	**T4** - Y() is called again with p=2, and its locals are allocated a second time.	**T5** - Y() exits and its locals are deallocated. X()'s locals will be deallocated when it exits.
X() a 1, b 2	Y() p 1, q 3 X() a 1, b 2	X() a 1, b 2	Y() p 2, q 4 X() a 1, b 2	X() a 1, b 2

Contoh error

```
// T.A.B. -- The Ampersand Bug function
// Returns a pointer to an int
int* tab() {
  int temp;
  return(&temp); // return a pointer to the local int
}

void victim() {
  int* ptr;
  ptr = tab();
  *ptr = 42; // Runtime error! The pointee was local to tab
}
```

tab adalah sebuah pointer

memanggil tab yang berupa pointer

error karena di tab punya local variable

Function Call Stack

Lihat materi Nick Parlante halaman 15-16

Passing parameter by value vs Passing parameter by reference

Passing Parameter: by Value

yg diproses itu nilainya, bukan pointer

```
void B(int worth) {
  worth = worth + 1;
  // T2
}
void A() {
  int netWorth;
  netWorth = 55; // T1

  B(netWorth);
  // T3 -- B() did not change netWorth
}
```

T1 -- The value of interest `netWorth` is local to `A()`.	**T2** -- `netWorth` is copied to `B()`'s local worth. `B()` changes its local worth from 55 to 56.	**T3** -- `B()` exits and its local `worth` is deallocated. The value of interest has not been changed.
A() netWorth 55	B() worth ~~55~~ 56 A() netWorth 55	A() netWorth 55

Passing Parameter: by Reference

```
// B() now uses a reference parameter -- a pointer to the value of interest.
// B() uses a dereference (*) on the reference parameter to get at the value
// of interest.
void B(int* worthRef) {        // reference parameter
    *worthRef = *worthRef + 1; // use * to get at value of interest
    // T2     merupakan pointer
}
void A() {
    int netWorth;
    netWorth = 55; // T1 -- the value of interest is local to A()
    B(&netWorth);  // Pass a pointer to the value of interest.
                   // In this case using &.
    // T3 -- B() has used its pointer to change the value of interest
}
```

T1 -- The value of interest `netWorth` is local to `A()` as before.	**T2** -- Instead of a copy, `B()` receives a pointer to `netWorth`. `B()` dereferences its pointer to access and change the real `netWorth`.	**T3** -- `B()` exits, and `netWorth` has been changed.
A() netWorth 55	B() worthRef; A() netWorth ~~55~~ 56	A() netWorth 56

tidak selalu diperlukan

Apakah “&” selalu diperlukan?

```
// Takes the value of interest by reference and adds 2.
void C(int* worthRef) {
  *worthRef = *worthRef + 2;
}

// Adds 1 to the value of interest, and calls C().
void B(int* worthRef) {
  *worthRef = *worthRef + 1; // add 1 to value of interest as
                             // before
  C(worthRef); // NOTE: no & required. We already have
               // a pointer to the value of interest, so
               // it can be passed through directly.
}
```

C(worthRef) gaperlu C(&worthRef) karena tipenya uda sama" pointer

Heap Memory

Heap Memory == Dynamic Memory

- Berbeda dengan Local Memory yang mengalokasi dan dealokasi memory secara otomatis saat function call
- Pada Heap Memory, programmer harus melakukan alokasi dan dealokasi

Keuntungan heap memory:

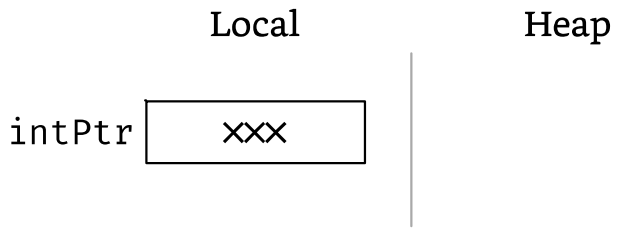
- Lifetime
- Ukuran ukurannya dinamis, bisa makin besar

Kekurangan:

- more works
- more bugs

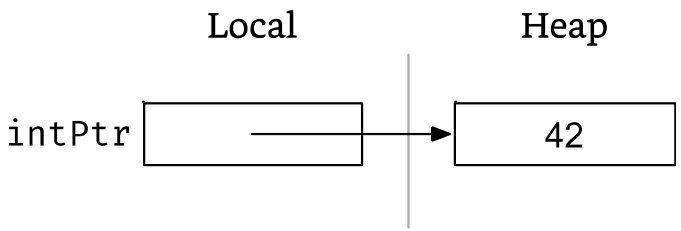
Contoh Penggunaan Heap Memory (1)

```
void Heap1() {
    int* intPtr;
    // Allocates local pointer local variable (but not its pointee)
    // T1:
```




```
    // Allocates heap block and stores its pointer in local
    // variable.
    // Dereferences the pointer to set the pointee to 42.
    intPtr = malloc(sizeof(int));
    *intPtr = 42;
    // T2:
```

allocate to heap memory

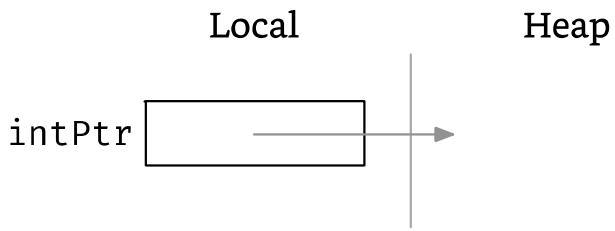


kalau misal ada nilai pointer baru, pointer lama tuh gabisa diakses lagi

contoh misal di samping kan pointernya nunjuk nilai 42
terus ada ketambahan nilai 43, jadi pointernya bisanya akses 43, bukan 42 lagi

Contoh Penggunaan Heap Memory (2)

```
    // Deallocates heap block making the pointer bad.
    // The programmer must remember not to use the pointer
    // after the pointee has been deallocated (this is
    // why the pointer is shown in gray).
    free(intPtr);
    // T3:
```




```
}
```

Referensi Tambahan

Materi pointer, array dan string:
A Tutorial on Pointers and Arrays in C, Ted Jensen, 2003.
Bab 2, 3, dan 4

http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

http://pw1.netcom.com/~tjensen/ptr/pointers.htm