CENG 242 - Chapter 3: Variables and Storage

Burak Metehan Tunçel - May 2022

1 Variables and Storage

The variables of imperative programs *do not* behave like mathematical variables. (A mathematical variable stands for a fixed but unknown value; there is no implication of change over time.) The variables of functional and logic programs *do* behave like mathematical variables.

In imperative (and object-oriented and concurrent) programming languages, a ***variable*** is a container for a value, and may be *inspected* and *updated* as often as desired.

To understand how the variables of imperative programs really do behave, we need some notion of ***storage***. We use an abstract model of storage that is simple but adequate:

- A store is a collection of ***storage cells***, each of which has a unique address.
- Each storage cell has a current *status*, which is either *allocated* or *unallocated*. Each allocated storage cell has a current *content*, which is either a *storable value* or *undefined*.

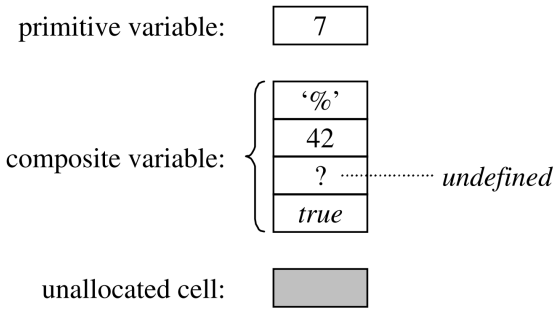


Figure 1: An abstract storage model.

In terms of this storage model, we can view a variable as a container consisting of one or more storage cells. More precisely:

- A ***simple variable*** occupies a single allocated storage cell.
- A ***composite variable*** occupies a group of contiguous allocated storage cells.

A ***storable*** value is one that can be stored in a single storage cell. Each programming language counts certain types of values as storable:

- `C++`'s storable values are primitive values and pointers. (Structures, arrays, unions, and objects are not storable, since none of these can be stored in a single storage cell. Functions also are not storable, since they cannot be stored at all. However, pointers to all of these things are storable.)
- `JAVA`'s storable values are primitive values and pointers to objects. (Objects themselves are not storable, but every object is implicitly accessed through a pointer.)

As a rule of thumb, most programming languages count *primitive values and pointers as storable, but not composite values.*

As an example, in the code 1:

- When function is called, cells are initially unallocated.
- In line 2, cell is allocated/undefined. It is ready to use but value is unknown.
- In line 4, it is storable.
- In line 6, After the including block terminates, the cell is again unallocated.

```
1  void f() {
2    int x;
3    ...
4    x=5;
5    ...
6    return;
7  }
8
9  f();
```

Code 1

2 Simple and Composite Variables

2.1 Simple Variables

A ***simple variable*** is a variable that may contain a storable value. *Each simple variable occupies a single storage cell.*

2.2 Composite Variables

A ***composite variable*** is a variable of a composite type. *Each composite variable occupies a group of contiguous storage cells.*

2.2.1 Total vs Selective Update

A composite variable may be updated either *in a single step* or *in several steps*, one component at a time. ***Total update*** of a composite variable means updating it with a new (composite) value in a single step. ***Selective update*** of a composite variable means updating a single component.

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

Code 2

2.2.2 Static vs Dynamic vs Flexible Arrays

We can view an array variable as a mapping from an index range to a group of component variables. How and when a given array variable's index range is determined can be changed. There are several possibilities:

- the index range might be fixed at compile-time
- the index range might be fixed at run-time when the array variable is created
- the index range might not be fixed at all

A ***static array*** is an array variable whose index range is fixed at compile-time. In other words, the program code determines the index range.

A ***dynamic array*** is an array variable whose index range is fixed at the time when the array variable is created.

A ***flexible array*** is an array variable whose index range is not fixed at all. A flexible array's index range may be changed when a new array value is assigned to it.

3 Copy Semantics vs Reference Semantics

When a program assigns a composite value to a variable of the same type, what happens depends on the language. There are in fact two distinct possibilities:

- ***Copy semantics***. The assignment copies all components of the composite value into the corresponding components of the composite variable.
- ***Reference semantics***. The assignment makes the composite variable contain a pointer (or reference) to the composite value.

Copy semantics is adopted by `C` and `C++`. However, programmers can also achieve the effect of reference semantics by using explicit pointers.

`JAVA` adopts copy semantics for primitive values, and reference semantics for objects. However, programmers can achieve the effect of copy semantics even for objects by using the clone method.

The semantics of the *equality test* operation in any programming language should be consistent with the semantics of assignment. This enables the programmer to assume that, immediately after an assignment of V_1 to V_2, V_1 is equal to V_2, regardless of whether copy or reference semantics is used. It follows that the equality test operation should behave as follows:

- *Copy semantics.* The equality test operation should test whether corresponding components of the two composite values are equal.
- *Reference semantics.* The equality test operation should test whether the pointers to the two composite values are equal (i.e., whether they point to the same variable)

Copy semantics is slower than reference semantics. Reference semantics cause problems from storage sharing (all operations effect both variables), deallocation of one makes the other invalid.

4 Lifetime

Every variable is ***created*** (or *allocated*) at some definite time, and ***destroyed*** (or *deallocated*) at some later time when it is no longer needed. *The interval between creation and destruction* of a variable is called its ***lifetime***.

The concept of lifetime is pragmatically important. A variable needs to occupy storage cells only during its lifetime. We can classify variables according to their lifetimes:

- A ***global variable***'s lifetime is *the program's run-time.*
- A ***local variable***'s lifetime is *an activation of a block.*
- A ***heap variable***'s lifetime is *arbitrary, but is bounded by the program's run-time.*
- A ***persistent variable***'s lifetime is *arbitrary, and may transcend the run-time of any particular program.*

4.1 Global and Local Variables

A ***global variable*** is one that is declared for use throughout the program. A global variable's lifetime is the program's entire run-time: *the variable is created when the program starts, and is destroyed when the program stops.*

A ***local variable*** is one that is declared within a block, for use only within that block. A lifetime of a local variable is an activation of the block containing that variable's declaration: *the variable is created on entry to the block, and is destroyed on exit from the block.*

A ***block*** is a program construct that includes local declarations. In all programming languages, the body of a procedure is a block. An ***activation*** of a block is the time interval during which that block is being executed. In particular, *an activation of a procedure is the time interval between call and return.* During a single run of the program *a block may be activated several times, and so a local variable may have several lifetimes.*

A local variable *cannot retain its content over successive activations of the block* in which it is declared. In some programming languages, a variable may be initialized as part of its declaration. But if a variable is not initialized, its content is *undefined*, *not* the value it might have contained in a previous activation of the block.

Some programming languages (such as `C`) allow a variable to be declared as a ***static variable***, which defines its lifetime to be the program's entire run-time (even if the variable is declared inside a block). Thus static variables have the same lifetime as global variables. Although this feature addresses a genuine need, there are better ways to achieve the same effect, such as class variables in object-oriented languages.

4.2 Heap Variables

A ***heap variable*** is one that can be created, and destroyed, at any time during the program's run-time. A heap variable is created by an expression or command. *It is anonymous, and is accessed through a pointer. (By contrast, a global or local variable is created by a declaration, and has an identifier.)* The lifetimes of heap variables follow no particular pattern.

Pointers are first-class values, and thus may be stored, used as components of composite values, and so on.

An ***allocator*** is an operation that creates a heap variable, yielding a pointer to that heap variable. In `C++` and `JAVA`, an expression of the form "`new` ..." is an allocator. (In `C`, `malloc()`.)

A ***deallocator*** is an operation that destroys a given heap variable. `C++`'s deallocator is a command of the form "`delete` ...". `JAVA` has no deallocator at all. (In `C`, `free()`.) Deallocators are unsafe, since any remaining pointers to a destroyed heap variable become *dangling pointers*.

A heap variable remains ***reachable*** as long as it can be accessed by following pointers from a global or local variable. A heap variable's lifetime extends from its creation until it is destroyed or it becomes unreachable.

Sometimes, operating system tolerates dangling references. However, sometimes, it generates run-time erros like "protection fault" or "segmentation fault" are generated.

4.2.1 Garbage Variables

Garbage variables are 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)

4.2.2 Garbage Collection

A solution to dangling reference and garbage problem is garbage collection. Garbage collection (GC) is a form of automatic memory management. The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced-also called garbage. In other words, PL does management of heap variable deallocation automatically. `Java, Lisp, ML, Haskell` and most functional languages provide garbage collection.

Also, Calls like free() or delete does not exist. 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. Since it may delay execution of tasks, GC is not immediately done. GC usually works in a separate thread, in low priority, works when CPU is idle.

Another but too restrictive solution to garbage is that *reference cannot be assigned to a longer lifetime variable.* Local variable references cannot be assigned to global reference/pointer.

4.3 Persistent Variables

Files may be seen as composite variables. In particular, a ***sequential file*** is a sequence of components, and a direct file is (in effect) an array of components.

Usually files contain large bodies of long-lived data: they are *persistent.* A ***persistent variable*** is one whose lifetime transcends an activation of any particular program. By contrast, a ***transient variable*** is one whose lifetime is bounded by the activation of the program that created it. *Global, local, and heap variables are transient.*

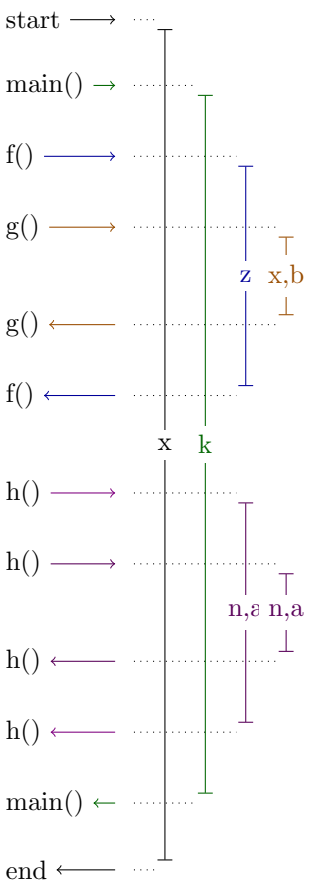
There are certain analogies between persistent variables and transient variables. Persistent variables usually have arbitrary lifetimes, like heap variables; but some systems also allow persistent variables to have nested lifetimes, like local variables. Just as transient variables occupy primary storage, persistent variables occupy secondary storage.

4.3.1 Type Completeness Principle

The ***Type Completeness Principle*** suggests that all the types of the programming language should be available for both transient and persistent variables. A language applying this principle would be simplified by having no special file types, and no special commands or procedures for reading/writing data from/to files. The programmer would be spared the unprofitable effort of converting data from a persistent data type to a transient data type on input, and vice versa on output.

4.4 Examples

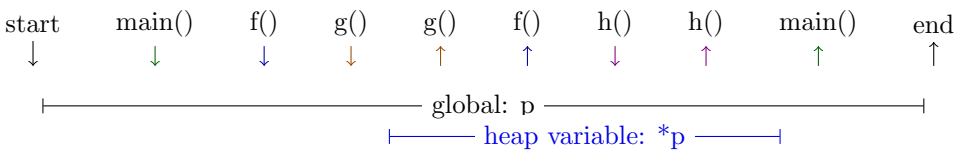
```c
double x;

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;
}
```



Code 3: Example of Global and Local Variables.

```c
double *p;

int h() { ...
}

void g() { ...
  p = malloc(sizeof(double));
}

int f() { ...
  g(); ...
}

int main() { ...
  f();    ...
  h();    ...;
  free(p); ...
}
```

Code 4: Example of Heap Variables.



5 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`.

5.1 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.*

5.2 Function Call

When a function is called:

- **Caller Side**:
  - Push parameters
  - Push return address and jump to function code start (usually a single CPU instruction like callq)
- **Function entry**:
  - Set base pointer to current stack pointer
  - Advance stack pointer to size of local variables
- Function body can access all local variables relative to base pointer
- **Function return**:
  - Set stack pointer to base pointer
  - Pop return address and jump to return address (single CPU instruction like retq)
- **Caller side after return**:
  - Recover stack pointer (remove parameters on stack)
  - 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.

- 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.

6 Pointers

A ***pointer*** is a reference to a particular variable. In fact, pointers are sometimes called *references*. The variable to which a pointer refers is called the pointer's ***referent***.

6.1 Pointers and Recursive Types

Pointers and heap variables can be used to represent recursive values such as lists and trees, but the pointer itself is a low-level concept.

Nearly all imperative languages provide pointers rather than supporting recursive types directly. The reasons for this lie in the semantics and implementation of assignment.

Given the `C++` declarations:

```
IntNode* listA;   IntNode* listB;
```

the assignment "`listA = listB`'' updates `listA` to contain the same pointer value as `listB`. In other words, `listA` now points to the same list as `listB`; *the list is* ***shared*** *by the two pointer variables*. Any selective update of the list pointed to by `listA` also selectively updates the list pointed to by `listB`, and vice versa, because they are one and the same list.

Suppose that `C++` were extended to support list types directly, e.g.:

```
int list listA;   int list listB;
```

Now how should we expect the assignment "`listA = listB`" to be interpreted? There are two possible interpretations:

- *Copy semantics*. Store in `listA` a complete copy of the list contained in `listB`. Any subsequent update of either `listA` or `listB` would have no effect on the other. This would be consistent with assignment of structures in `C++`, and is arguably the most natural interpretation. However, *copying of lists is expensive*.
- *Reference semantics*. Store in `listA` a pointer to the list referred to by `listB`. This interpretation would involve sharing, and would amount to using pointers in disguise. It would be consistent with the assignment of arrays in `C++`. Another advantage of this interpretation would be ease of implementation.

A possible compromise would be to prohibit selective update of lists. Then assignment could be *implemented* by sharing. *In the absence of selective updates, we cannot tell the difference between copy and reference semantics.*

6.2 Dangling Pointers

A ***dangling pointer*** is a pointer to a variable that has been destroyed. Dangling pointers arise from the following situations:

- A pointer to a heap variable still exists after the heap variable is destroyed.
- A pointer to a local variable still exists (e.g., it is stored in a global variable) at exit from the block in which the local variable was declared.

7 Commands

A ***command*** 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. Commands are often called *statements*.

Commands may be formed in various ways. We survey the following fundamental forms of commands. Some commands are primitive:

- *skips*
- *assignments*
- *proper procedure calls*

Others are composed from simpler commands:

- *sequential commands*
- *collateral commands*
- *conditional commands*
- *iterative commands*

There are also *block commands* and *exception-handling commands*.

A well-designed imperative, object-oriented, or concurrent language should provide *all or most of the above forms of command*; it is impoverished if it omits (or arbitrarily restricts) any important forms. Conversely, a language that provides additional forms of command is probably bloated; the additional ones are likely to be unnecessary accretions rather than genuine enhancements to the language's expressive power.

All the above commands exhibit *single-entry single-exit* control flow. This pattern of control flow is adequate for most practical purposes. But sometimes it is too restrictive, so modern imperative and object-oriented languages also provide *sequencers* (such as exits and exceptions) that allow us to program *single-entry multi-exit* control flows.

7.1 Skips

The simplest possible kind of command is the ***skip*** command, which has no effect whatsoever. In `C`, `C++` and `JAVA`, the skip command is written simply ";".

7.2 Assignments

We have already encountered the concept of ***assignment***. The assignment command typically has the form "V `=` E". Here E is an expression which yields a value, and V is a variable access which yields a reference to a variable.

More general kinds of assignment are possible. A ***multiple assignment***, typically written in the form "$V_1 = \ldots = V_n = E$;", causes the same value to be assigned to several variables.

7.3 Proper Procedure Call

A ***proper procedure call*** is a command that achieves its effect by applying a proper procedure (or method) to some arguments. The call typically has the form "$P(E_1, \ldots, E_n)$;", where P determines the procedure to be applied, and the expressions $E_1, \ldots, E_n$ are evaluated to determine the arguments.

7.4 Sequential Commands

Since commands update variables, the order in which commands are executed is important.

A ***sequential command*** specifies that *two (or more) commands are to be executed in sequence.* A sequential command might be written in the form:

$$C_1 \ ; \ C_2$$

meaning that command C_1 is executed before command C_2.

7.5 Collateral Commands

A ***collateral command*** specifies that *two (or more) commands may be executed in any order.* A collateral command might be written in the form:

$$C_1 \ , \ C_2$$

where both C_1 and C_2 are to be executed, but in no particular order.

An unwise collateral command would be:

```
n = 7,  n = n + 1;
```

The net effect of this collateral command depends on the order of execution. Let us suppose that `n` initially contains 0:

- If "`n = 7`" is executed first, `n` will end up containing 8.
- If "`n = 7`" is executed last, `n` will end up containing 7.
- If "`n = 7`" is executed between evaluation of "`n + 1`" and assignment of its value to `n`, `n` will end up containing 1.

Collateral commands are said to be *nondeterministic. A computation is* ***deterministic*** *if the sequence of steps it will perform is entirely predictable; otherwise the computation is* ***nondeterministic***.

If we perform a deterministic computation over and over again, with the same input, it will always produce the same output. But if we perform a nondeterministic computation over and over again, with the same input, it might produce different output every time.

Although the sequence of steps performed by a nondeterministic computation is unpredictable, its output might happen to be predictable. We call such a computation ***effectively deterministic***. A collateral command is effectively deterministic if no subcommand inspects a variable updated by another subcommand.

7.6 Concurrent Commands

- Concurrent paradigm languages:

$$\{ \ C_1 \mid C_2 \mid \ \ldots \ \mid C_n \ \}$$

- 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++`.

7.7 Conditional Commands

A ***conditional command*** has two or more subcommands, of which exactly one is chosen to be executed.

The most elementary form of conditional command is the ***if-command***, in which a choice between two subcommands is based on a boolean value. The if-command is found in every imperative language, and typically looks like this:

if (E) C_1
else C_2

If the boolean expression E yields *true*, C_1 is chosen; if it yields *false*, C_2 is chosen.

The if-command can be generalized to allow choice among several subcommands:

if (E_1) C_1
else if (E_2) C_2
...
else if (E_n) C_n
else C_0

Here the boolean expressions $E_1, E_2, \ldots, E_n$ are evaluated sequentially.

The above conditional commands are *deterministic*: in each case we can predict which subcommand will be chosen. A *nondeterministic* conditional command is also sometimes useful, and might be written in the following notation:

if (E_1) C_1
or if (E_2) C_2
...
or if (E_n) C_n

Here the boolean expressions $E_1, E_2, \ldots, E_n$ would be evaluated collaterally, and any E_i that yields *true* would cause the corresponding subcommand C_i to be chosen. If no E_i yields *true*, the command would fail.

Nondeterministic conditional commands tend to be available only in concurrent languages, where nondeterminism is present anyway, but their advantages are not restricted to such languages.

A more general form of conditional command is the ***case command***, in which a choice between several subcommands is typically based on an integer (or other) value.

The nearest equivalent to a case command in `C`, `C++`, and `JAVA` is the **switch** command.

switch (E) :
  case v_1 : C_1; `break`;
  ...
  case v_n : C_n; `break`;
  case default: C_0; `break`;

7.8 Iterative Commands

An ***iterative command*** (commonly known as a *loop*) has a subcommand that is executed repeatedly. The latter subcommand is called the loop ***body***. Each execution of the loop body is called an ***iteration***.

We can classify iterative commands according to when the number of iterations is fixed:

- ***Indefinite iteration:*** the number of iterations is not fixed in advance.
- ***Definite iteration:*** the number of iterations is fixed in advance.

Indefinite iteration is typically provided by the ***while-command***. The while-command typically looks like this:

```
while (E) C
```

The meaning of the while-command can be defined by the following equivalence:

```
while (E) C  ≡ if (E) {
                  C
                  while (E) C
               }
```

This definition makes clear that the loop condition in a while-command is tested *before* each iteration of the loop body.

Note that this definition of the *while-command is recursive*. In fact, *iteration is just a special form of recursion*. `C`, `C++`, and `JAVA` also have a **do-while-command**, in which the loop condition is tested *after* each iteration.

Definite iteration is characterized by a ***control sequence***, a predetermined sequence of values that are successively assigned (or bound) to a ***control variable*** such as ***for-command*** in `ADA`.

```
for V in T loop
  C
end loop;
```

The control sequence consists of all components of the collection. Iteration over an array or list is deterministic, since the components are visited in order. Iteration over a set is nondeterministic, since the components are visited in no particular order.

Note that the for-command of `C` and `C++` (and the old-style for-command of `JAVA`) is nothing more than syntactic shorthand for a while-command, and thus supports indefinite iteration:

```
for (C1; E1; E2) C2; ≡ C1 while (E1) {
                           C2;
                           E2;
                         }
```

8 Expressions with Side Effects

The primary purpose of evaluating an expression is to yield a value. In some imperative and object-oriented languages, however, it is possible that evaluating an expression has the ***side effect*** of updating variables.