CS571: Programming Languages

Section 5.5
Allocation, Lifetime, and Environment

Environment

- Locations of variables may change during the execution of the program.
- The Environment is responsible for maintaining bindings from names to (memory) locations.
- The environment may be constructed statically (at load time), dynamically (at execution time) or mixture of the two.
- The process of setting up bindings from names to locations is known as storage allocation.
- Fortran: complete static environment – all locations are bound statically.
- LISP: complete dynamic environment.
- C, C++, Ada, Java: mixture.

Static, Stack, Dynamic Allocation

- Static storage allocation: before execution.
  - All variables in original FORTRAN
  - All global variables in C/C++/Java
- Stack storage allocation: needed in any language that supports the notion of local variables for procedures.
  - All local variables in C/C++/Java procedures and blocks.
- Dynamic storage allocation: runtime
  - Functional languages like Scheme and ML
  - In C, objects that are pointed by pointers.

Not All Names are Bound to Locations

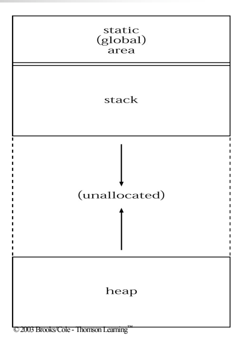
- The C global constant declaration

  const int MAX = 10

  - MAX is never allocated a location -- MAX will be replaced with value 10 by a compiler.

Static, Stack, Dynamic Allocation (Cont.)

- Most languages use a mixture (C, C++, Java, Ada).
- Three components:
  - A fixed area for static allocation
  - A stack area for stack allocation
  - A heap area for dynamic allocation (with or without garbage collection)



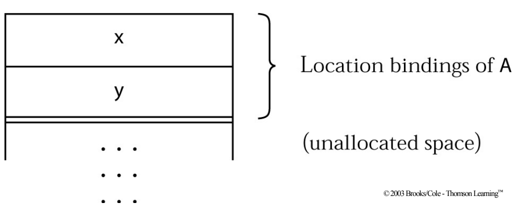
The Runtime Stack

- The environment in a block-structured language also uses the stack to bind the locations to Local variables
  - Local variables are allocated storage when execution enters the block and are automatically deallocated when execution leaves the block.

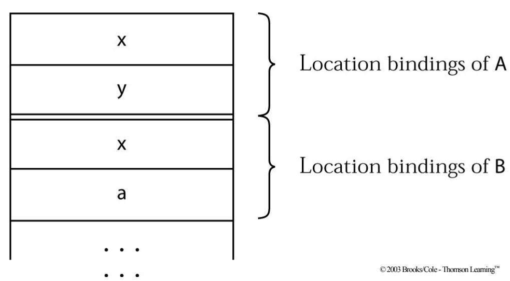
Example: stack Allocation in C within a procedure:

```
(1)   A: {  int x;
(2)         char y;
(3)      B: {  double x;
(4)            int a;
(5)         } /* end B */
(6)      C: {  char y;
(7)            int b;
(8)         D: {  int x;
(9)               double y;
(10)           } /* end D */
(11)        } /* end C */
(12)     } /* end A */
```

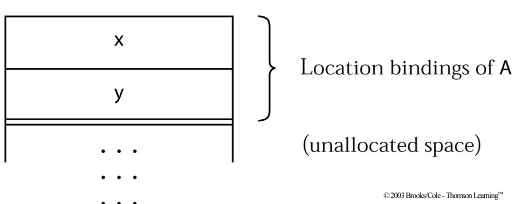

Stack: After the Entry into A



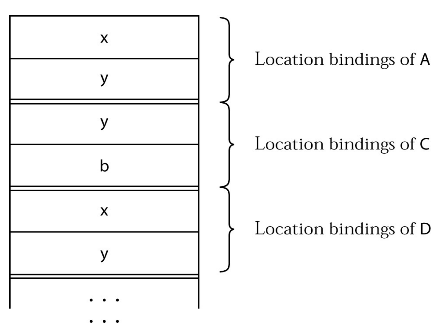
Stack: After the Entry into B



Stack: On Exit from B



Stack: After the Entry to C and D



Heap Allocation

- When pointers are available in the languages, we need to use heap allocation.
- A pointer is an object whose stored value is a reference to another object.

  int* x;

  - Allocation to a pointer variable x, but not the allocation of an object to which x points.

Allocation and Deallocation (C, C++)

- C
  - Allocation:
  - Deallocation:
- C++
  - Allocation:
  - Deallocation:

Allocation and Deallocation (C, C++)

- C
  - Allocation:
    `int* x = (int*)malloc(sizeof(int))`
  - Deallocation:
- C++
  - Allocation:
  - Deallocation:

Allocation and Deallocation (C, C++)

- C
  - Allocation:
    `int* x = (int*)malloc(sizeof(int))`
  - Deallocation:
    `free(x)`
- C++
  - Allocation:
  - Deallocation:

Allocation and Deallocation (C, C++)

- C
  - Allocation:
    `int* x = (int*)malloc(sizeof(int))`
  - Deallocation:
    `free(x)`
- C++
  - Allocation:
    `int* x = new int;`
  - Deallocation:

Allocation and Deallocation (C, C++)

- C
  - Allocation:
    `int* x = (int*)malloc(sizeof(int))`
  - Deallocation:
    `free(x)`
- C++
  - Allocation:
    `int* x = new int;`
  - Deallocation:
    `delete x;`

Allocation and Deallocation (Java)

- Java
  - Allocation:
  - Deallocation:

Allocation and Deallocation (Java)

- Java
  - Allocation:

    Set x = new Set( );
  - Deallocation:

Allocation and Deallocation (Java)

- Java
  - Allocation:

    Set x = new Set( );
  - Deallocation:
    - You cannot do this manually
    - Java takes care of deallocation through garbage collection.

Heap Allocation (Summary)

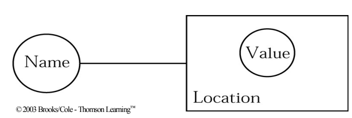
- In C++, Java, heap allocation requires a special operator: **new**.
- In C/C++, deallocation is typically by hand.
- Functional languages (Scheme, ML): heap allocation is performed automatically
  - Everything, including function calls, is allocated on the heap.

Section 5.6

Variables and Constants

Variables and Constants

- A variable is an object whose stored value can change during execution.



- Variables are associated with a location and value.
  - The location is called l-value
  - The value stored in this location is called r-value

l-value and r-value

x = y;

- A name appearing on the left-hand side of an assignment statement (*x*) must have an l-value.
- A name appearing on the right-hand side must have an r-value.
- Some languages make the distinction between l-value and r-value explicitly.
  - ML: x := !x + 1

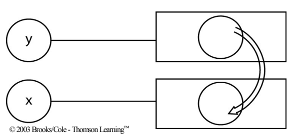
Address of Operator in C

- int x;
- &x is the address of x and can be assigned to a pointer
- For example

```
int x;
x = 10;
int* y = &x;
```


Storage Semantics

x = y;



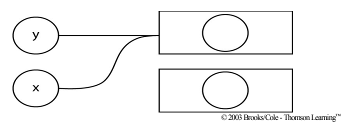
y is evaluated to a value which is then copied into the location of x.

- Most programming languages (e.g. C, C++) use storage semantics, some use pointer semantics.

Pointer Semantics (Assignment by Sharing)

x = y;

- The location of x and y are simply shared.

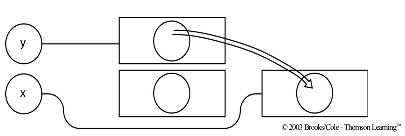


- A future assignment to y may change the value of x.
- Used by Java for object assignment

Pointer Semantics (Assignment by Cloning)

x = y;

- Allocate a new location, copy the value of y, and bind x to the new location



Semantics

- Java supports all kinds of assignment semantics
  - Assignment of simple data

Semantics

- Java supports all kinds of assignment semantics
  - Assignment of simple data:
    - Storage semantics
  - Assignment of object variables:

```
A a1 = new A();
A a2 = new A();
a1 = a2;
```


Semantics

- Java supports all kinds of assignment semantics
  - Assignment of simple data:
    - Storage semantics
  - Assignment of object variables:

```
A a1 = new A();
A a2 = new A();
a1 = a2; //a1 and a2 refer to the same object
         //Assignment by sharing
```

  - Object cloning

```
A a1 = new A();
A a2 = new A();
a1 = a2.clone()
```


Semantics

- Java supports all kinds of assignment semantics
  - Assignment of simple data:
    - Storage semantics
  - Assignment of object variables:

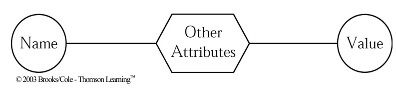
```
A a1 = new A();
A a2 = new A();
a1 = a2; //a1 and a2 refer to the same object
         //Assignment by sharing
```

  - Object cloning

```
A a1 = new A();
A a2 = new A();
a1 = a2.clone()  //create a clone object of type A
                 //with the same content as a2
```


Constants

- A constant is an object whose value does not change throughout its lifetime.



- The semantics of constants: value semantics.
  - Once the value is computed, it cannot change
  - The location of the constant cannot be explicitly referred to by a program

Pointers & Aliases

What makes aliases?

- An alias occurs when the same object is bound to two different names at the same time
- What makes aliases?

What makes aliases?

- An alias occurs when the same object is bound to two different names at the same time
- What makes aliases?
  - Pointer assignment
  - call-by-reference parameters
  - explicit-mechanism for aliasing: EQUIVALENCE in FORTRAN (save memory)
- Why explicit-mechanism for aliasing in Fortran?
  - Save memory - the memory was a valuable resource at that time

Pointers and aliases

(1) int *x, *y;
(2) x = new int;
(3) *x = 1;
(4) y = x;

(5) *y = 2;

(6) printf("%d\n", *x);

Pointers and aliases

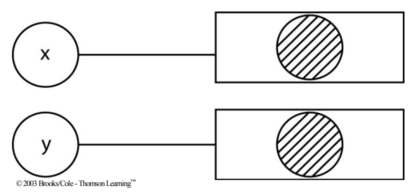
(1) int *x, *y;
(2) x = new int;
(3) *x = 1;

/* *x and *y now aliases*/
(4) y = x;

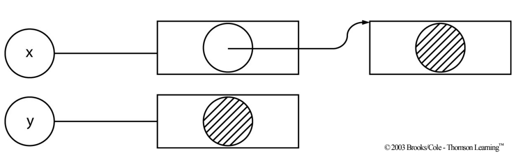
(5) *y = 2;

(6) printf("%d\n", *x);

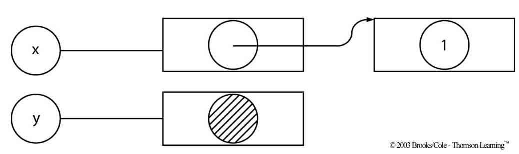
After line 1, both x and y have been allocated, but the value has not been defined



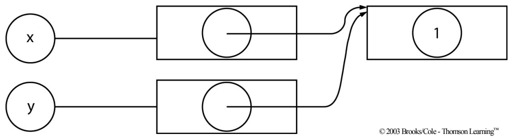
After line 2, *x has been allocated and x has been assigned a value which is equal to the location of *x, but *x is still undefined.



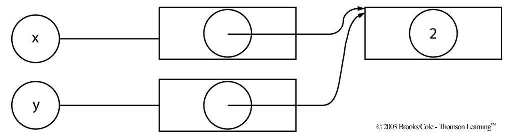
After line 3, the value of *x is 1



Line 4 copies the value of x to y and hence makes *y and *x aliases of each other



After line 5, x also has a value 2.



Example

```
main()
{
    int* x; int** y;
    x = (int*)malloc(sizeof(int));
    y = (int**)malloc(sizeof(int*));
    *y = (int*)malloc(sizeof(int));
    **y = 6;
    x = *y;
}
```


Dangling References

- Locations that have been deallocated, but can still be accessed by a program
- What makes dangling references?

Dangling References

- Locations that have been deallocated, but can still be accessed by a program
- What makes dangling references?

Dangling References

- Locations that have been deallocated, but can still be accessed by a program
- What makes dangling references?
  - Pointer assignment and explicit deallocation
    - e.g. function free in C
  - Pointer assignment and implicit deallocation
    - by block exit
    - by function exit

Dangling References: Example (ex10.c)

```
main(){
  int* x, *y;
  x = (int *) malloc(sizeof(int));
  *x = 2;
  y = x;
  free(x);
  x= 0;
  int* z;
  z = (int *) malloc(sizeof(int));
  *z = 5;
  *y = 4;
  printf("%d\n", *y);
  printf("%d\n", *z);
}
```


Dangling References: Example (ex10.c)

```
main(){
  int* x, *y;
  x = (int *) malloc(sizeof(int));
  *x = 2;
  y = x;    /* *y and *x are now aliases*/
  free(x);  /* *y now a dangling reference*/
  x= 0;
  int* z;
  z = (int *) malloc(sizeof(int));
  *z = 5;
  *y = 4;
  printf("%d\n", *y);
  printf("%d\n", *z);
}
```


Dangling References: Example (ex10.c)

```
main(){
  int* x, *y;
  x = (int *) malloc(sizeof(int));
  *x = 2;
  y = x;    /* *y and *x are now aliases*/
  free(x);  /* *y now a dangling reference*/
  x= 0;
  int* z;
  z = (int *) malloc(sizeof(int));
  *z = 5;
  *y = 4;
  printf("%d\n", *y);
  printf("%d\n", *z);
}
```

Output: 4
4

Sometimes, the space that was previously allocated to *y may be allocated to *z.

Dangling References (Cont.)

- In C, they can occur if a pointer is assigned to a location that has automatic storage management and the lifetime of the pointer is longer than that of the location.

```
{ int *x;
 { int y;
   y = 2;
   x = &y;
 }
}
```


Dangling References (Cont.)

- In C, they can occur if a pointer is assigned to a location that has automatic storage management and the lifetime of the pointer is longer than that of the location.

```
{ int *x;
 { int y;
   y = 2;
   x = &y;
 }
 /* *x is now a dangling reference */
}
```


Garbage

- A location that has been allocated, but is no longer accessible in a program.

```
void p(void)
{   int * x;
    x = (int *) malloc(sizeof(int));
    x = 0;
}
```

Garbage

- A location that has been allocated, but is no longer accessible in a program.

```
void p(void)
{   int * x;
    x = (int *) malloc(sizeof(int));
    x = 0;
}
```

- After x=0, the memory allocated for *x is no longer accessible.

Garbage

- A location that has been allocated, but is no longer accessible in a program.
- Garbage leads to the loss of available memory, but does not affect the correctness of programs.
- Long-running programs eventually run out of memory and crash.
- Not as serious as dangling pointer.

Garbage Collection

- Deallocation is explicit in some languages (e.g. C, C++, Pascal)
- In some languages (e.g. java, SML, C#), it is possible to detect garbage automatically and reclaim it – garbage collection.

Garbage Collection

- Deallocation is explicit in some languages (e.g. C, C++, Pascal)
- In some languages (e.g. java, SML, C#), it is possible to detect garbage automatically and reclaim it – garbage collection.
- Advantages
  - Explicit deallocation: faster
    - The implementation of automatic garbage collection may add significant complexity to the implementation of a language.
  - Garbage collection: manual deallocation errors are among the most common and costly bugs in real-world programs

Pinky Pointer Fun Video

https://www.youtube.com/watch?v=5VnDaHBi8dM

Procedures and Parameter Passing Mechanisms

Procedure

- A procedure is a mechanism in a programming language for abstracting a group of actions or computations.

```
int max (int x, int y)
{
        return x > y? x: y
}
```

body

formal parameters

Procedure Calls

- A procedure is called or activated by stating its name, together with arguments (actual parameters) to the call, which correspond to its parameters.
- Parameter passing is the mechanism of substitution of formal parameters by actual parameters.

```
int max (int x, int y)
{ return x > y? x: y
}

z = max(10, 50).
```


Procedure Calls (Cont.)

```
int max (int x, int y) { return x > y? x: y;}
f( ) { z = max(10, 50);}
```

- A call to procedure transfers control to the beginning of the body of the called procedure (the callee).
- When execution reaches the end of the body, control is returned to the caller.
- In some languages, e.g. FORTRAN, to call a procedure one must also include the keyword CALL, e.g. CALL max(10,50)

Parameter Passing

- By value: Evaluate the actual parameters; assign their values to the corresponding formal parameters.
- By reference: Evaluate the locations of the actual parameters; set the formal parameters to refer to the corresponding locations.
- By name: Evaluate the actual parameters only when the corresponding formal parameters are used.

Call-by-value

- Most commonly used mechanism for parameter passing
- Evaluate the actual parameters, assign them to corresponding formal parameters, execute the body of the procedure

```
int p(int x) {
        x = x + 1;
        return x;
}
```

- An expression y = p(5+3) is executed as follows

Call-by-value

- Most commonly used mechanism for parameter passing
- Evaluate the actual parameters, assign them to corresponding formal parameters, execute the body of the procedure

```
int p(int x) {
        x = x + 1;
        return x;
}
```

- An expression y = p(5+3) is executed as follows
  - Evaluate 5+3=8, call p with 8, assign 8 to x, increase x, return x which is assigned to y.

Call-by-value (Cont.)

- Default parameter passing mechanism in C++ and Pascal, the only parameter passing mechanism in C and Java.
- In C, C++, and Java, parameters are viewed as local variables of the procedure, with initial values given by the values of the arguments in the call

Call-by-value: Pointer (ex1.c)

- If the parameter has a pointer type, then the value is an address and can be used to change memory outside the procedure.

```
void init_p (int* p)
{ *p = 2; }

main()
{   int* q;
    q = (int*) malloc(sizeof(int));
    *q = 1;
    init_p(q);
    printf("%d\n", *q);
}
```


Call-by-value: Pointer (ex1.c)

- If the parameter has a pointer type, then the value is an address and can be used to change memory outside the procedure.

```
void init_p (int* p)
{ *p = 2; }

main()
{   int* q;
    q = (int*) malloc(sizeof(int));
    *q = 1;
    init_p(q);
    printf("%d\n", *q);
}
```

2

Call-by-value: Pointer (ex3.c)

```
void init_p (int* p)
{   p = (int*) malloc(sizeof(int));
    *p = 2;
}

main()
{ int* q;
  q = (int*) malloc(sizeof(int));
  *q = 1;
  init_p(q);
  printf("%d\n", *q);
}
```


Call-by-value: Pointer (ex3.c)

```
void init_p (int* p)
{   p = (int*) malloc(sizeof(int));
    *p = 2;
}

main()
{ int* q;
  q = (int*) malloc(sizeof(int));
  *q = 1;
  init_p(q);
  printf("%d\n", *q);
}
```

- Output: 1
- Directly assigning to p does not change the argument outside the procedure

Call-by-value: Pointer (ex2.c)

```
void init_p (int* p)
{
  p = (int*) malloc(sizeof(int));
  *p = 2;
}

main()
{
  int* q;
  init_p(q);
  printf("%d\n", *q);
}
```

Call-by-value: Pointer (ex2.c)

```
void init_p (int* p)
{
  p = (int*) malloc(sizeof(int));
  *p = 2;
}

main()
{
  int* q;
  init_p(q);
  printf("%d\n", *q);
}
```

- Output: Segmentation fault

Call-by-reference

- Instead of passing the value of the variable, it passes the location of the variable.
  - The parameter becomes an alias for the argument and any changes made to the parameter occurs to the argument as well.
- The only parameter passing mechanism in Fortran.
- In C++ and Pascal, call-by-reference can be specified using extra syntax
  - C++: &
  - Pascal: var

Call-by-reference (Cont.)

- Actual parameters must have l-values. Assign these l-values to l-values of corresponding formal parameters. Execute the body.
- In C++:

```
int p(int& x) {
        x = x + 1;
        return x;
}

int z = 8;
int y = p(z);
```


Call-by-reference (Cont.)

- Actual parameters must have l-values. Assign these l-values to l-values of corresponding formal parameters. Execute the body.
- In C++:

```
int p(int& x) {
        x = x + 1;
        return x;
}

int z = 8;
int y = p(z);
```

  - After the call, both y and z have value 9.

Call-by-reference (Cont.)

ex8.cpp:

```
int p(int& x) {
        x = x + 1;
        return x;
}

int y = p(2);  //??
```


Call-by-reference (Cont.)

ex8.cpp:

```
int p(int& x) {
        x = x + 1;
        return x;
}

int y = p(2);  //??
```

```
bingsun2% g++ ex8.cpp -o ex8
ex8.c: In function `int main()':
ex8.c:10: error: could not convert `2' to `int&'
ex8.c:3: error: in passing argument 1 of `int p(int&)'
```

Call-by-name

- Introduced in Algol60
- On every call to a procedure:
  - Rename all local variables of the procedure to fresh variables: avoid conflict between local variables and variables in the actual parameters.
  - In the procedure body, replace every occurrence of formal parameters by the expressions representing the actual parameters.
  - Evaluate the procedure body.
    - The actual parameters are evaluated only when the corresponding formal parameters are used.

Call-by-name: Example

```
int x;
void p(int i, int j) {
if (i==0)  x = 0;
else  x = j;
}
call p(0, 10/0);
```

Result: ?

Call-by-name: Example

```
int x;
void p(int i, int j) {
if (i==0)  x = 0;
else  x = j;
}
call p(0, 10/0);
```

Result: x=0

```
if (0 == 0) x = 0;
else x = 10/0;
```


Call-by-name: Another Example

```
int i;
int a[10];

void inc(int x)
{ i++;
  x++;
}

main()
{ i = 1;
   a[1] = 1;
   a[2] = 2;
   inc(a[i])
   return 0;
}
```


Call-by-name: Another Example

```
int i;
int a[10];

void inc(int x)
{ i++;
  x++;
}

main()
{ i = 1;
   a[1] = 1;
   a[2] = 2;
   inc(a[i])
   return 0;
}
```

➔

```
int i;
int a[10];

main()
{  i = 1;
   a[1] = 1;
   a[2] = 2;
   i ++;
   a[i] ++;
   return 0;
}
```

Result: i = 2, a[2] = 3, a[1] = 1

Call-by-name: One More Example

```
void intswap(int x, int y)
{ int t = x;
  x = y;
  y = t;
}

main()
{  i = 1;
   a[1] = 2;
   a[2] = 3;
   intswap(i, a[i])
}
```

Call-by-name: One More Example

```
void intswap(int x, int y)
{ int t = x;
  x = y;
  y = t;
}

main()
{  i = 1;
   a[1] = 2;
   a[2] = 3;
   intswap(i, a[i])
}
```

➔

```
main()
{  i = 1;
   a[1] = 2;
   a[2] = 3;
   int t = i;
   i = a[i];
   a[i] = t;
   return 0;
}
```

Result: i = 2, a[1] = 2, a[2] = 1

Parameter Passing: Summary

- By-Value, By-Reference
  - Strict Evaluation: Actual parameters are evaluated whether or not they are needed in the procedure.
- By-Name
  - Lazy Evaluation: Actual parameters are evaluated at most once, and only when they are needed in the procedure.