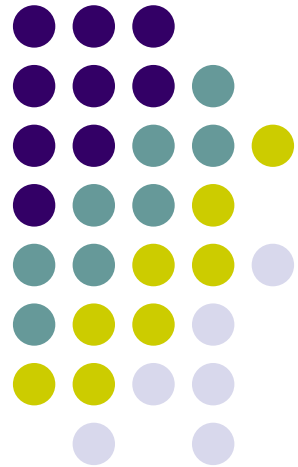
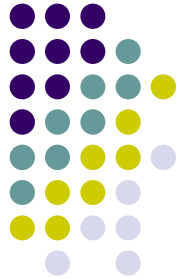
Teknik Kompiler 10

oleh: **antonius rachmat c, s.kom**



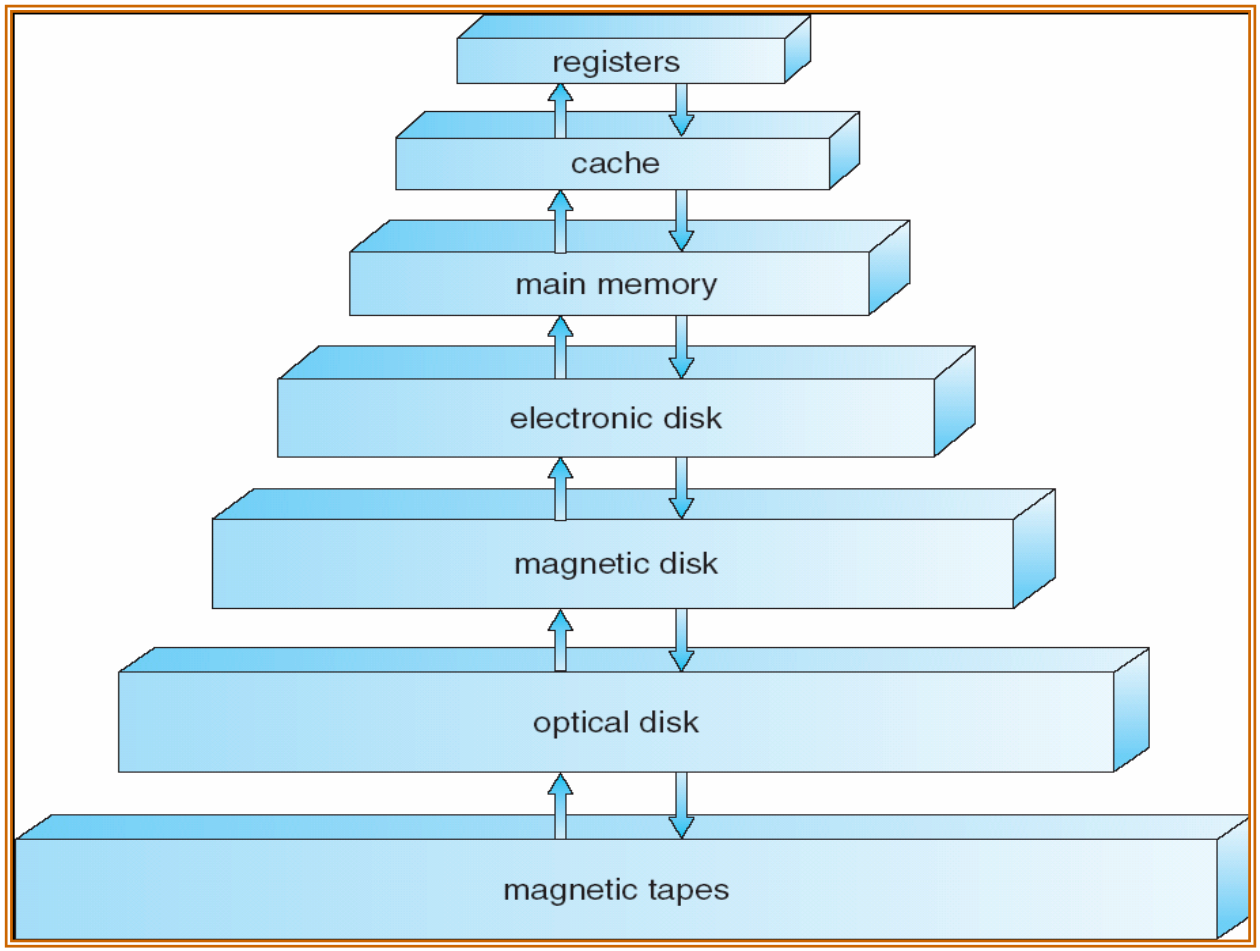
Tempat Penyimpanan (Storage)



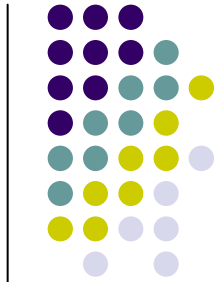
- Storage systems organized in hierarchy.
  - Speed (kecepatan)
  - Cost (harga)
  - Volatility (kemampuan menyimpan data)

- Register: cepat tapi ukuran terbatas
- Memory: lambat tapi ukuran besar

Storage Hierarchy

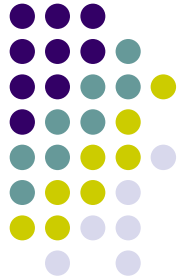


Runtime Environment



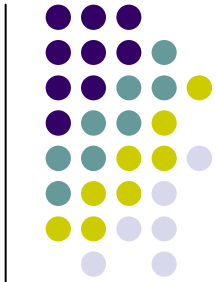
- Mengatur hubungan antara source program dan aksi yang harus dilakukan pada saat runtime untuk mengimplementasikan program itu.
- Mengatur alokasi atau dealokasi memory dari dan ke data objects pada saat runtime
- Mengatur perbedaan bagian-bagian prosedur mana saja yang aktif pada saat tertentu. Namun pada kenyataanya, user hanya tahu bahwa semua bagian-bagian prosedur tersebut harus keliahatan **hidup semua** pada saat runtime.
- Eksekusi suatu prosedur disebut dengan *aktivasi suatu prosedur*. Jika suatu prosedur rekursif, maka beberapa aktivasinya akan hidup pada saat yang sama. Masing-masing pemanggilan fungsi rekursif akan menggunakan memori yang berbeda-beda.

Asumsi



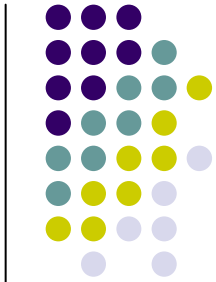
- Kita menggunakan nama prosedur baik itu untuk procedure maupun function.
- Prosedur memiliki nama identifier, kadang memiliki variabel/parameter formal maupun aktual.

Flow Control



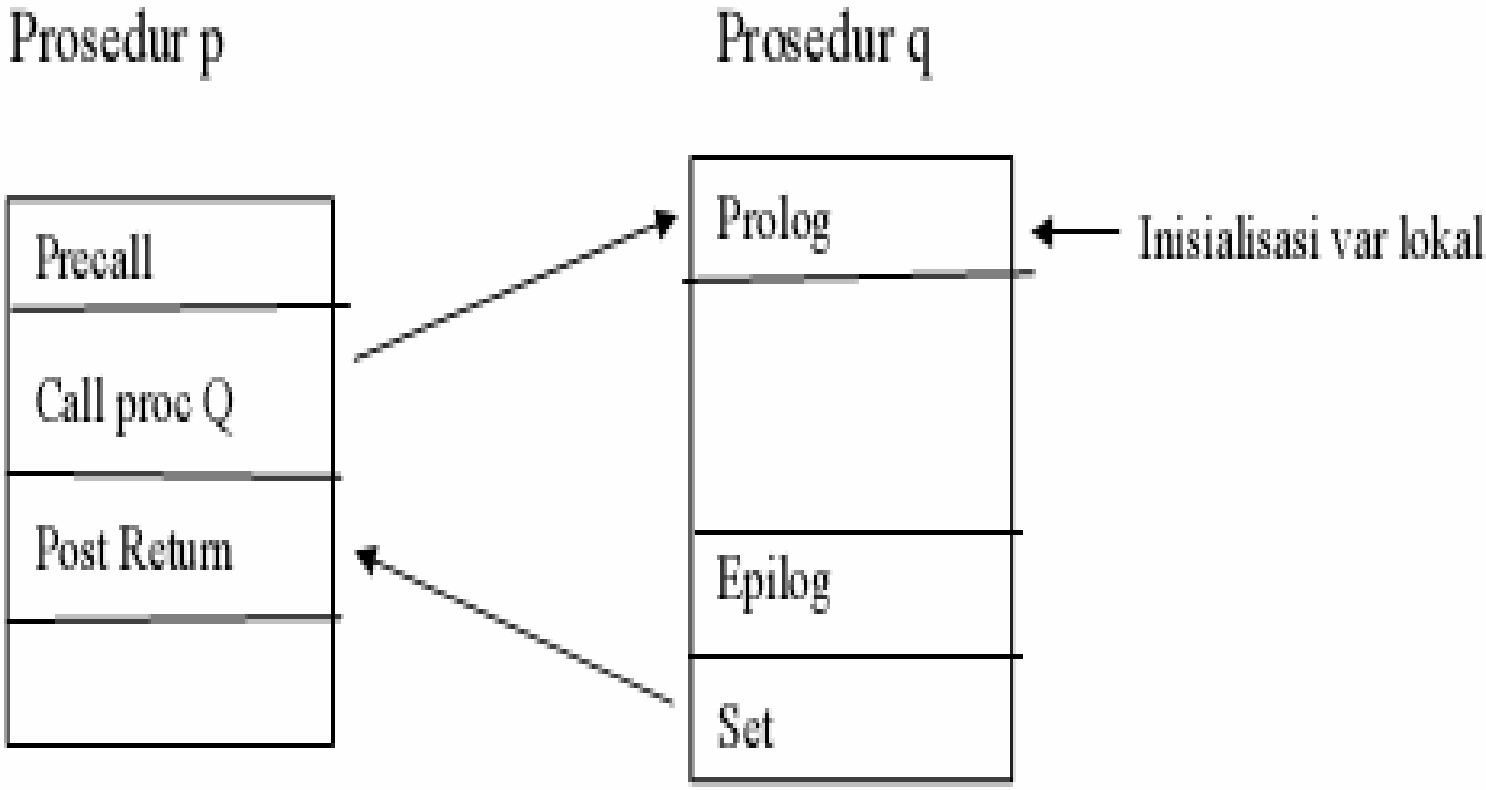
- Sequential Flow Control
  - Eksekusi program terdiri dari barisan langkah-langkah, dengan control pada tempat-tempat tertentu di program.
  - Tidak dieksekusi secara paralel
- Procedure execution
  - Setiap eksekusi prosedur dimulai dari awal badan prosedur.
  - Setiap suatu prosedur selesai dieksekusi akan mengembalikan kendali ke tempat yang tepat sesudah dimana prosedur itu dipanggil.
  - Tidak ada *backtraking*.

Procedure Lifetime

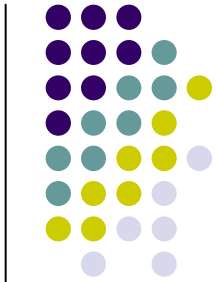


- Waktu hidup (*lifetime*) suatu prosedur adalah dimulai dari barisan langkah pertama sampai dengan langkah terakhir pengeksekusian badan prosedur itu.
- Jika dalam suatu program besar memiliki prosedur-prosedur kecil di dalamnya, maka pada saat pemanggilan prosedur kecil dan setelah eksekusi prosedur kecil itu selesai, maka point eksekusi akan kembali lagi melanjutkan eksekusi program yang memanggilnya.
- Contoh ketika prosedur q dipanggil dari prosedur p, dan prosedur q selesai diekseskusi, maka kendali akan kembali lagi ke prosedur p.

Procedure Activation

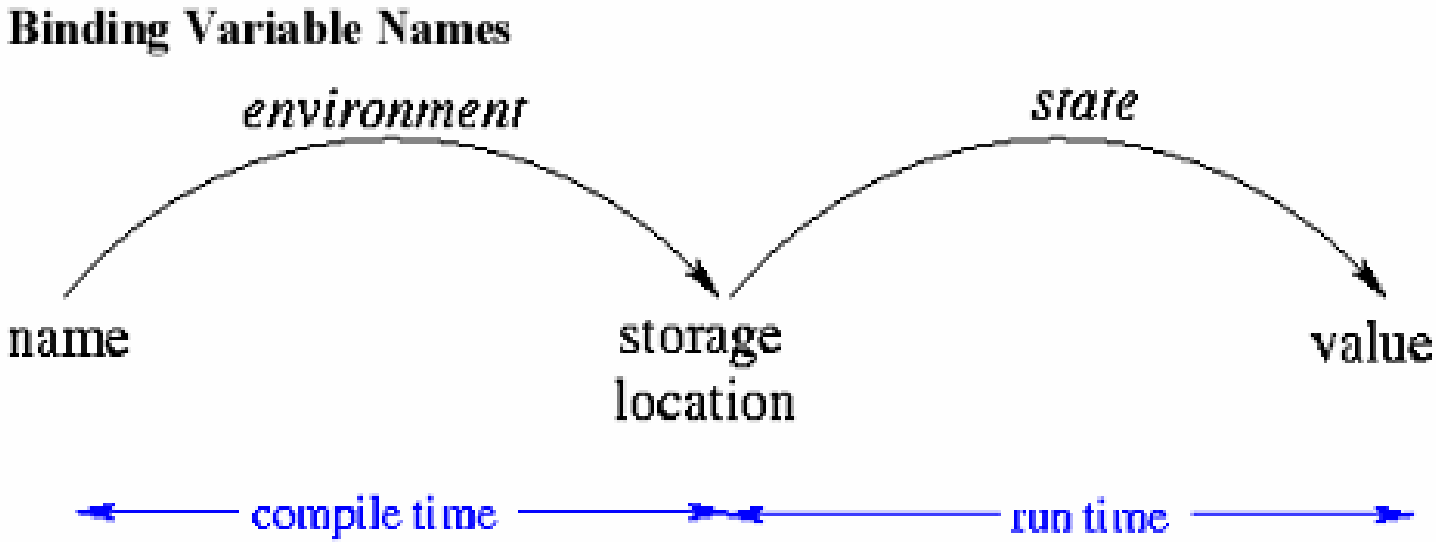


Procedure Activation



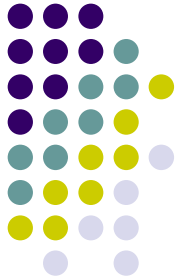
- Prolog : inisialisasi variabel lokal
- Epilog :
  - copy nilai return value ke posisinya
  - ubah stack pointer ke nilai return value
  - goto return address dan hapus variabel lokal.
- Post Return :
  - copy return value ke variabel
  - geser stack pointer dan hapus parameter dan variabel bagian itu
  - goto next execution
- Aktivasi prosedur dapat diimplementasikan dengan menggunakan *control stack*:
  - *push* pada saat aktivasi sebuah prosedur
  - *pop* node pada saat selesai eksekusi sebuah prosedur.

Binding Variable Name



Jika nama variabel dideklarasikan di dalam program, maka nama yang sama dapat menyatakan obyek data yang berbeda. Istilah obyek data mengacu pada tempat penyimpanan yang menyimpan nilai data.

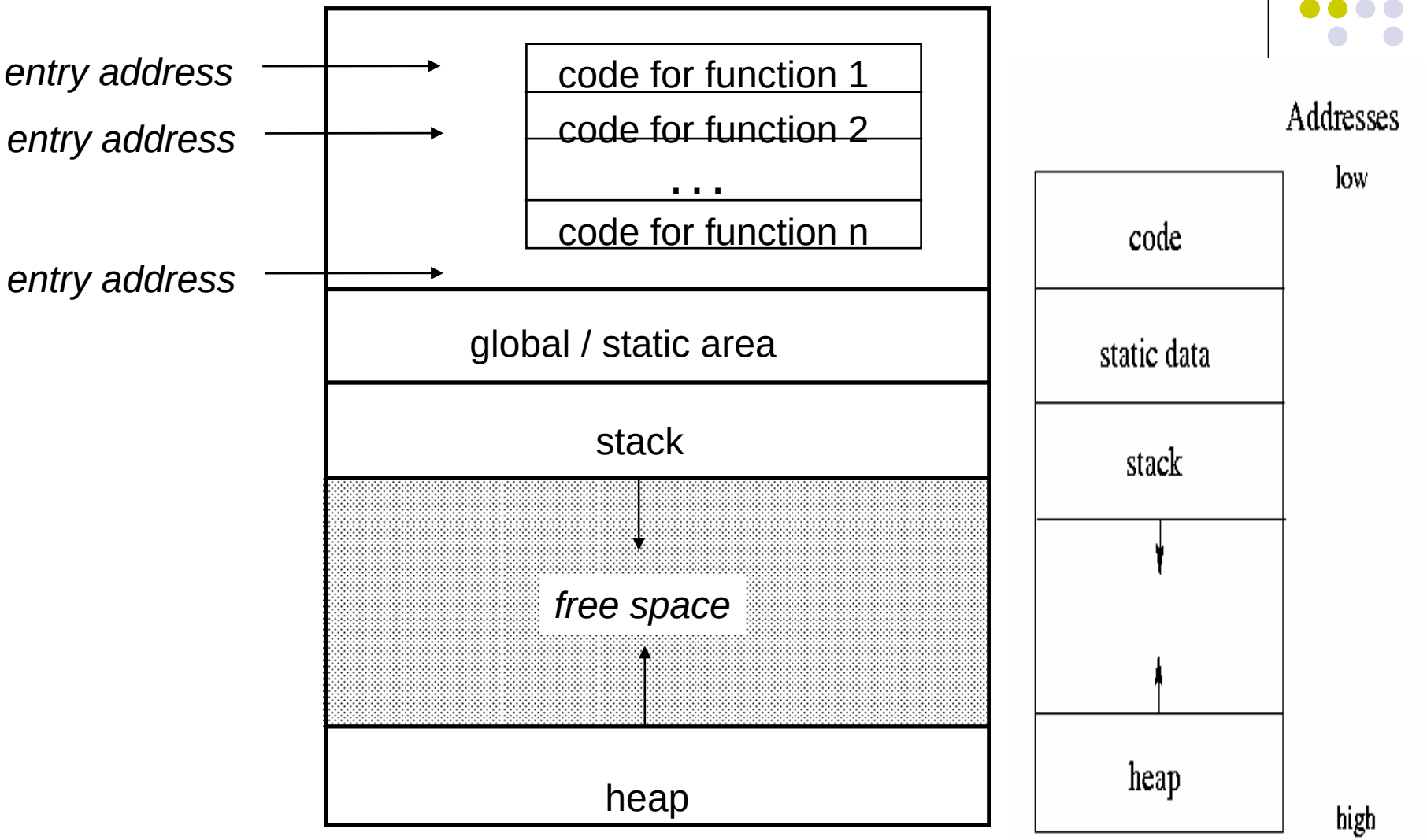
Binding Variable Name



- *Environment* : suatu fungsi yang memetakan suatu nama ke lokasi tempat penyimpanan
- *State* : suatu fungsi yang memetakan lokasi penyimpanan ke nilai yang disimpan.

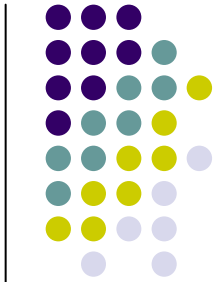
- Suatu assigment akan mengubah state, bukan environment-nya.
  - Misalkan suatu variabel “pi” disimpan pada address 100 mempunyai isi “0”, dan kemudian diisi dengan nilai “3.14” maka tempat penyimpanannya tetap ada di alamat 100.

Runtime Memory Organization



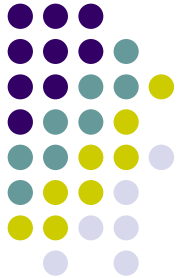
Runtime memory is organized to hold the components of an executing program

Runtime Memory Organization



- Kode target/code for function biasanya sudah dapat ditentukan besar ukurannya, sehingga memori yang dialokasikan dapat ditentukan dengan mudah.
- Obyek data seperti pemesanan variabel-variabel dinamis tetap harus ditentukan pada saat kompilasi.
- Stack menyimpan informasi mengenai aktivasi prosedur.
- Pada waktu suatu pemanggilan prosedur terjadi, eksekusi suatu aktivasi behenti sejenak dan informasi mengenai nilai program counter dan register mesin saat itu disimpan dalam stack. Sewaktu kendali kembali dari pemanggilan prosedur, maka eksekusi ini dapat berjalan kembali dengan cara mengembalikan nilai-nilai register tadi dan menset program counter ke kode setelah pemanggilan prosedur itu.
- Heap menyimpan informasi - informasi lainnya, Besarnya stack dan heap dapat berubah-ubah pada saat runtime.

Procedure Activation Record



activation record :	*functionX*
par1 (type) :	<value>
...	
parN (type):	<value>
start code ptr:	<ptr>
current ptr:	<ptr>
return address:	<ptr>
var1 (type) :	<value>
...	
varN (type):	<value>

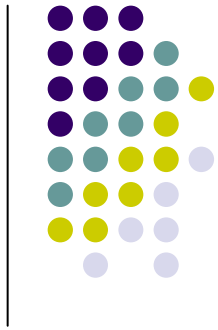
Each time the flow of control enters a function or procedure, we update its ***procedure activation record***.

This maintains the **values** of the function arguments and all **local variables** defined inside the function, and pointers to the **start** of the code segment, the **current** location in the code segment, and the segment of code to which we **return** on exit.

Penjelasan Record Activation

- Record aktivasi berisi informasi yang dibutuhkan untuk mengatur aktivasi sebuah prosedur yang terdiri dari beberapa field:
  - Return value (nilai kembalian), dari prosedur yang dipanggil ke prosedur pemanggilnya.
  - Parameter aktual, yang dipakai oleh prosedur pemanggil untuk memberikan parameter kepada prosedur yang dipanggil.
  - Link kendali aktivasi, yang menunjuk pada record aktivasi pemanggil
  - Link akses pemanggil, yang berguna sebagai tempat untuk menyimpan data non lokal milik record aktivasi lain.
  - Address
    - Current adress
    - Return address
    - Start address, dan lain-lain

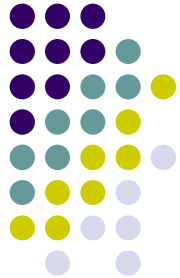
Record Activation



- Status mesin, menyimpan informasi tentang state mesin itu tepat sebelum prosedur dipanggil. Informasi ini terdiri dari:
  - nilai dari counter program dan
  - register mesin yang harus dikembalikan sewaktu kembali dari prosedur yang dipanggil.
- Data lokal, yang dipakai menyimpan data variabel lokal pada suatu pengeksekusian prosedur.
- Data sementara (temporary value), seperti nilai yang muncul pada saat evaluasi suatu ekspresi.

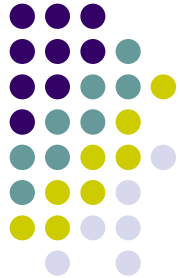
- Besarnya masing-masing field-field ditentukan pada saat suatu prosedur dipanggil, pada saat kompilasi.
  - Pengecualian misalnya pada saat pembuatan variabel array dinamis yang hanya bisa dialokasikan pada saat runtime.
  - Pengecualian pada pembuatan pointer

Ukuran Variabel Temporer



- Ukuran penyimpanan yang dibutuhkan disesuaikan dengan tipenya. Misal tipe data byte : 1 byte.
- Susunan penyimpanan sangat tergantung dari mesin target.
  - Misalnya instruksi untuk menambah (ADD) harus ditempatkan pada posisi **tertentu** di memori pada suatu mesin target **tertentu**
  - Misalnya array sepuluh karakter sebenarnya cukup ditempatkan pada 10 byte saja, namun kompiler harus menyediakan 12 byte, dan 2 byte tidak terpakai.
    - Hal semacam ini disebut padding. Namun jika kekurangan tempat, kompiler akan melakukan pack data.

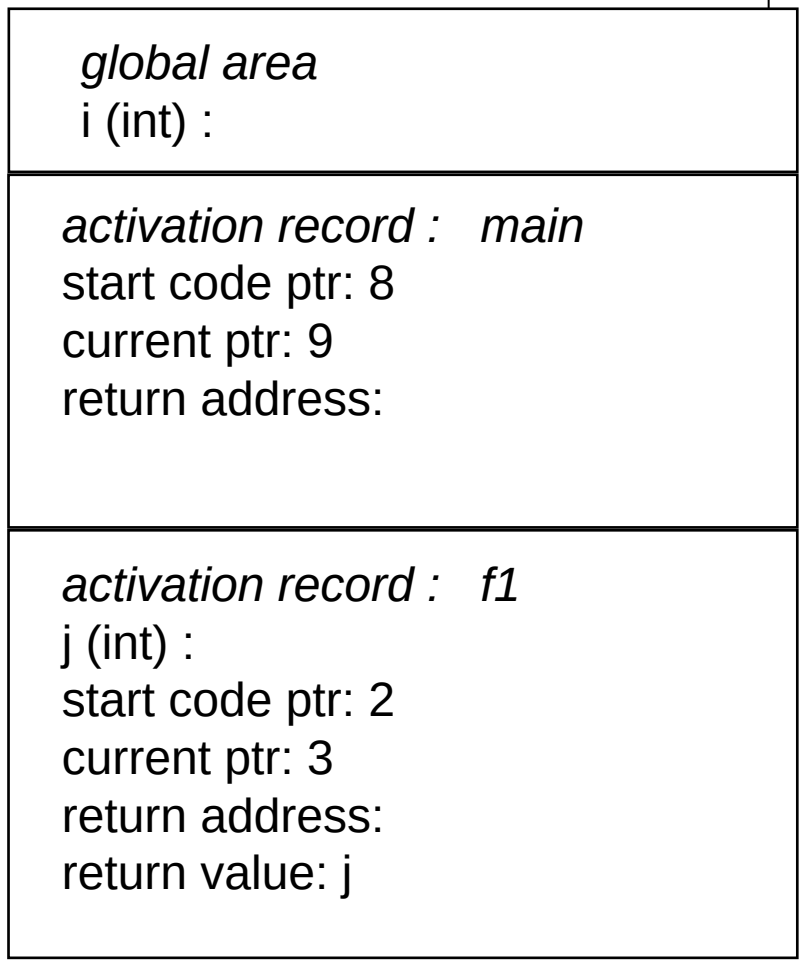
Strategi Penyimpanan Runtime Environment



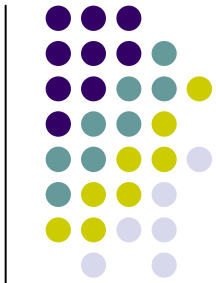
- Alokasi Full - Statis
  - contoh: Fortran
  - semua dialokasikan oleh kompiler pada waktu kompilasi
  - Tidak mendukung pointer
  - Tidak membutuhkan stack
  - tidak ada rekursi dan alokasi memori secara dinamis
  - digunakan untuk: variabel global, konstanta
  - Berarti pada suatu saat hanya boleh ada satu record aktivasi yang aktif

Static example

```
1  int i = 10;

2  int f1(int j) {
3     int k;
4     k = 3 * j;
5     if (k<i) return(i);
6     else return(j);
7  }

8  main() {
9     int k = 1;
10    while (k<5) {
11       f1(k);
12       k = k+1;
13    }
14 }
```



Strategi Penyimpanan

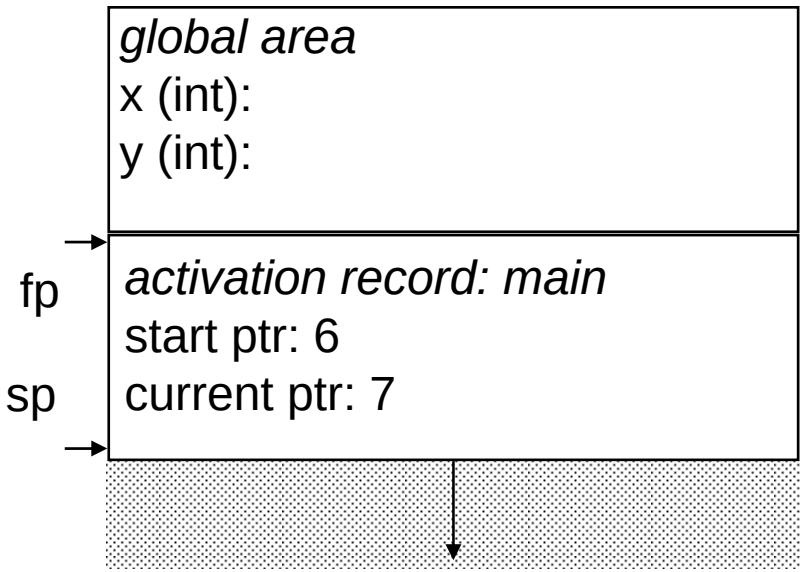


- Alokasi Stack
  - contoh: high level language= C, C++, Pascal
  - tempat penyimpanan diatur sebagai suatu stack
  - bisa digunakan untuk prosedur rekursif dan pointer (alokasi dinamis)
  - Pada suatu saat boleh banyak record aktivasi aktif
  - record aktivasi dibuat pada saat dibutuhkan, dipush ke stack pada saat dipanggil dan dipop dari stack pada saat selesai pemanggilan.

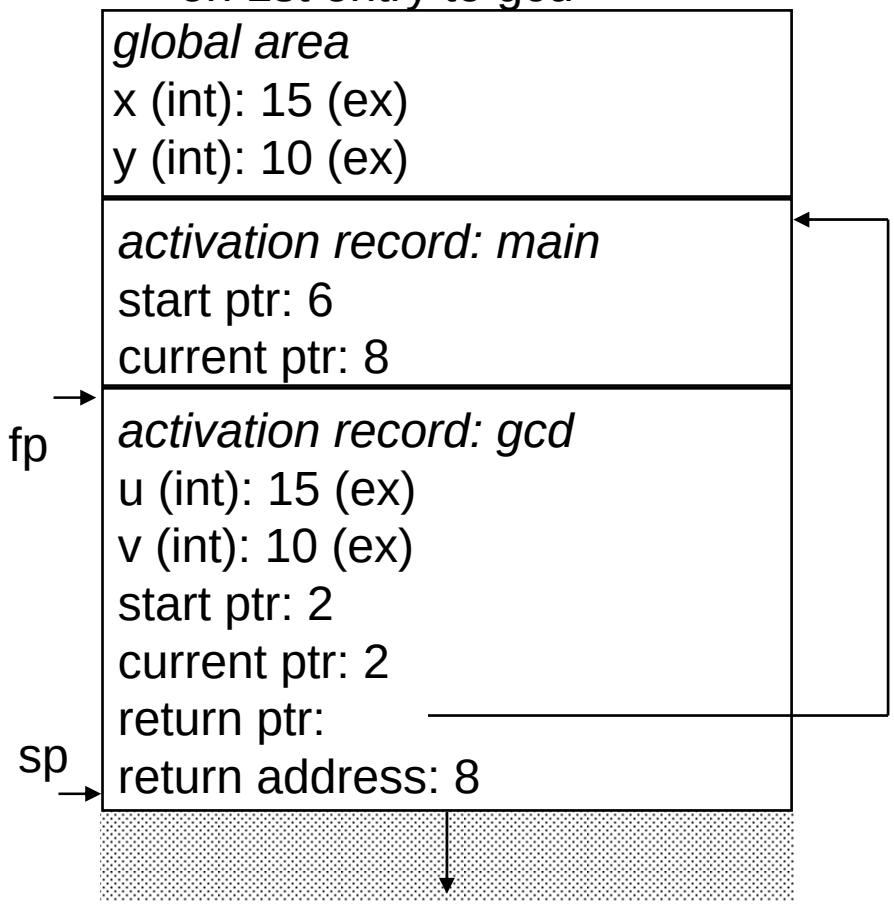
Stack-based Example

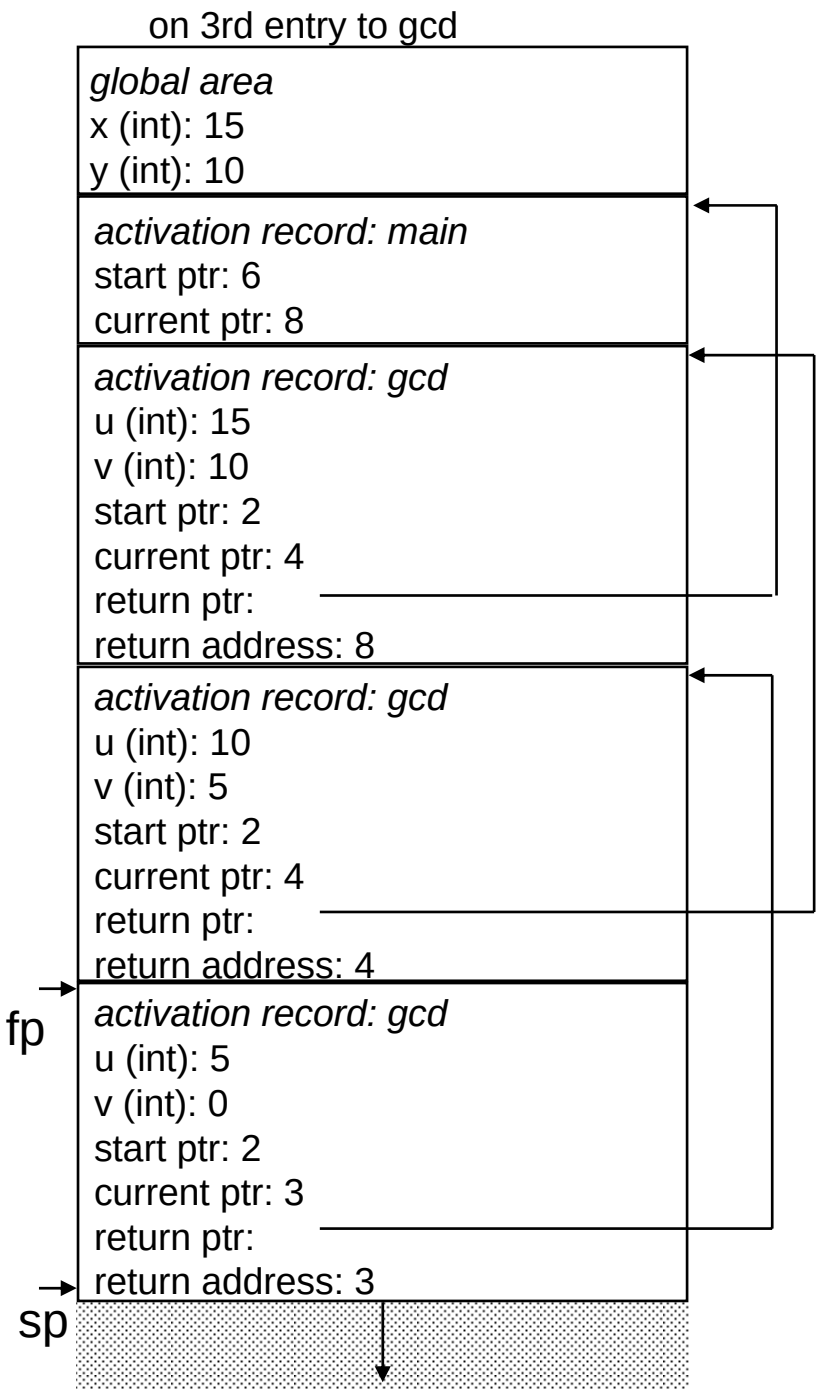
```
1   int x, y;

2   int gcd(int u, int v) {
3      if (v == 0) return u;
4      else return gcd(v, u % v);
5   }

6   main() {
7      scanf("%d %d", &x, &y);
8      printf("%d\n", gcd(x, y));
9   }
```

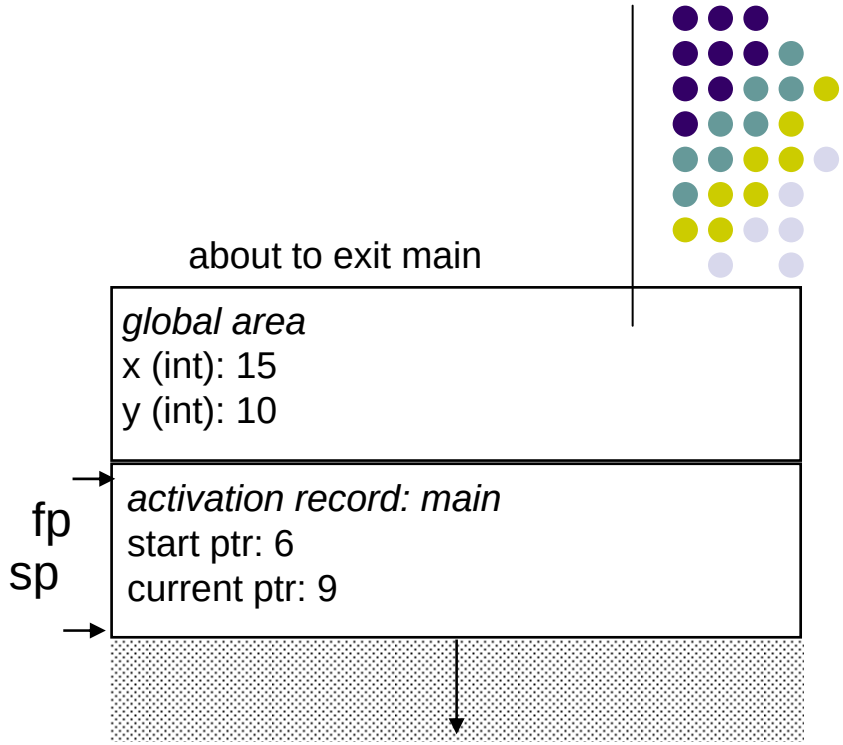
initial environment



on 1st entry to gcd



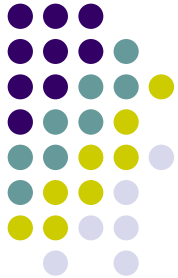

on 3rd entry to gcd
global area
x (int): 15
y (int): 10
activation record: main
start ptr: 6
current ptr: 8
activation record: gcd
u (int): 15
v (int): 10
start ptr: 2
current ptr: 4
return ptr:
return address: 8
activation record: gcd
u (int): 10
v (int): 5
start ptr: 2
current ptr: 4
return ptr:
return address: 4
fp
activation record: gcd
u (int): 5
v (int): 0
start ptr: 2
current ptr: 3
return ptr:
return address: 3
sp


about to exit main
global area
x (int): 15
y (int): 10
fp
sp
activation record: main
start ptr: 6
current ptr: 9

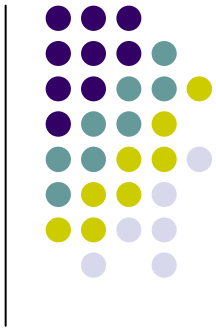
Dangle Function (berkesinambungan)

```
int *dangle() {
   int x = 3;
   return &x;
}

int inc(int y) {
   int z;
   z = y + 1;
   return z;
}

main() {
   int *w, t;

   w = dangle();
   printf("w = %d\n", *w);
   *w = inc(10);
   printf("w = %d\n", *w);
}
```

Hasil:
W = 3
W = 11

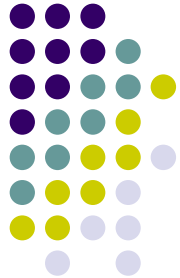


Strategi Penyimpanan



- Alokasi **Heap**
  - contoh: Lisp, Java dan Scheme
  - mengalokasikan dan mendealokasikan tempat penyimpanan seperlunya.
  - Biasanya menggunakan heap (karena dinamis)
  - dibutuhkan garbage collection dan compaction untuk memperoleh kembali space yang dibutuhkan.
  - Beberapa kemungkinan dealokasi:
    - No Dealocation: stop ketika sudah out of space
    - Explicit Dealocation: free(); dispose();
    - Implicit Dealocation: garbage collection
  - contoh coding:
    - `fungsi(int n) {`
      - `float a[n], b[n*10];`
    - `… }`

Heap Management



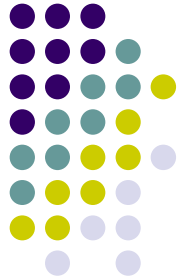
Stack-based languages like C also require a heap to maintain pointer allocation and dynamically allocated memory.

	C	C++	Java
allocate	`malloc`	`new`	`new`
free	`free`	`delete`	•

The main challenges in heap management is:

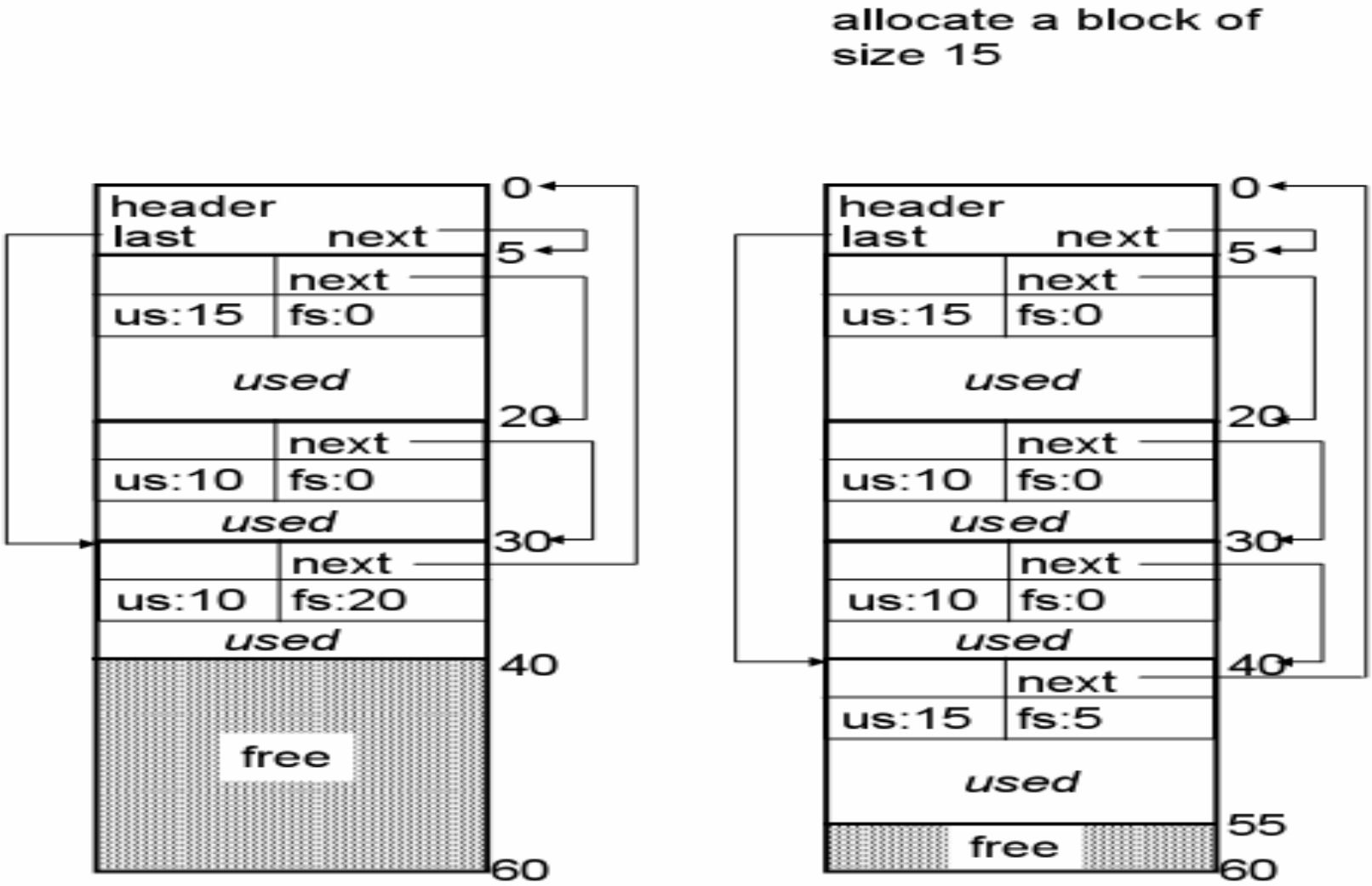
- stopping the heap becoming fragmented, by combining successive freed blocks into one

a simple heap management scheme



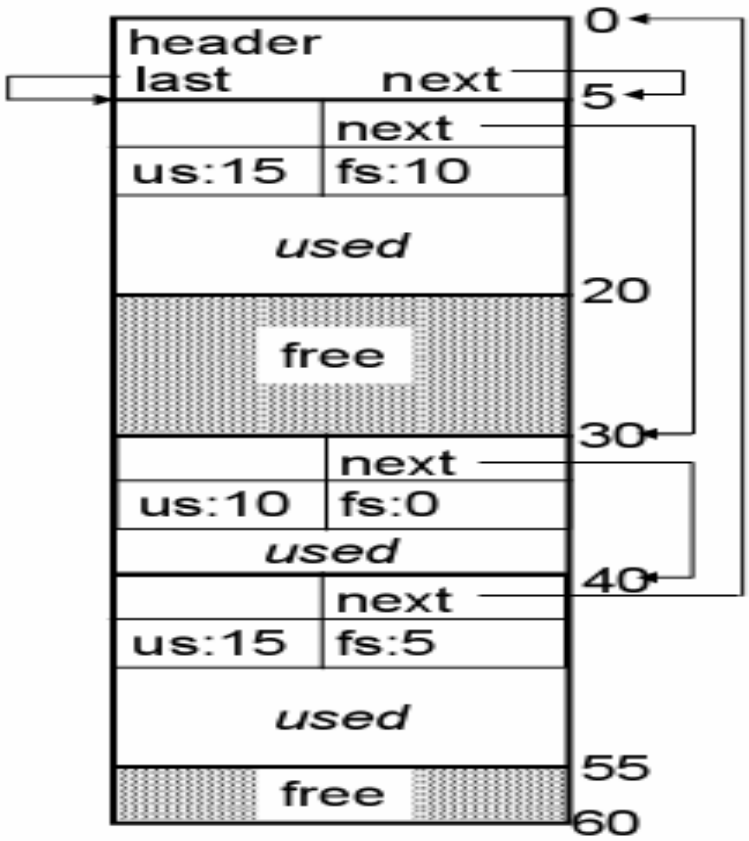
- circular linked list of allocated memory blocks
- each block records its used size, the size of the free space after it, and points to the next block
- to allocate a block
  - find a block with enough free space after it
  - jump to end of that block's used space
  - insert the new block into the heap
  - compute the reduced free space
  - set the previous block's free space to 0
  - update the list
- to free a block
  - move to top of the heap
  - step through list of blocks until required address
  - add the block's free size and used size to the free size of its previous block

Example allocation sequence



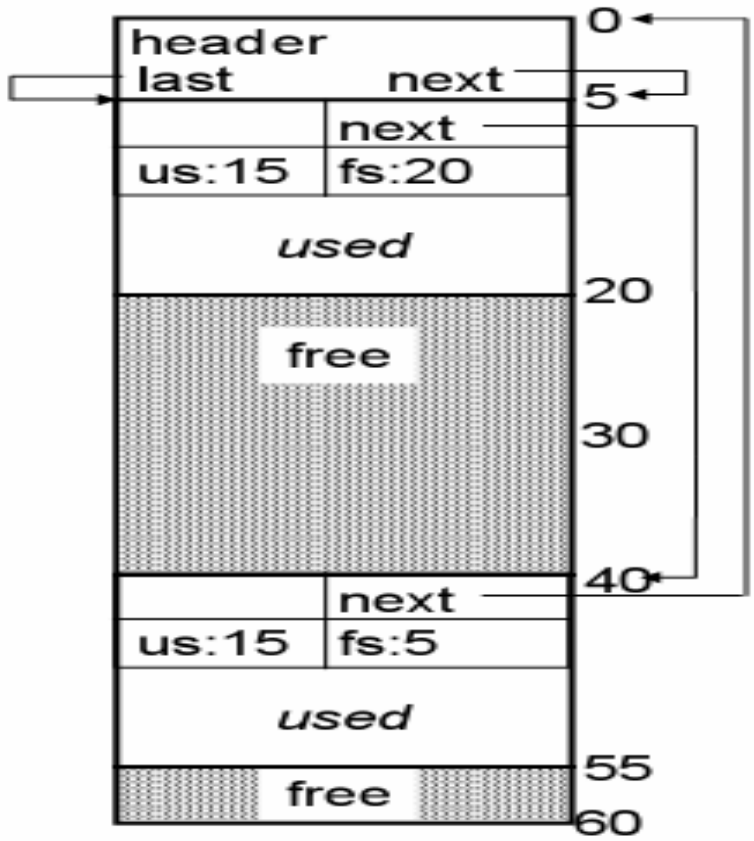
Example allocation sequence

free block (size 10)
at 20

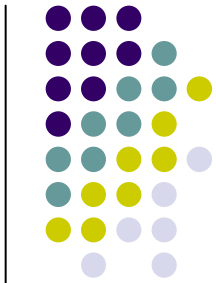


free block (size 10)
at 35 - **fails**

free block (size 10)
at 30

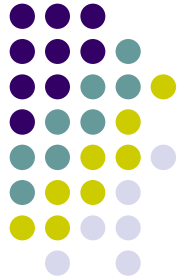


Procedure Calls and Returns



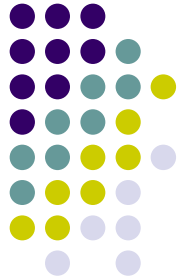
- *Calling sequence*: menangani pemanggilan prosedur
- Caller (pemanggil) dan callee (tubuh prosedur):
  - Caller me-load parameter aktual
  - Caller menyimpan state mesin dan alamat kembalian
  - Caller melompat ke alamat pemanggilnya
  - Callee mengalokasikan record aktivasi dan informasi ke dalam field-fieldnya
  - Callee menginisialisasi local data dan mulai mengeksekusi

Procedure Calls and Returns

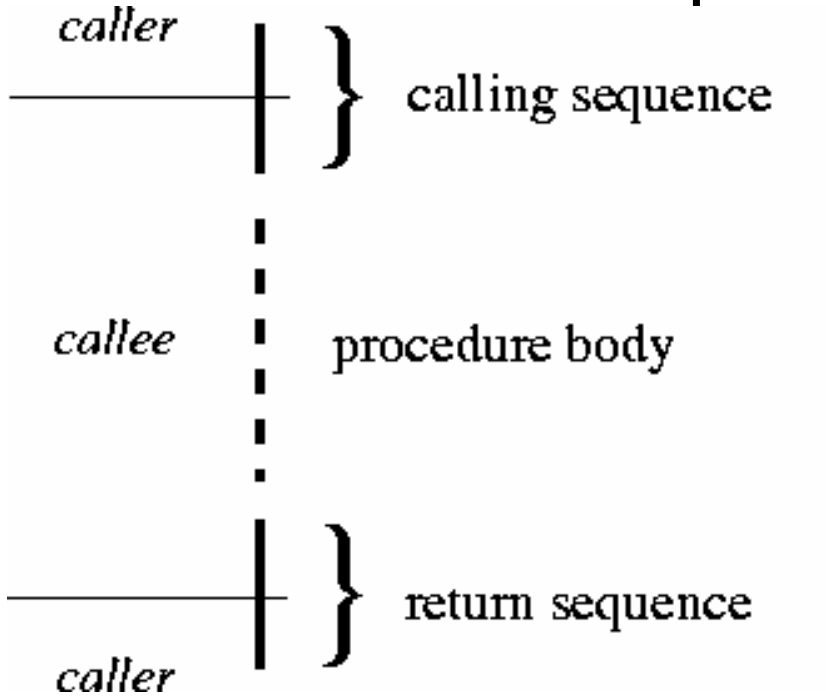


- *Return sequence*: menangani kembalian dari sebuah pemanggilan prosedur:
  - Caller me-load nilai kembalian
  - Calle mendealokasikan record aktivasi
  - Caller me-restore state mesin (nilai register yang telah disimpan, nilai program counter yang telah disimpan)
  - Caller mengkopikan hasil kembalikan ke variabelnya sendiri

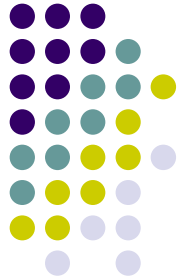
Procedure Calls and Returns: cont'd



Structure of code executed for a procedure call:

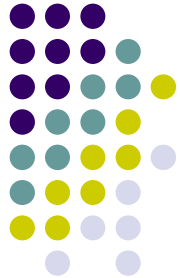


Non fixed Variable size strategy



- Misalkan kita memiliki 3 array dinamis, maka kita harus menyediakan 3 pointer yang masing-masing menunjuk pada alamat pertama masing-masing array dan yang disimpan pada record aktivasi adalah pointer-pointernya saja.

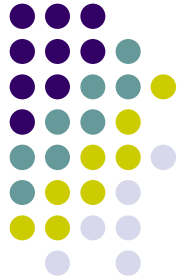
Nested Functions



- Beberapa bahasa pemrograman mengizinkan adanya prosedur/fungsi berkalang.
- Nested functions biasanya bisa mengakses variable pada scope tertutup.
- Contoh:

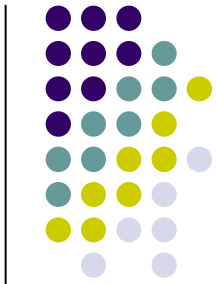
```
int foo(int x, int y)
{
  int bar(int x) { return x*y; }
  return bar(x) + bar(y);
}
```

Nested Functions



- Problem:
  - Pada nested function variabel kadang bisa mengakses scope yang tidak seharusnya
  - Kita tidak pernah tahu seberapa dalam stack yang ada dan scope variabel yang ada.

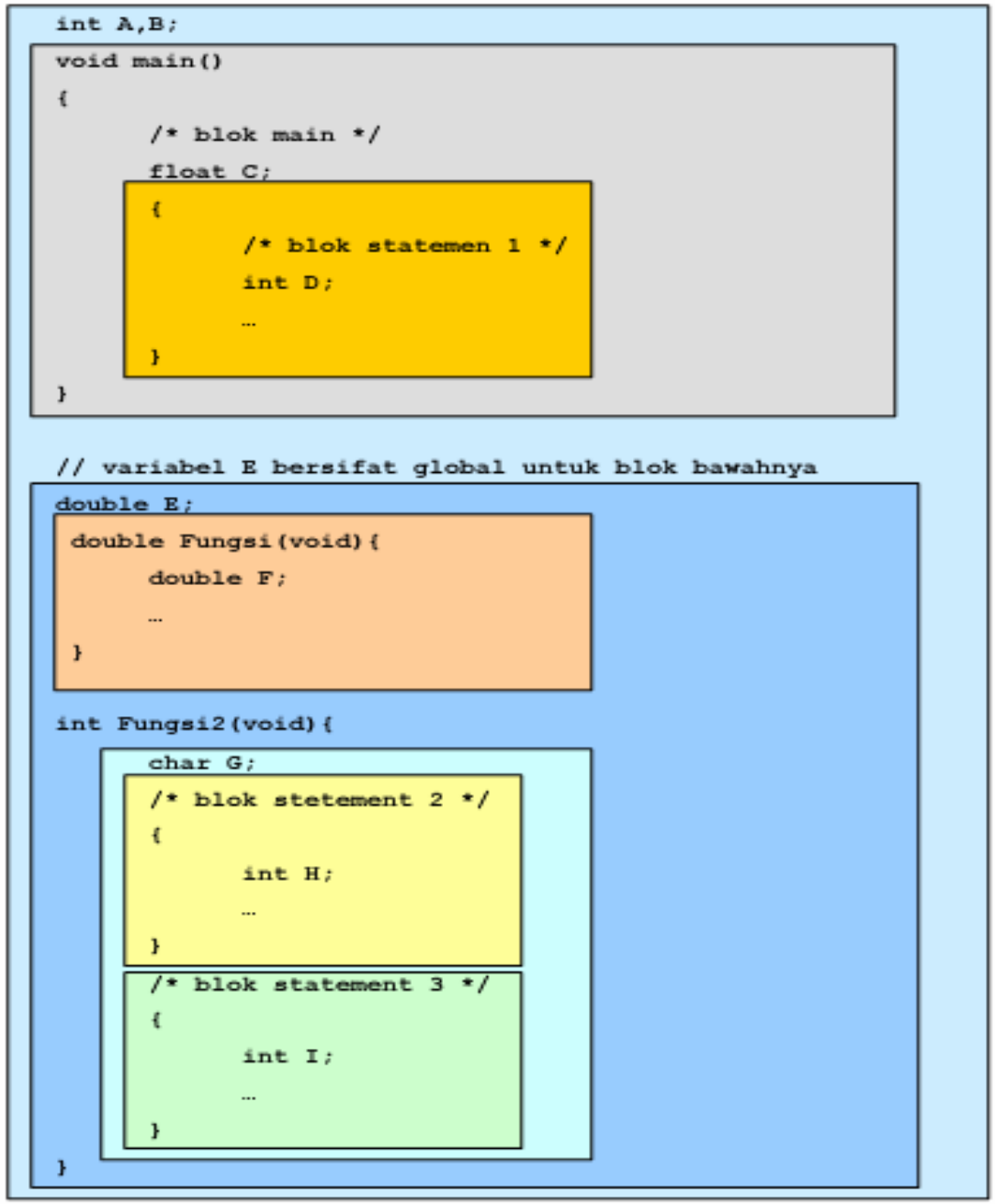
Accessing non local variable



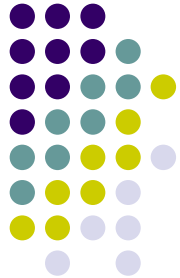
Contoh:

```
int p(int m)
{
    int x;  /* x is local to p, hence in p's
activation record */
    int q(int n)
    {
        if (n > 0) return 2*q(n-1);
        else return x+1;
  }
      printf("%d\n", q(m+2));
}
```

Using block


int A,B;
void main()
{
/* blok main */
float C;
{
/* blok statemen 1 */
int D;
…
}
}
// variabel E bersifat global untuk blok bawahnya
double E;
double Fungsi(void){
double F;
…
}
int Fungsi2(void){
char G;
/* blok stetement 2 */
{
int H;
…
}
/* blok statement 3 */
{
int I;
…
}
}

Parameter Passing of Function



Function:

- Void
- Non-void

Parameter:

- Nama-nama yang muncul pada deklarasi prosedur, disebut parameter formal
- Variabel dan ekspresi yang dimasukkan ke prosedur disebut parameter aktual atau disebut juga argumen

Parameter Review

```
#include <stdio.h>
int JUMLAH(int X, int Y);

void main(){
    int A,B,T;
    A=5; B=2;
    T = JUMLAH(A,B);
    printf("%d",T);
}

int JUMLAH(int X, int Y){
    int H;
    H = X + Y;
    return(H);
}
```

X, Y disebut parameter formal

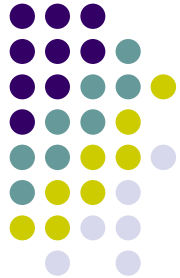
Variabel A,B,C lokal dalam main

A dan B disebut parameter aktual

X, Y disebut parameter formal

Variabel X,Y lokal dalam JUMLAH

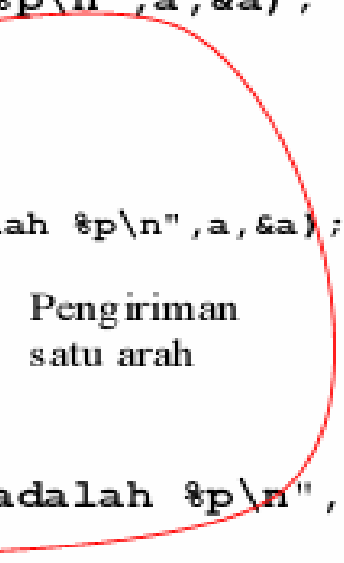
Mode Passing Parameter



- **Call by value**
  - Parameter aktual dievaluasi dan return valuenya di keluarkan ke prosedur pemanggil.
  - Pengubahan nilai variabel hanya akan berefek secara lokal saja dalam prosedur itu
  - Parameter di C, Pascal secara default bersifat “by value”.
  - Yang dikirimkan ke fungsi adalah nilainya, bukan alamat memori letak dari datanya
  - Fungsi yang menerima kiriman nilai ini akan menyimpannya di alamat terpisah dari nilai aslinya yang digunakan oleh program yang memanggil fungsi tersebut
  - Karena itulah pengubahan nilai di dalam fungsi tidak akan berpengaruh pada nilai asli di program yang memanggil fungsi walaupun keduanya menggunakan nama variabel yang sama
  - Pengiriman by value adalah pengiriman searah, dari program pemanggil fungsi ke fungsi yang dipanggilnya
  - Pengiriman by value dapat dilakukan untuk suatu statement, tidak hanya untuk suatu variabel, value, array atau konstanta saja

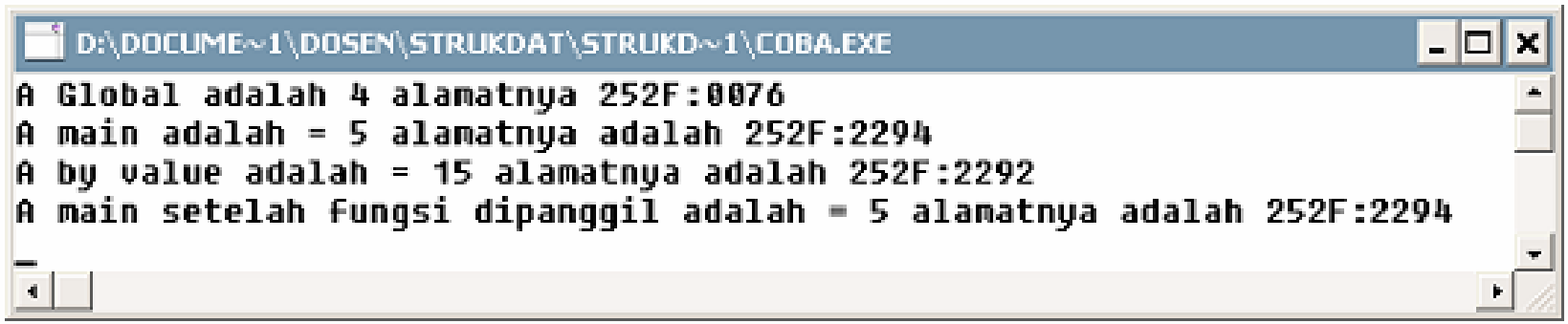
Call by value

```
int a=4;

void getAGlobal(){
     printf("A Global adalah %d alamatnya %p\n",a,&a);
}

void fungsi_by_value(int a){
     a = a * 3;
     printf("A by value adalah = %d alamatnya adalah %p\n",a,&a);
}

void main(){
     int a = 5;
     getAGlobal();
     printf("A main adalah = %d alamatnya adalah %p\n",a,&a);
     fungsi_by_value(a);
     printf("A main setelah fungsi dipanggil adalah = %d
alamatnya adalah %p\n",a,&a);
     getch();
}
```



Call by value

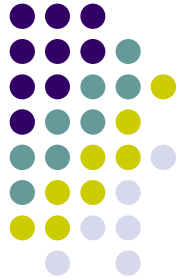
Hasil:



Di dalam Memori:

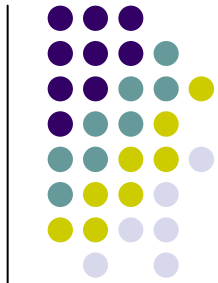
a di *global* nilai **4** alamat **252F:0076**	a di *main* nilai **5** alamat **252F:2294**	a di *fungsi_by_value* nilai **15** alamat **252F:2292**	a di *main after function* nilai **5** alamat **252F:2294**

Mode Passing Parameter



- Call by reference
  - Disebut juga call-by-address atau call-by-location.
  - Jika parameter berupa ekspresi, ekspresi dievaluasi di lokasi baru, dan kemudian alamat nilai tersebut akan dikembalikan.
  - Pengubahan nilai variabel akan berpengaruh tidak hanya di dalam prosedur itu tapi juga diluar prosedur itu.

Call by reference



- Yang dikirimkan adalah alamat memori letak dari nilai datanya, bukan nilai datanya
- Fungsi yang menerima parameter ini akan menggunakan alamat yang sama dengan alamat nilai datanya
- Karena itulah pengubahan nilai di fungsi akan mengubah juga nilai asli di program pemanggil fungsi tersebut
- Pengiriman parameter by reference adalah pengiriman dua arah, yaitu dari program pemanggil fungsi ke fungsi dan sebaliknya dari fungsi ke program pemanggilnya
- Pengiriman parameter by reference tidak dapat digunakan untuk suatu ungkapan, hanya bisa untuk variabel, konstanta atau elemen array saja

Call by reference

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

int a=4;

void getAGlobal(){
     printf("A Global adalah %d alamatnya %p\n",a,&a);
}

void fungsi_by_ref(int *a){
     *a = *a * 3;
     printf("A by ref adalah = %d alamatnya adalah %p\n",*a,a);
}

void main(){
     int a = 5;
     getAGlobal();
     printf("A main adalah = %d alamatnya adalah %p\n",a,&a);
     fungsi_by_ref(&a);
```

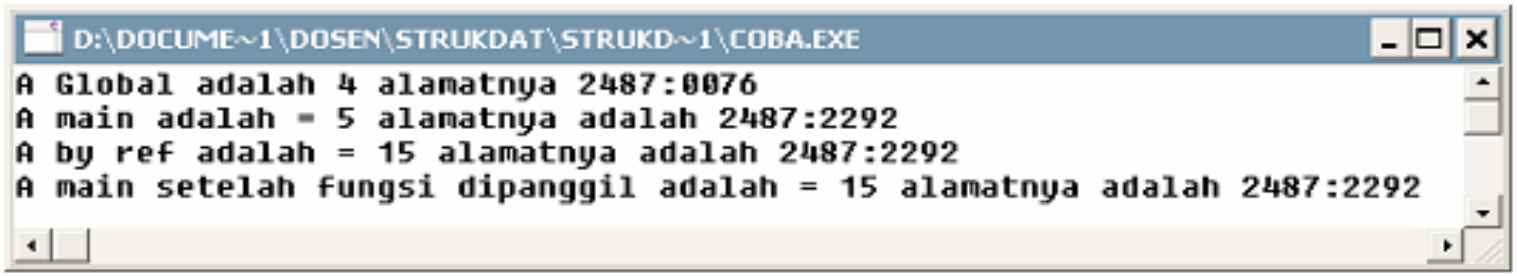
Menggunakan asteris/bintang (*)

Menggunakan asteris/bintang (*)

Call by reference

```
      printf("A main setelah fungsi dipanggil adalah = %d alamatnya
adalah %p\n",a,&a);
      getch();
}
```

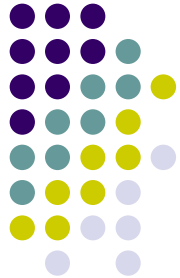
Hasil1:



Di dalam Memori:

a di *global* nilai **4** alamat **2487:0076**	a di *main* nilai **5** alamat **2487:2292**	a di *fungsi_by_value* nilai **15** alamat **2487:2292**	a after *fungsi_by_ref* nilai **15** alamat **2487:2292**

Mode Passing Parameter



Copy-restore

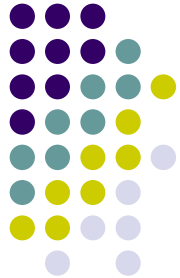
- Sebuah keturunan antara call by value dan call by reference.
- Parameter aktual akan dievaluasi dan kemudian return valuenya akan dilemparkan secara call by value.
- Setelah kendali kembali, nilai dari return valuenya akan dikopikan ke parameter aktual.
- Copy-Restore menghindari problem ketika prosedure memiliki lebih dari satu cara untuk mengakses sebuah variabel.

Contoh

```
#include <stdio.h>
int a;
void fungsi(int  *x)
{
        *x=2; a=0;
}

void main(){
a=1;
fungsi(&a);
printf("%d",a) ;
}
```

- Copy-Restore akan print 2 sedangkan call by reference akan print 0

NEXT



- **Code Optimization**