CENG 442 PROGRAMMING LANGUAGES

Faris Serdar Tasel

Department of Computer Engineering

VARIABLES & STORAGE

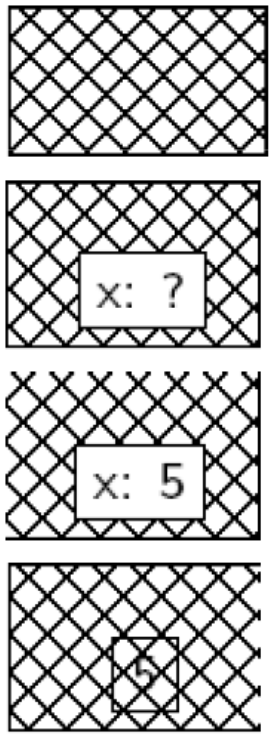
Variables

- Mathematical variables stand for a fixed but unknown value, no change over time
- Functional and logic programming variables behave like mathematical variables
- Imperative (also object-oriented and concurrent) variables are containers for values
  - Inspect
  - Update

Computer Memory

- Computer memory can be considered as a collection of **cells**

- Cells are initially **unallocated**
- Then, **allocated** / **undefined** (ready to use but value unknown)
- Then, **storable**
- After the variable's lifetime ends, again **unallocated**




```
void f()
{
    int x;
    ...
    x=5;
    ...
    return;
}
```

Storable Values

- A **storable value** is one that can be stored in a single storage cell.

- C++ : primitive values and pointers
  - Structures, unions, arrays, objects, functions not storable
- Java : primitive values and pointers to objects
  - Objects themselves are not storable
- Ada : primitive values and pointers
  - Records, arrays, procedures not storable

Simple vs. Composite Variables

- A **simple variable** is a variable that may contain a storable value.
  - Occupy a single storage cell
- A **composite variable** is a variable of a composite type.
  - Occupy a group of contiguous cells

Total vs. Selective Update

- A composite variable may be updated either in a single step or in several steps
  - **Total update** : update all components (composite value)
  - **Selective update** : update only a single component

```
struct Complex { double x,y; } a, b;
...
a=b;                    // Total update
a.x=b.y*a.x;            // Selective update
```

Array Variables

- Different approaches exist in implementation of array variables:
  - **Static arrays**
  - **Dynamic arrays**
  - **Flexible arrays**

Static Arrays

- Array size is fixed at compile time to a constant value or expression
- C Example

```
  #define MAXELS 100
  int a[10];
  double x[MAXELS*10][20];
}
```

Dynamic Arrays

- Array size is defined when variable is allocated and remains constant afterwards
- Example : C with GCC Extension (NOT ANSI!)

```
int f(int n) {
    double a[n]; ...
}
```

- Example: C++ with **Templates**

```
template<class T>  class Array {
      T *content;
  public:
      Array(int s) { content=new T[s]; }
      ~Array()     { delete [] content; }
};
...
Array<int>  a(10);              Array<double> b(n);
```

Flexible Arrays

- Array size is completely variable. Arrays may expand or shrink at run time. Script languages like Perl, PHP, and Python adopt this approach
- PERL Example :

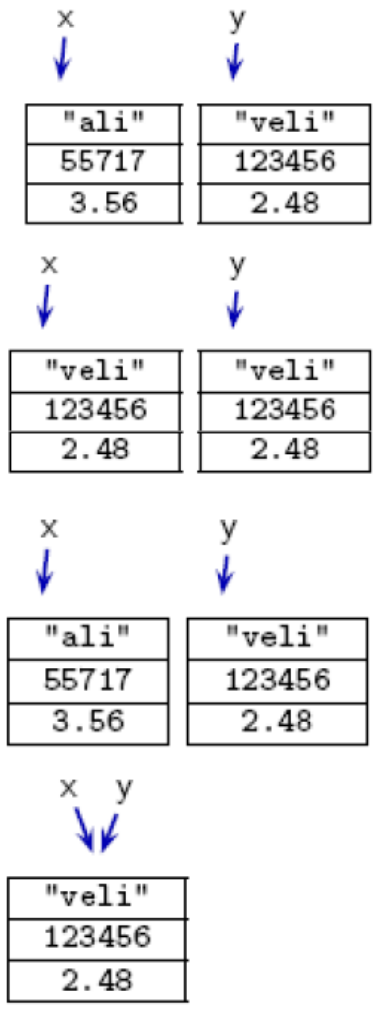
```
@a=(1,3,5);           # array size: 3
print $#a , "\n";     # output: 2 (0..2)
$a[10] = 12;          # array size 11 (intermediate elements
$a[20] = 4;           # array size 21
print $#a , "\n";     # output: 20 (0..20)
delete $a[20];        # last  element erased,  size is 11
print $#a , "\n";     # output: 10 (0..10)
```

- C++ and object orient languages allow overloading of [ ] operator to make flexible arrays possible. STL (Standard Template Library) classes in C++ like **vector**, **map** are like such flexible array implementations.

Semantic of assignment in composite variables

- Two distinct possibilities
  - Copy semantics
  - Reference semantics

- **Copy Semantics** : All content is copied into the other variables storage (two copies with same values in memory)
- **Reference Semantics** : Reference of variable is copied to other variable (two variables share the same storage and values)



Semantic of assignment in composite variables (cont.)

- Assignment semantics is defined by the language design
- Copy semantics is slower
- Reference semantics cause problems from storage sharing (all operations effect both variables).
  Deallocation of one makes the other invalid

Semantic of assignment in composite variables (cont.)

- C
  - structures follow copy semantics
  - arrays cannot be assigned
  - pointers are used to implement reference semantics.
- Java
  - copy semantics for primitive types
  - reference semantics for objects
- Java also provides copy semantic via a member function called clone(). Java garbage collector avoids invalid values (in case of deallocation)

A question

You are given following definitions in C:

```
int a[5] = {10, 20, 30, 40, 50};
int b[5];
```

- Will this work?

  b = a;

A question

You are given following definitions in C:

```
int a[5] = {10, 20, 30, 40, 50};
int b[5];
```

- How to solve it via copy semantics?

  for(int i = 0; i < 5; i++) b[i] = a[i];

- Any other idea?

A question

Remember that C uses copy semantics for structures!

```
struct arrStruc {
    int arr[5];
} a = {10, 20, 30, 40, 50};

struct arrStruc b;

b = a;
```

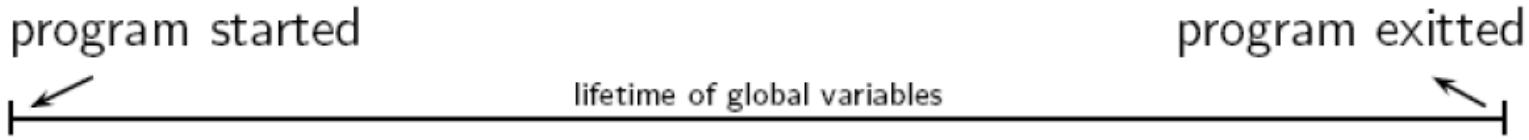
Variable Lifetime

- **Variable Lifetime** : The interval between creation and destruction of a variable

- Classification according to lifetime
  - Global variables (*while program is running*)
  - Local variables (*while declaring block is active*)
  - Heap variables (*arbitrary*)
  - Persistent variables (*continues after program termination*)

Global Variables

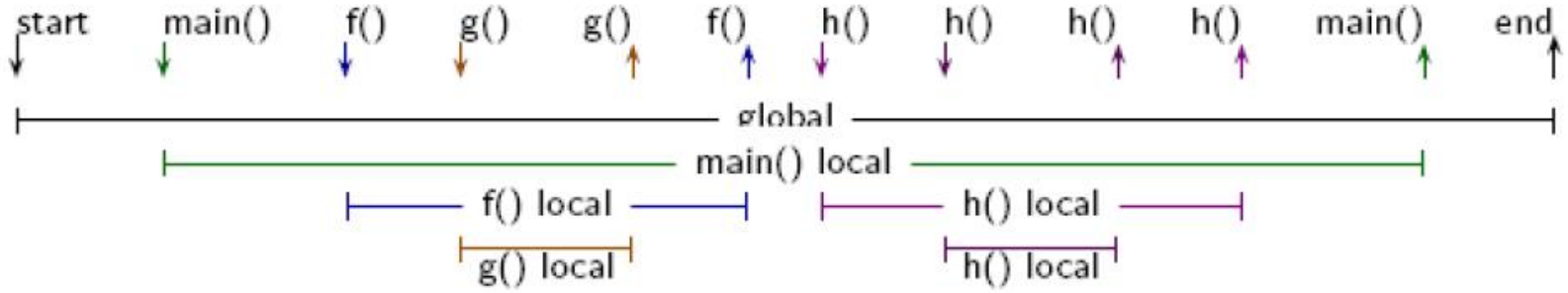
- A **global variable** is created when the program starts, and is destroyed when the program stops.



- C
  - All variables declared NOT inside of a function
    - What about main()?
  - What about static variables inside functions?

Local Variables

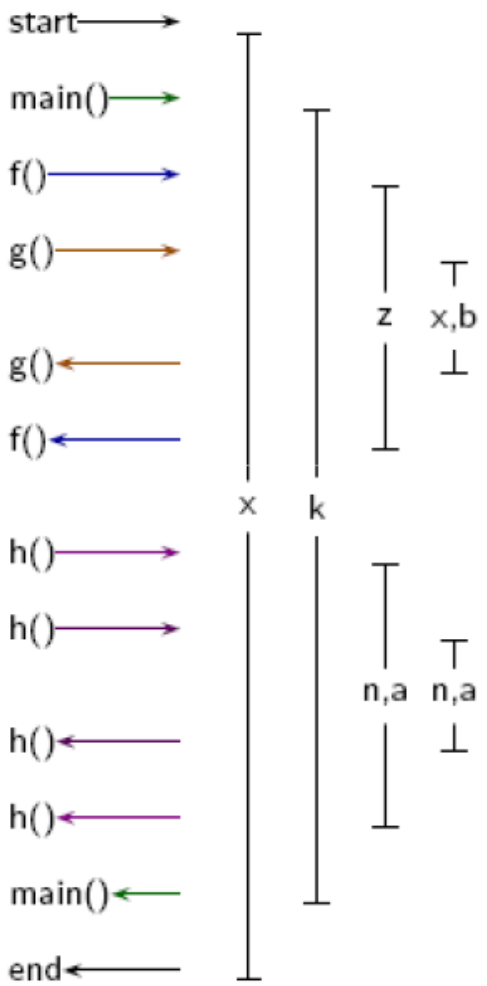
- Lifetime of a **local variable**, a variable declared within a block, is the time between the entry of the block and the exit from block.



- Multiple instances of the same local variable may be alive at the same time in recursive functions
- What about the formal parameters?
  - Formal parameters are local variables

Example

```
double x;
int h(int n) {
   int a;
   if (n<1) return 1
   else return h(n-1);
}
void g() {
   int x;
   int b;
...
}
int f() {
   double z;
   ...
   g();
   ...
}
int main() {
    double k;
    f();
    ...
    h(1);
    ...;
    return 0;
}
```



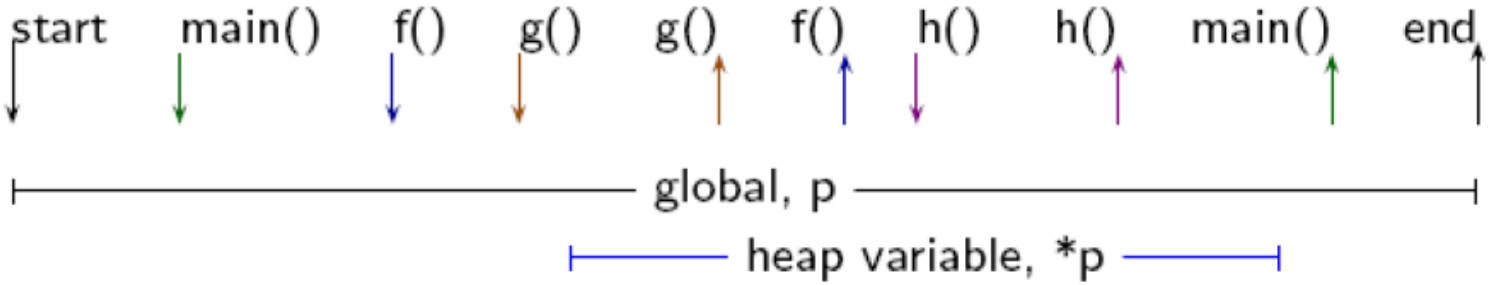
Heap Variables

- Allocation and deallocation of heap variables are not automatic but explicitly requested by programmer via function calls.
- C: malloc(), free()
- C++: new, delete.
- A heap variable's lifetime can start or end at anytime
- Heap variables are accessed via pointers. Some languages use references.

Heap Variables – Example

- p and *p are different variables!
- **p** has pointer type and usually a local or global variable
- ***p** is heap variable

```
double *p;
int h() { ...
}
void g() { ...
   p=malloc(sizeof(double));
}
int f() { ...
   g(); ...
}
int main() { ...
     f();     ...
     h();     ...;
     free(p); ...
}
```



Dangling Reference

- **Dangling reference** : trying to access a variable whose lifetime is ended or already deallocated

```
char *p, *q;

p=malloc(10);
q=p;
...
free(q);
printf("%s",p);
```

```
char *f() {
    char a[]="ali";
    ....
    return a;
}
....
char *p;
p=f();
printf("%s",p);
```

- **p** is deallocated (left); **p**'s lifetime ended (right), thus dangling reference

Garbage variables

- **Garbage variables** : The variables whose lifetime has not ended but there is no way to access

```
char *p, *q;
...
p=malloc(10);
p=q;
...
```

```
void f() {
    char *p;
    p=malloc(10); ...
    return
}
....
f();
```

- When the pointer value is lost or lifetime of the pointer is over, heap variable (*p in examples) is inaccessible.

Garbage Collection

- **Garbage collection** is a solution to dangling reference and garbage problem.
  - PL does management of heap variable deallocation automatically
  - No call like free() or delete exists
  - Count of all possible references is kept for each heap variable.
  - When reference count gets to 0 garbage collector deallocates the heap variable
  - Garbage collector usually works in a separate thread when CPU is idle

Garbage Collection (cont.)

- Garbage collection method is adopted by Java (which is intended for highly robust applications) and most functional languages like Lisp, ML, and Haskell.

- Another solution, which is too restrictive, is adopted by Ada:
  - A reference cannot be assigned to a longer lifetime variable
  - local variable references cannot be assigned to global reference/pointers

Persistent Variables

- A **persistent variable** is one whose lifetime transcends an activation of any particular program.
  - Files, databases, web service objects…
- A **transient variable** is one whose lifetime is bounded by the activation of the program that created it.
  - Global, local, and heap variables
- Persistent variables are stored in secondary storage or external process

Persistent Variables (cont.)

- Only a few experimental language has transparent persistence. In many languages persistence is achieved via IO instructions
  - C files: fopen(), fseek(), fread(), fwrite()
- Object oriented languages has the concept **serialization** : Converting an object into a binary image that can be written on disk or sent to network.
  - This way objects snapshot can be taken, saved, restored and object continue from where it remains.

Commands

- A **command** (*often called* **statement**) is a program construct that will be executed in order to update variables.
- Commands are a characteristic feature of imperative, object-oriented, and concurrent languages
- Expression vs. Command
  - An expressions is a program segment with a value
  - A command (statement) is a program segment without a value but with purpose of altering the state

Commands (cont.)

- Commands may be formed in various ways
  - Assignment
  - Procedure call
  - Block commands
  - Conditional commands
  - Iterative commands

Assignments

- The assignment command typically has the form

  V = E;

  where E an expression which yields a value, V a variable access
- C : “Var = Expr;”
- Pascal : “Var := Expr;”
- Multiple assignment : x = y = z = 0;
- Parallel assignment (PHP, Perl) :
  - list($name, $surname, $no) = split(‘-’, “Serkan-Soylu-219”);
  - list($a, $b) = array($b, $a);
- Assignment with operator:
  - x += 3; x *= 2;

Procedure Call

- A procedure is user defined commands
- Typically has the form

  P(E1, E2, …, En)

- C : function returning void
- Pascal : procedure

- void functName($param_1$, $param_2$, …, $param_n$)

- Usage is similar to functions but call is in a command position (on a separate line of program)

Block Commands

- A **block** is a composition of multiple commands
- Commands enclosed in a block behaves like single command: “if” blocks, loop bodies, …
- Composition may be in
  - Sequential
  - Collateral
  - Concurrent

Sequential vs. Collateral Commands

- **Sequential Commands**
  - {C1 ; C2 ; … ; Cn ; }
  - A command is executed, after it finishes the next command is executed, and so on…
- **Collateral Commands**
  - {C1 , C2 , … , Cn }
  - The order of execution is non-deterministic, compiler or optimizer can choose any order.
  - If commands are independent, effectively deterministic
    {y = 3 , x = x + 1} vs. {x=3 , x = x + 1}
  - Can be executed in parallel

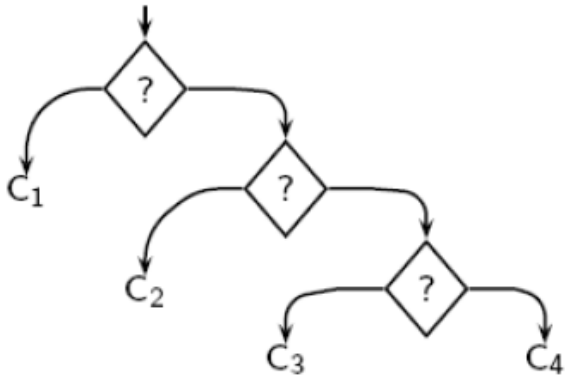
Concurrent Commands

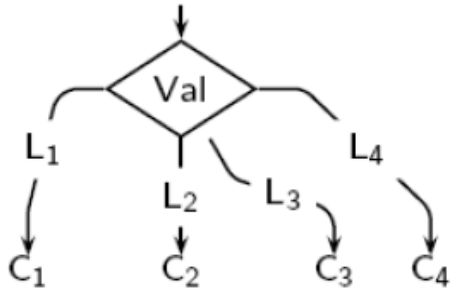
- **Concurrent Commands**
  - {C1 | C2 | … | Cn }
  - All commands start concurrently in parallel. Block finishes when the last active command finishes.
- Real parallelism in multi-core/multi-processor machines
  - Transparently handled by only a few languages
  - Thread libraries required in languages like Java, C, C++

Conditional Commands

- A **conditional command** has two or more subcommands, of which exactly one is chosen to be executed.
- C : if (*cond*) C1 else C2 ;

  switch (*value*) { case L1 : C1 ; case L2 : C2 ; ...}
- If commands can be nested for multi-conditioned selection





Non-deterministic Conditionals

- In case of **non-deterministic conditionals** conditions are evaluated in collaterally and commands are executed if condition holds.

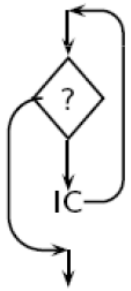
- Hypothetically:
  - if (*cond1*) C1 or if (*cond2*) C2 or if (*cond3*) C3 ;
  - switch (*val*) {case L1: C1 | case L2: C2 | case L3: C3 }

- Tests can run concurrently

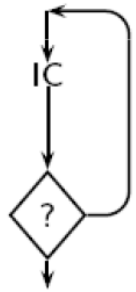
Iterative Commands

- An **iterative command** has a subcommand that is executed repeatedly.
- A classification : minimum number of iteration 0 or 1

```
C:
while ( … )
{
…
}
```




```
C:
do
{
…
} while ( … )
```



- Another classification : definite vs. indefinite iteration

Definite vs. Indefinite iteration

- **Indefinite iteration** : Number of iterations of the loop is not known until loop finishes
  - C loops are indefinite iteration loops

- **Definite iteration** : Number of iterations is fixed when loop is started
  - Pascal for loop is a definite iteration loop

  for i:= k to m do begin .... end;

  has (m − k + 1) iterations

  (Pascal forbids update of the loop index variable)

List and Set Based Iterations

- Languages like PHP, Perl, Python, Shell also have this kind of information

```
$colors=array('yellow','blue','green','red','white');
foreach ($colors as $i) {
    print $i," is a color","\n";
}
```