Topics

Variables and Constants

- Variables are stored in memory, whereas constants need not be.
  - Value of variables can change at runtime.
- Variables have a location (l-value) and value (r-value).
- Constants have a value, but no location.
Constants

- Constants may sometimes be stored in memory
- If so, they have r-values but not l-values
- Since values stored in constants cannot be changed, there is no use in accessing l-values
- Thus constants have a “Value semantics”

Values and Constants

- **Values** are quantities manipulated by a program (e.g. integers, strings, data structures, etc.)
- **Constants** have a fixed value for the duration of its existence in a program.
- Constants in a program may be
  - **Literals**: unnamed values specified using a particular representation. e.g.:
    - 42
    - "Markov"
    - 0x2eff
  - **Symbolic**: names associated with fixed values. e.g.
    - `const int n = 100;`
    - `static final int limit = 1024`

Binding Time of Constants

- **Compile-time**
  ```
  const int n = 100;
  ```
  Binding of `n` (to value 100) is known at compile time.
- **Load-time**
  ```
  static final Date d = new Date();
  ```
  Constant `d` is bound to the value of today’s date at load time.
- **Execution-time**
  ```
  int f(int x) { const int y = x+1; ...}
  ```
  Constant `y` is bound to its value at execution time!
- Note that `y` is `local to f` and refers to different entities for each invocation of `f`. The above declaration says that `y` will be constant for any particular invocation.
Variables

- **Variables** are associated with **locations** in **Store** (memory)
- Representation of variables (for explanations only):

![Variable Location Value Diagram]

- The stored values are changed through **assignments**: e.g. \( x = y \);
  - The value stored at the location associated with \( y \) is copied to the location associated with \( x \)

L-value, R-value and Assignment

- In an assignment \( x = y \)
  - we refer to l-value of \( x \) on the lhs ("l" for location or lhs of assignments)
  - r-value of \( y \) on the rhs ("r" for right-hand-side of assignments)
  - **Storage semantics**: update location of \( x \) with the value of \( y \)

![L-value R-value Assignment Diagram]

- Accessing a value stored at a location is called “dereferencing”.
  - C/C++/Java: l-values of variables on rhs are implicitly dereferenced to get their r-values.
  - In ML, dereferencing should be explicit, as in \( x := !y \)

Pointers

- C/C++ “address-of” operation to explicitly turn a reference into a **pointer**.
  - e.g. \&\( x \) evaluates to the location of \( x \).

Example:
```
int x;
// x's location stores int
int *y;
// y's location stores
// pointers to int
x = 20;
y = &x;
```
- The “*” operator is used to dereference a pointer
  - e.g. in the above example, the value stored at \(*y\) is 20
L-value and R-value (Continued)

- **Pointer semantics**
  - x simply “points” to y
  - more efficient if the size of y is large
  - but causes confusion in languages with assignment

- Java uses storage semantics for basic types, and pointer semantics for objects
- C/C++ use value semantics for all types
- In a language free of side-effects (i.e., memory updates), both semantics are equivalent.

### Arrays Vs Pointers in C

- In C, arrays are similar to pointers
  - int a[5];
  - int *b
  - a and b have the same type, but semantically, they differ
  - b = a is allowed, but a = b is not!
    - the l-value of a cannot be changed (it is a const)

### Arrays vs. Pointers in C

- *a=3 and *b=3 have very different effects

  ![Image](image1.png)

  - For this to work correctly, b should have been previously initialized to hold a valid pointer value)
Garbage

- Location that has been allocated, but no longer accessible
  - int *x = new int; *x = 5;
  - int y = 3; x = &y;

Garbage (Continued)

- Accumulation of garbage can lead to programs running out of memory eventually
- But no immediate adverse impact on program
  - correctness of program is unaffected by garbage
- A program that produces garbage is said to have memory leaks

Dangling Pointer

- A pointer that points to memory that has been deallocated
- Consider:
  - int *x, *y, *z;
  - x = new int;
  - *x = 3;
  - y = x
  - delete x;
  - x = NULL;
  - z = new int;
  - *z = 5;
  - *y = 2;
Dangling Pointer (Continued)

- Dangling pointers have an immediate impact on correctness
  - they cause program to fail
- Failure may be immediate
  - access through NULL pointer
- or be delayed
  - corruption of data structures reached through dangling pointers

Dangling Pointer Vs. Garbage

- As compared to garbage, dangling pointers cause much more serious errors
- So, it is safer to never free memory
  - But programs will run out of memory after a period of time
    - Not an issue for programs that run for short times
  - To avoid this, can use garbage collection
    - automatically release unreachable memory
    - used in OCAML, Java
    - garbage collection is much harder for languages with weak type systems (e.g., C and C++)

Aliases

- Alias: Two variables have the same l-value
- C does not support references, but C++ does
  - Use the syntax <typename>&:
    - int& y
  - References have to be initialized with their l-value
    - int x = 1; int& y = x;

![Diagram of alias example]
Aliases

- x and y are aliased
  - they both have same l-value
- when two variables are aliased, assignments to one variable have the side-effect of changing the r-value of the other variable
- side-effects cause confusion
  - They should be used sparingly
  - Aliasing should be used very carefully

Aliases (Continued)

- Aliases may be created using pointer variables as well
- int *x = NULL;

Aliases (Continued)

- x = new int;
- *x = 4;
aliases (continued)

- int *y;
- y = x;

```c
int *y;
y = x;
```