Programming Languages

Variables and Storage

Onur Tolga Şehitoğlu

Computer Engineering

Outline

Storage

- Functional language variables: math like, defined or solved. Remains same afterwards.

Storage

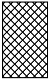
- Functional language variables: math like, defined or solved. Remains same afterwards.
- Imperative language variables: variable has a state and value. It can be assigned to different values in same phrase.

Storage

- Functional language variables: math like, defined or solved. Remains same afterwards.
- Imperative language variables: variable has a state and value. It can be assigned to different values in same phrase.
- Two basic operations on a variable: inspect and update.

Computer memory can be considered as a collection of cells.

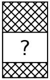
- Cells are initially unallocated.



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

Computer memory can be considered as a collection of cells.

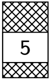
- Cells are initially unallocated.
- Then, allocated/undefined. Ready to use but value unknown.




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

Computer memory can be considered as a collection of cells.

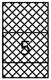
- Cells are initially unallocated.
- Then, allocated/undefined. Ready to use but value unknown.
- Then, storable




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

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 including block terminates, again unallocated




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

Total or Selective Update

- Composite variables can be inspected and updated in total or selectively
-
```
struct Complex { double x,y; } a, b;
...
a=b;                    // Total update
a.x=b.y*a.x;            // Selective update
```
- Primitive variables: single cell
  Composite variables: nested cells

Array Variables

Different approaches exist in implementation of array variables:

1. Static arrays
2. Dynamic arrays
3. 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. Remains constant afterwards.
- Example: C90/GCC (not in 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, Python
- Perl example:

```
@a=(1,3,5);              # array size: 3
print $#a , "\n";        # output: 2 (0..2)
$a[10] = 12;             # array size 11 (intermediate elements ur
$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)
```

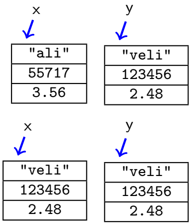
- C++ and object orient languages allow overload of [] operator to make flexible arrays possible. STL (Standard Template Library) classes in C++ like vector, map are like such flexible array implementations.

Semantic of assignment in composite variables

- Assignment by Copy vs Reference.

Semantic of assignment in composite variables

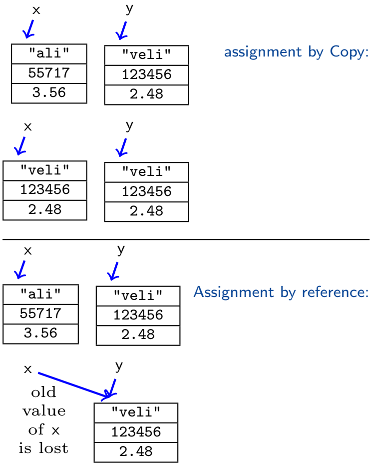
- Assignment by Copy vs Reference.
- Copy: All content is copied into the other variables storage. Two copies with same values in memory.



assignment by Copy:

Semantic of assignment in composite variables

- Assignment by Copy vs Reference.
- Copy: All content is copied into the other variables storage. Two copies with same values in memory.
- Reference: Reference of variable is copied to other variable. Two variables share the same storage and values.



- Assignment semantics is defined by the language design
- C structures follows copy semantics. Arrays cannot be assigned. Pointers are used to implement reference semantics. C++ objects are similar.
- Java follows copy semantics for primitive types. All other types (objects) are reference semantics.
- Copy semantics is slower
- Reference semantics cause problems from storage sharing (all operations effect both variables). Deallocation of one makes the other invalid.
- Java provides copy semantic via a member function called `copy()`. Java garbage collector avoids invalid values (in case of deallocation)

Variable Lifetime

- Variable lifetime: The period between allocation of a variable and deallocation of a variable.
- 4 kinds of variable lifetime.
  1. Global lifetime (while program is running)
  2. Local lifetime (while declaring block is active)
  3. Heap lifetime (arbitrary)
  4. Persistent lifetime (continues after program terminates)

Global lifetime

- Life of global variables start at program startup and finishes when program terminates.

Global lifetime

- Life of global variables start at program startup and finishes when program terminates.
- In C, all variables not defined inside of a function (including `main()`) are global variables and have global lifetime:

Global lifetime

- Life of global variables start at program startup and finishes when program terminates.
- In C, all variables not defined inside of a function (including `main()`) are global variables and have global lifetime:

- What are `static` variables inside functions in C?

Local lifetime

- Lifetime of a local variable, a variable defined in a function or statement block, is the time between the declaring block is activated and the block finishes.

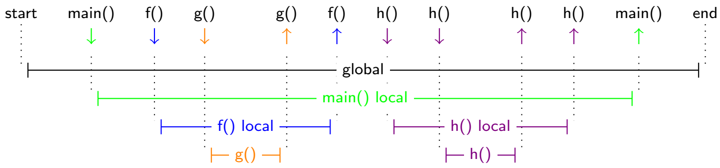
Local lifetime

- Lifetime of a local variable, a variable defined in a function or statement block, is the time between the declaring block is activated and the block finishes.
- Formal parameters are local variables.

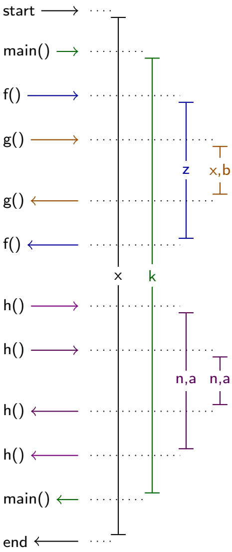
Local lifetime

- Lifetime of a local variable, a variable defined in a function or statement block, is the time between the declaring block is activated and the block finishes.
- Formal parameters are local variables.
- Multiple instances of same local variable may be alive at the same time in recursive functions.

Local lifetime

- Lifetime of a local variable, a variable defined in a function or statement block, is the time between the declaring block is activated and the block finishes.
- Formal parameters are local variables.
- Multiple instances of same local variable may be alive at the same time in recursive functions.
- 

```
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 Variable Lifetime

- Heap variables: Allocation and deallocation is not automatic but explicitly requested by programmer via function calls.

Heap Variable Lifetime

- Heap variables: Allocation and deallocation is not automatic but explicitly requested by programmer via function calls.
- C: `malloc()`, `free()`, C++: `new`, `delete`.

Heap Variable Lifetime

- Heap variables: Allocation and deallocation is not automatic but explicitly requested by programmer via function calls.
- C: `malloc()`, `free()`, C++: `new`, `delete`.
- Heap variables are accessed via pointers. Some languages use references

```
double *p;
p=malloc(sizeof(double));
*p=3.4; ...
free(p);
```

Heap Variable Lifetime

- Heap variables: Allocation and deallocation is not automatic but explicitly requested by programmer via function calls.
- C: `malloc()`, `free()`, C++: `new`, `delete`.
- Heap variables are accessed via pointers. Some languages use references

```
double *p;
p=malloc(sizeof(double));
*p=3.4; ...
free(p);
```

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

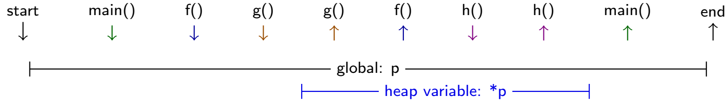
Heap Variable Lifetime

- Heap variables: Allocation and deallocation is not automatic but explicitly requested by programmer via function calls.
- C: `malloc()`, `free()`, C++: `new`, `delete`.
- Heap variables are accessed via pointers. Some languages use references

```
double *p;
p=malloc(sizeof(double));
*p=3.4; ...
free(p);
```

- p and *p are different variables p has pointer type and usually a local or global lifetime, *p is heap variable.
- heap variable lifetime can start or end at anytime.

```
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 and 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);
```

- both p's are deallocated or ended lifetime variable, thus dangling reference
- sometimes operating system tolerates dangling references. Sometimes generates run-time erros like "protection fault", "segmentation fault" are generated.

Garbage variables

- garbage variables: The variables with lifetime still continue 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 is unaccessible. (*p in examples)

Garbage collection

- A solution to dangling reference and garbage problem: PL does management of heap variable deallocation automatically. This is called garbage collection. (Java, Lisp, ML, Haskell, most functional languages)
- no call like `free()` or `delete` exists.
- Language runtime needs to:
  - Keep a reference counter on each reference, initially 1.
  - Increment counter on each new assignment
  - Decrement counter at the end of the reference lifetime
  - Decrement counter at the overwritten/lost references
  - Do all these operations recursively on composite values.
  - When reference count gets 0, deallocate the heap variable

- Garbage collector deallocates heap variables having a reference count 0.
- It usually works in a separate thread, in low priority, works when CPU is idle.
- Another but too restrictive solution to garbage: Reference cannot be assigned to a longer lifetime variable. local variable references cannot be assigned to global reference/pointer.

Persistent variable lifetime

- Variables with lifetime continues after program terminates: file, database, web service object,...
- Stored in secondary storage or external process.
- Only a few experimental language has transparent persistence. Persistence achieved via IO instructions
  C files: `fopen()`, `fseek()`, `fread()`, `fwrite()`
- In object oriented languages; serialization: Converting object into a binary image that can be written on disk or sent over the network.
- This way objects snapshot can be taken, saved, restored and object continue from where it remains.

Memory Management

- Memory manamegement of variables involves architecture, operating system, language runtime and the compiler.
- A typical OS divides memory in sections (segments):
  - Stack section: run time stack
  - Heap section: heap variables
  - Data section: global variables
  - Code section: executable instructions, read only.
- Global variables are fixed at compile time and they are put in data section.
- Heap variables are stored in the dynamic data structures in heap section. Heap section grows and shrinks as new variables are allocated and deallocated.
- Heap section is maintained by language runtime. For C, it is `libc`.

Local Variables

- Local variables can have multiple instances alive in case of recursion.

Local Variables

- Local variables can have multiple instances alive in case of recursion.
- For recursive calls of a function, there should be multiple instances of a variable and compiled code should know where it is depending on the current call state.

Local Variables

- Local variables can have multiple instances alive in case of recursion.
- For recursive calls of a function, there should be multiple instances of a variable and compiled code should know where it is depending on the current call state.
- The solution is to use run-time stack. Each function call will introduce an activation record to store its local context. It is also called stack frame, activation frame.

Local Variables

- Local variables can have multiple instances alive in case of recursion.
- For recursive calls of a function, there should be multiple instances of a variable and compiled code should know where it is depending on the current call state.
- The solution is to use run-time stack. Each function call will introduce an activation record to store its local context. It is also called stack frame, activation frame.
- In a typical architecture, activation record contains:

Local Variables

- Local variables can have multiple instances alive in case of recursion.
- For recursive calls of a function, there should be multiple instances of a variable and compiled code should know where it is depending on the current call state.
- The solution is to use run-time stack. Each function call will introduce an activation record to store its local context. It is also called stack frame, activation frame.
- In a typical architecture, activation record contains:
  - Return address. Address of the next instruction after the caller.

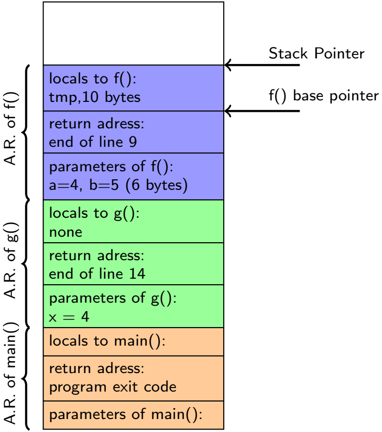
Local Variables

- Local variables can have multiple instances alive in case of recursion.
- For recursive calls of a function, there should be multiple instances of a variable and compiled code should know where it is depending on the current call state.
- The solution is to use run-time stack. Each function call will introduce an activation record to store its local context. It is also called stack frame, activation frame.
- In a typical architecture, activation record contains:
  - Return address. Address of the next instruction after the caller.
  - Parameter values.

Local Variables

- Local variables can have multiple instances alive in case of recursion.
- For recursive calls of a function, there should be multiple instances of a variable and compiled code should know where it is depending on the current call state.
- The solution is to use run-time stack. Each function call will introduce an activation record to store its local context. It is also called stack frame, activation frame.
- In a typical architecture, activation record contains:
  - Return address. Address of the next instruction after the caller.
  - Parameter values.
  - A reserved area for local variables.

```
1   int f(short a, int b) {
2       char tmp[10];
3       ...
4       return a+b;
5   }
6   int g(int x) {
7       int tmp, p;
8       ...
9       tmp = f(x, x+1);
10      ...
11      return tmp+p;
12  }
13  int main() {
14      return g(4);
15  }
```

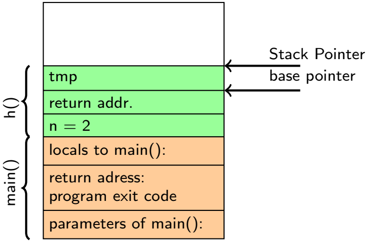


Function Call

- Caller side:
  1. Push parameters
  2. Push return address and jump to function code start (usually a single CPU instruction like `callq`)
- Function entry:
  1. Set base pointer to current stack pointer
  2. Advance stack pointer to size of local variables
- Function body can access all local variables relative to base pointer
- Function return:
  1. Set stack pointer to base pointer
  2. Pop return address and jump to return address (single CPU instruction like `retq`)
- Caller side after return:
  1. Recover stack pointer (remove parameters on stack)
  2. Get and use return value if exists (typically from a register)

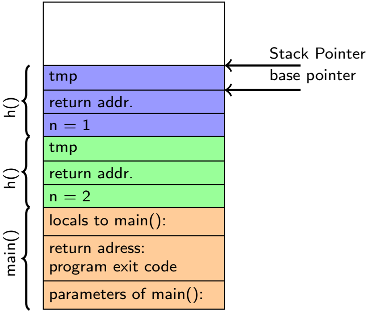
- All locals and parameters have the same offset from base pointer
- Recursive calls execute same instructions

```c
int h(int n) {
    int tmp;
    if (n <= 1) return 0;
    else {
        tmp = h(n-1);
        return n+tmp;
    }
}
int main() {
    printf("%d\n", h(2));
    return 0;
}
```



- All locals and parameters have the same offset from base pointer
- Recursive calls execute same instructions

```c
int h(int n) {
    int tmp;
    if (n <= 1) return 0;
    else {
        tmp = h(n-1);
        return n+tmp;
    }
}
int main() {
    printf("%d\n", h(2));
    return 0;
}
```



- Order of values in the activation record may differ for different languages
- Registers are used for passing primitive value parameters instead of stack
- Garbage collecting languages keep references on stack with actual variables on heap
- Languages returning nested functions as first order values require more complicated mechanisms

```
def multiplier(a):
        def f(x):
                return a*x
        return f

twice = multiplier(2)
# multiplier/a is not alive anymore but twice=f is using it
print(twice(14))
```

Commands

Expression: program segment with a value.
Statement: program segment without a value, but alters the state.
Input, output, variable assignment, iteration...

1. Assignment
2. Procedure call
3. Block commands
4. Conditional commands
5. Iterative commands

Assignment

- C: "`Var = Expr;`", Pascal "`Var := Expr;`".
- Evaluates RHS expression and sets the value of the variable at RHS
- `x = x + 1` . LHS `x` is a variable reference (l-value), RHS is the value
- multiple assignment: `x=y=z=0;`
- parallel assignment: (Perl, PHP) `($a,$b) = ($b, $a);`
  `($name, $surname, $no) =`
  `("Onur","Şehitoğlu",55717);`
  Assignment: "reference aggregate" → "value aggregate"
- assignment with operator: `x += 3; x *= 2;`

Procedure call

- Procedure: user defined commands. Pascal: procedure, C: function returning void
- `void` *functname*(*param1*, *param2*, ..., *paramn*)
- Usage is similar to functions but call is in a statement position (on a separate line of program)

Block commands

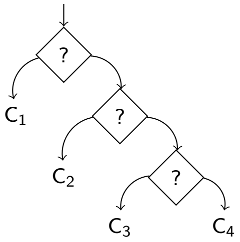
- Composition of a block from multiple statements
- Sequential commands: `{ C1 ; C2; ... ; Cn }`
  A command is executed, after it finishes the next command is executed,...
- Commands enclosed in a block behaves like single command: "if" blocks, loop bodies,...
- Collateral commands: `{ C1, C2, ... , Cn }` (not C ',')!
  Commands can be executed in any order.
- 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.

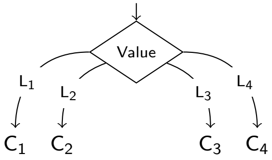
- Concurrent commands: concurrent paradigm languages:
  `{ C_1 | C_2 | ...  | C_n }`
- All commands start concurrently in parallel. Block finishes when the last active command finishes.
- Real parallelism in multi-core/multi-processor machines.
- Transparently handled by only a few languages. Thread libraries required in languages like Java, C, C++.

```
void producer(...) {....}
void collectgarbage(...) {....}
void consumer(...) {....}
int main() {
        ...
        pthread_create(tid1,NULL,producer,NULL);
        pthread_create(tid2,NULL,collectgarbage,NULL);
        pthread_create(tid3,NULL,consumer,NULL);
        ...
}
```

Conditional commands

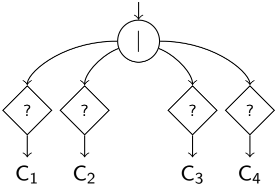
- Commands to choose between alternative commands based on a condition
- in C : `if (`*cond*`)` C_1 `else` C_2 `;`
  `switch (`*value*`) { case` L_1 `:` C_1 `; case` L_2 `:` C_2 `; ...}`
- `if` commands can be nested for multi-conditioned selection.
- `switch` like commands chooses statements based on a value





- non-deterministic conditionals: conditions are evaluated in collaterally and commands are executed if condition holds.
- hyphotetically:
  `if (`$cond_1$`)` C_1 `or if (`$cond_2$`)` C_2 `or if (`$cond_3$`)` C_3 `;`

  `switch (`val`) {`
  `case` L_1`:` C_1 | `case` L_2`:` C_2 | `case` L_3`:` C_3 `}`
- Tests can run concurrently. First test evaluating to True wins. Others discarded.



Iterative statements

- Repeating same command or command block multiple times possibly with different data or state. Loop commands.

Iterative statements

- Repeating same command or command block multiple times possibly with different data or state. Loop commands.
- Loop classification: minimum number of iteration: 0 or 1.

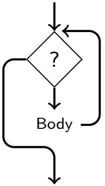
Iterative statements

- Repeating same command or command block multiple times possibly with different data or state. Loop commands.
- Loop classification: minimum number of iteration: 0 or 1.

Iterative statements

- Repeating same command or command block multiple times possibly with different data or state. Loop commands.
- Loop classification: minimum number of iteration: 0 or 1.

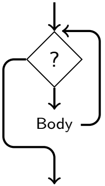
C: `while (...) { ... }`



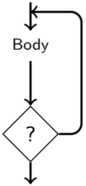
Iterative statements

- Repeating same command or command block multiple times possibly with different data or state. Loop commands.
- Loop classification: minimum number of iteration: 0 or 1.

C: `while (...) { ... }`



C: `do {...} while (...);`



- Definite vs indefinite loops
- 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 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: PHP, Perl, Python, Shell

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

Summary

- Variables with storage
- Variable update
- Lifetime: global, local, heap, persistent
- Commands