Names, Bindings and Scopes

The two main components of von Neumann architecture was :

1. Memory
   ↳ stores data and instructions
2. Processor
   ↳ provides operations for modifying the content of the memory

Generally, imperative programming languages are abstractions of von Neumann architecture.

↳ The abstractions for memory cells in all these programming languages are Variables

Variables : characterized by a number of properties or attributes. For instance,

Type of variable, scope of variable
Lifetime of variables,

In functional programming language, concept of variables is absent, but they allow expressions to be named.

↳ but it is different than the variable concept of imperative language, as they cannot be changed.

↳ similar to named constant as in imperative language

Attributes of Variables

Name & Forms

Design issues: Are names case sensitive ?
↳ Relates with readablity

↓ Length: if too short, the names may not be connotative

FORTRAN I : 6
COBOL : 30
~~C~~ Java, Ada : no limit
C++ : no limit, but to keep symbol table small, users impose constraints

↳ Fortran 95: has limitation of 31 character length.

C#, Java, Ada: they don't have any limit

Special characters: In PHP, variables name start with a dollar sign.

In Perl, three types of variables
- ✓ Scalar variable
- ✓ Array variable
- ✓ Hashes

Regardless of the variable types, all variables in Perl start with special characters

$	for Scalar :	number, string
@	Array :	Ordered lists of scalars
%	Hashes :	Key/value pairs

In C language, "camel notation" is used.
↳ all the words of a multiple word name except the first are capitalized.
"myStack"

Case Sensitivity:

Case sensitive languages suffer from readability problem

- names that look alike are different.
- Names in C-based languages are case sensitive

Special words:

- Make programs more readable
  - by naming actions to be performed
- Keyword: It is special only in ~~set~~ certain context. For instance, in Fortran

  Real VarName
  - Data type followed by a variable name
    - so, Real is keyword

  Real = 3.4
  - Here Real is variable

- Reserved word: Reserved words that can not be used as names.
  - for, while, if etc.

Too many reserved ~~language~~ words create problem. For instance, COBOL has about 300 reserved words.

Variables

Move from machine language was mainly because of replacing numeric memory address for data with names,

→ Variables ← Abstraction

So, a variable is an abstraction of a memory cell.

Variables can be characterized as a six-tuple attributes:

1. Name
2. Address
3. Value
4. Type
5. Lifetime
6. Scope

Address:

- Variables addresses may be different at different times during execution
- A variable may have different addresses at different places in a program.
- Two variable names can be used to access the same memory location

→ Known as aliases

→ created via pointers, refference variables, unions as in C and C++.

aliases

→ Harmful to readability

→ Programmers must remember all of those instances

Pointers

Before the concept of Pointers is explained, it is important that the concept of variable is revisited.

Variable:

- has a name
- value of which can vary
- Compiler and linkers assign a specific block of memory within the computer.
  ↳ to store the value of the variable
- Size of that ~~variable depends~~ block depends on the range over which the variable is allowed to vary.

Data Type	Memory	Format specifier
int	4 bytes	%d
float	4 bytes	%f
long int	8 bytes	%ld
double	8 bytes	%lf

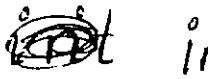
One can use sizeof() to know the variable.

Declaration of Variable:

When we declare a variable, we inform compiler two things —

1. Name of the variable
2. Type of the variable

For instance,



int x

Type of variable ← int

x → Name of variable

Once we define int x, compiler reserves 2 bytes to store the value of the integer.

```
int x;
x = 2;
```

When executed, value 2 is placed in the two-bytes memory ~~stored~~ reserved to store the value of x

⊞ Observation from the variable definition:

- two values associated with the object K
  a. Integer value stored there
  b. Other value is the memory location → address of x

These two values are also known as —

a. rvalue (right value)
b. lvalue (left value)

Sometimes

rvalue refers to the value that is on the right side of the assignment operator → "="

lvalue is the value permitted on the left side.

Types: Determines the range of values of variables and the set of operations that are defined for values of that type.

Floating point precision is also determined by type.

Value: Content in the memory location with which the variable is associated with.

Abstract memory cell:

Physical cell or collection of cells associated with a variable.

CONCEPT OF BINDING

→ An association between an entity and a symbol.

→ For instance, between an operation and symbol between a variable and its type

Binding time: is the time at which a binding takes place.

Possible Binding Times:

Language design time : bind operator symbols to operator

Language implementation time : bind floating point type to a representation

Compile time : bind a variable to a type in C or Java

Load time : bind a C or C++ static variable to a memory cell.

Runtime : bind a nonstatic local variable to a memory cell

Static and Dynamic Binding

Static : A binding is static if it first occurs before run time and remains unchanged throughout program execution

Dynamic : A binding is dynamic if it first occurs during execution or can change during execution of the program.

Language feature	Binding time
syntax. if (x>0) as in C y=x;	Language design
keywords, reserved words meaning of operators etc. +, -, * are defined	Language design
Primitive types such as float, struct in C	Language design
Internal representations of 3·414, 2·5 etc. and "test string"	Language implementation
specification of type of variables	Compile time
storage allocation method for a variable	Language design, language implementation and/or compile time
loading executable in memory and absolute address adjustment	Load time
Non-static allocation of space for variable	Run time

Binding and Binding times are prominent concepts of semantics of programming language.

↳ meaning of programming language

For instance,

"*" asterisk symbol is bound to multiplication operation, which is done at the language design time

A data type, such as, int in C-programming is bound to a range of possible values.

↳ Done during language implementation time

Consider the assignment as below: C++ assignment

```
count = count + 5
```

- Type of count is bound at compile time
- Set of possible values of count is bound at compiler design time [in C, implementation time]
- Meaning of operator symbol "+" is bound at a compile time
- Internal representation of the literal 5 is bound at compiler design time.
- Value of count is bound at execution time with this statement

To summarize, Understanding the binding times for the attributes of program entities is a prerequisite for understanding the semantics of a programming language.

Type Bindings

The two important aspects of variable binding to a data type are:

1. How is a type specified?
2. When does the binding take place?

If static,

Types can be specified by either —

a) an explicit
b) an implicit declaration.

Explicit declaration:

- done through a statement in a program that lists variable names and their types.

  int x

  Here, type is explicitly defined/declared

- Most widely used programming language use static type binding explicitly

Implicit ~~def~~ declaration:

- Associating ~~of~~ ~~var~~ variables with types through default conventions.

  No declaration statement

- In this case, First appearance of a variable name constitutes it implicit declarations.

Implicit declarations ... continues

- Implicit declaration example —

```
void main ( ) {
    int a;
    float b;
    char c;
    b = c;   /* execution of b = c */
}
```

↳ At the compile time, the compiler does the conversion of c to ensure that float type value is assigned to b

↳ known as implicit declaration

- Generally, language processor such as compiler or interpreter does the implicit type binding

Fortran, Perl, Ruby, JavaScript etc. provide implicit declarations.

Advantage: Improves writability, but it is a minor convenience.

Disadvantage: Very detrimental to the reliability

↳ Prevent the compilation process from detecting the typographical and program errors.

⊞ However, the disadvantages of implicit declarations can be avoided as well.

For instance,

Perl requires names of specific types to begin with particular special characters.

Name begin with $: Scalar (String / Numeric)
Name begin with @ : array
Name begin with % : hash structure

So, @apple , %apple are unrelated

Dynamic binding:

- Type of variable is not specified by a declaration statement
- Type cannot be determined from the spelling of its name.
  Instead
- Variable is bound to a type when it is assigned a value in an assignment statement.

JavaScript

previous Line: list = [10.2, 3.5]; single-dimension array of length 2

Next line: list = 47; list becomes a name of a scalar variable

Python, Ruby, JavaScript and PHP are examples

Advatages of Dynamic Binding

- Biggest ~~advat~~ advantage is the flexibility
- For instance, numeric data of any type can be processed using any variable.

↓ allows to write

Generic Program

↳ meaning that, it can deal with any types of data.

Disadvantages:

1. It makes program less reliable.

↳ Because, error detection capability of the compiler is diminished

↓

Incorrect types of right sides of assignments are not detected as errors.

↓ rather,

Left-side type is simply changed.

```
i = x;
```

i and x are scalar numeric variable

For keying error, if written as

```
i = y;
```

where y is array

The type of @i is changed to array

2. Cost is the another big disadvantage
   ↳ in terms of high execution time

Caused by the dynamic nature of attribute binding for each variable

Type-checking is done at run time.

requires run-time descriptor to keep track of current type.

Finally, languages using dynamic type binding for variables are usually implemented using pure interpreters

↳ About 10 times slower than executing machine code created by the compiler.

Because, computers do not have instructions whose operand types are not known at compile time.

Memory management : For a program

A written program is stored at the main memory, before the program is run by operating system.

memory, for what ?

Memory is allocated for the program instructions, and for the data the program uses.

How is it allocated ?

Apparently, the old way is to use the concept of segmented memory.

means

As the os loads a program, it allocates a contigious block or segment of memory to a program,

The program can not use more than the amount allocated.

Because of the limited memory,

It is important to divide it into specific regions that perform specific function within the program

obsolete

However, this process is now obsolete, and the memory management/organization is still performed, but in a different way.

Storage bindings and Life-time

One of the most important issues in imperative programming language design is to decide the storage bindings for its variables

- Storage binding defines fundamental character of an imperative programming language

Allocation: Memory cell that is bound to a variable is ~~gen~~ taken from the pool ~~fo~~ of available memory cell.

The process is known as Allocation

Deallocation: It is the process of placing a memory cell that is unbound from a variable back to the available pool of memory cell.

Life-time: Life-time of a variable is the total time during which it is bound to a specific memory cell.

So, the life-time of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell.

Based on the ~~cate~~ life-time, storage-bindings of variable is subdivided into four categories

- Static
- stack-dynamic
- explicit heap-dynamic
- implicit heap-dynamic

Stack and Heap Memory

Any program that runs an a computer takes up memory

Whenever a new variable is created, memory cells are allocated accordingly.

Generally,

Every running program has its different layout separated from other programs. The common segments are —

stack: stores local variables, created by functions

heap: dynamic memory; stores global variables
↳ for programmer to allocate

data: stores global variables
↳ separated into initialized and uninitialized

Text: Stores code being executed.
in program's memory

Each memory location is pin pointed using an address

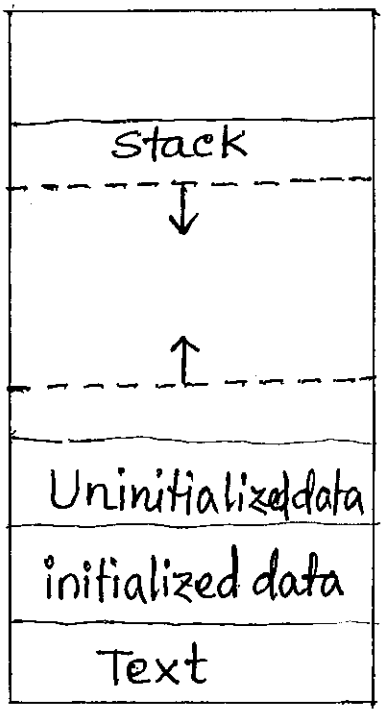
↓

expressed in base 16 numbers

smallest address possible is 0X00000000

largest address possible is 0XFFFFFFFF

stack has the higher address



High Address

Low Address

Consider a sample C-program, and when it executes the stack looks as below

```
7. int stack_ex(){
8.     int a=50;
9.     return a;
   }
1. int main(){
2.     int a;
3.     int b=5;
4.     int c=1234;
5.     int *p=&b;
6.     int d=stack_ex();
10.    return 0;
   }
```

(2) main: int a ??

memory for "a" is allocated

(3) main: int a ??, int b 5

memory for "b" is allocated and stored 5

(4) main: int a ??, int b 5, int c 1234

memory for c is allocated and stored 1234

(5) main: int a ??, int b 5, int c 1234, int *p

memory for p is allocated to store the address of b

(7) (8) (6) main: int a ??, int b 5, int c 1234, int* p, stack_ex: int a 50

memory for "a" of stack_ex is allocated to store 50

(9) main: int a ??, int b 5, int c 1234, int *p

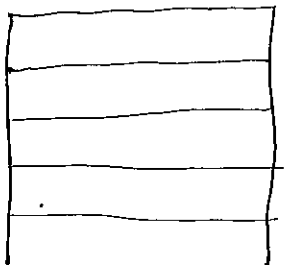
memory for stack_ex variable a is deallocated.

value 50 is returned to main().

6 main: int a ??, int b 5, int c 1234, int *p, int d 50

d is allocated to store 50

(10) Finally,



* stack memory of main is deallocated and return 0

Stack:

- Memory segment with high address
- Machine allocate some stack memory every time a fnc is called.
- When new local variables are defined additional stack memory is allocated
  - ↳ So, stack memory expands and grow downwards
    - ↳ fnc returns value
      - ↳ When the fnc returns value, the allocated memory is deallocated.
        - ↳ Means that, local variables will be invalid.
- Allocations and deallocations are automatically done.
  - ↳ done through compiler instructions.

Static variables

- They bound to memory cells before program execution begins.
- They remain bound to same memory cells until program execution is finished and later, are terminated.
- Globally accessible variables are necessary to remain bound to same memory address, as these variables are often accessed during execution.

→ static variables
- not deallocated until the program ends.

Advantage:
- Addressing is direct, so, comparatively faster than other variables that have indirect addressing.
- No-run-time overhead for memory allocation and de-allocation.

Dis-advantages:
- A language that has only static-variables cannot support recursive subprogram
- storage can not be shared among variables
- If two subprograms are never active at the same time and their arrays are static they cannot use the same storage.

Examples:
- Static variables as defined in C/C++ functions.

~~Static~~ Stack-dynamic variable

- Storage bindings are created for variables when their declarations are elaborated.
  - ↓ means that executable code
  - when the associated ↑ is executed.

Example: Variable declarations in C subprogram and Java methods

Here, types are statically bound

variables are de-allocated when the execution of the methods are done.

- As the name says, stack-variables are allocated from the run-time stack.

Advantages:
- allows recursion of subprograms
  - ↳ Provides dynamic local storage to each subprogram
- conserve storage
  - ↳ All the subprogram share the same memory space for their locals.

Disadvantages:
- Overhead of allocation and deallocation
- slower access because of indirect addressing.

Problem with function returning Pointer

As known, stack memory of a function is deallocated when the function returns value.

so, there is no gurantee that the value remains stored in the memory location

or, the value in the memory location may be changed.

Example: A mistake is to return a pointer to a stack variable in a helper function

Because

After the value is returned, that is, the caller gets the pointer, the stack memory may be overwritten.

Heap

Dynamic memory allocation generally occurs in the heap memory storage cells.

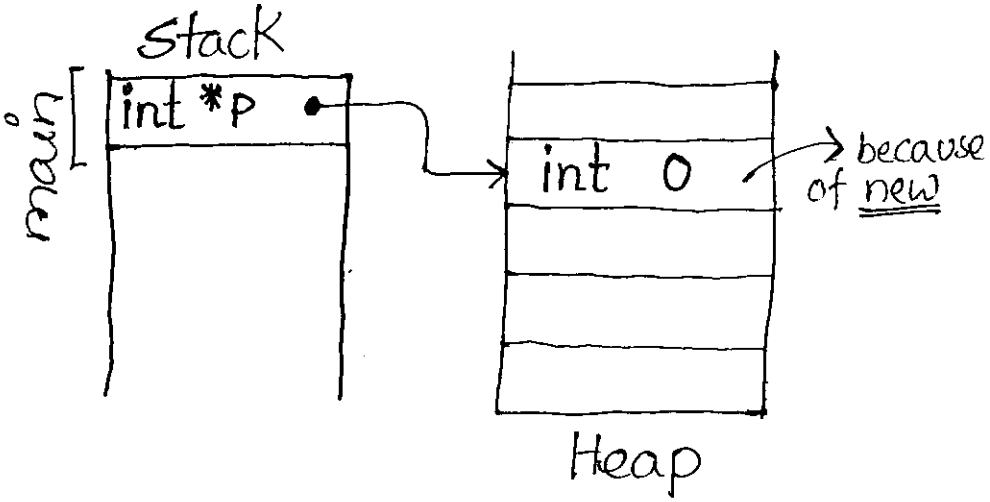
- Unlike the stack memory, heap memory is explicitly allocated by the programmer
- The heap memory is deallocated only if it is explicitly freed by the programmer.
- Some of the commands use for heap allocation

malloc realloc free	new in C++ int *intnode intnode = new int; : delete intnode;

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, *ptr, total=0;
    printf("Enter number of elements:");
    scanf("%d", &n);
    ptr = (int*) malloc(n * sizeof(int));
    :
    free(ptr)
    :
```

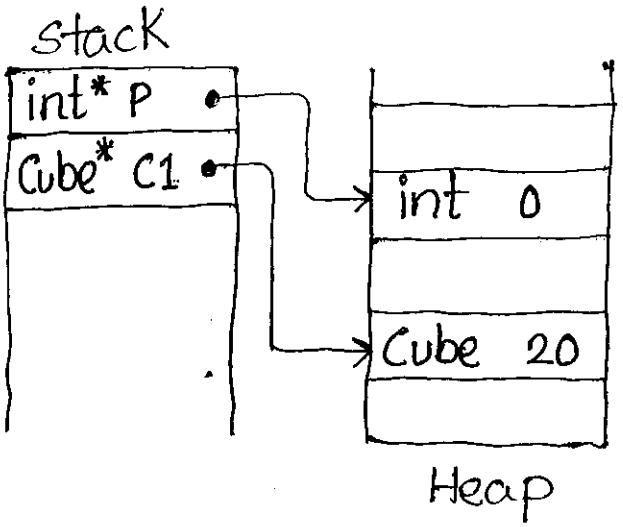
if n = 10 entered, then the command assigns 10 x 4 bytes = 40 bytes




```
1. int main()
2. { int *p = new int;
3.   Cube *c1 = new Cube()
4.   Cube *c2= c1;
5.   c2 -> setLength(20);
6.   return 0;
   }
```

1,2 We allocate an integer with default value "0".

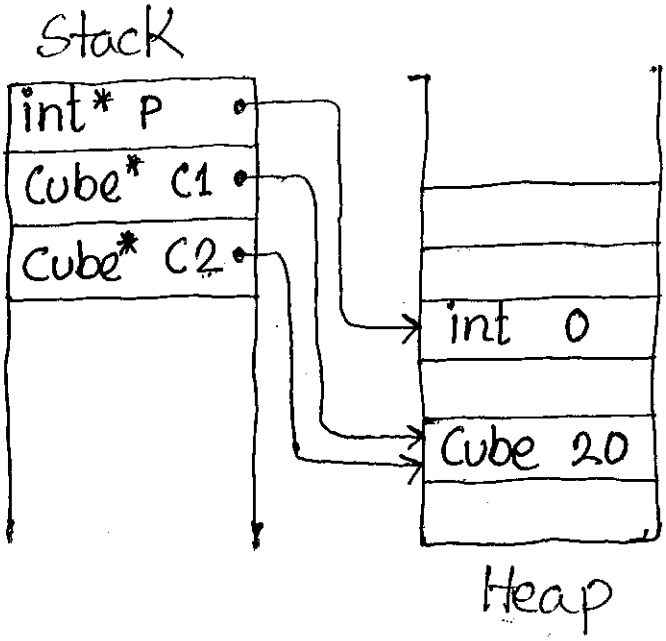
The address of "0" is placed stored in p, which allocated on main's stack

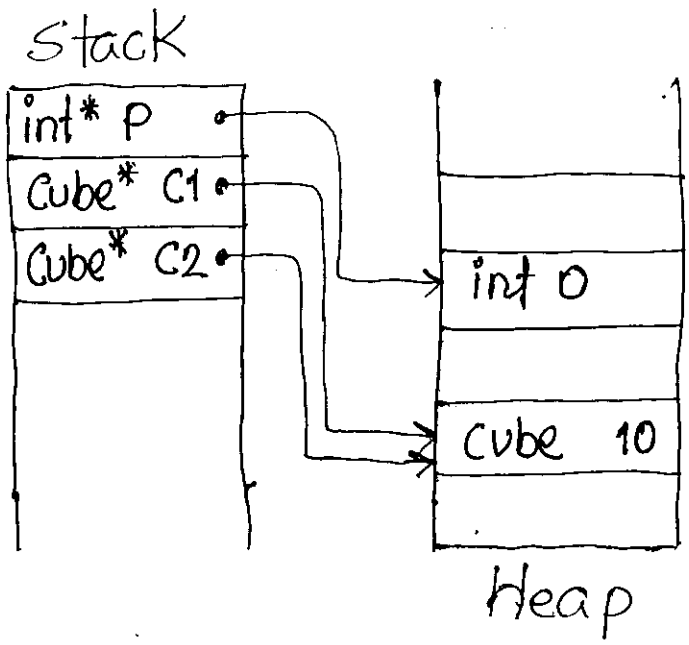


3 It allocates a Cube with certain width, say 20, on the heap memory.

Allocate c1 on the main stack to store the address of the Cube

4 Allocate c2 on main's stack and store a copy of c1



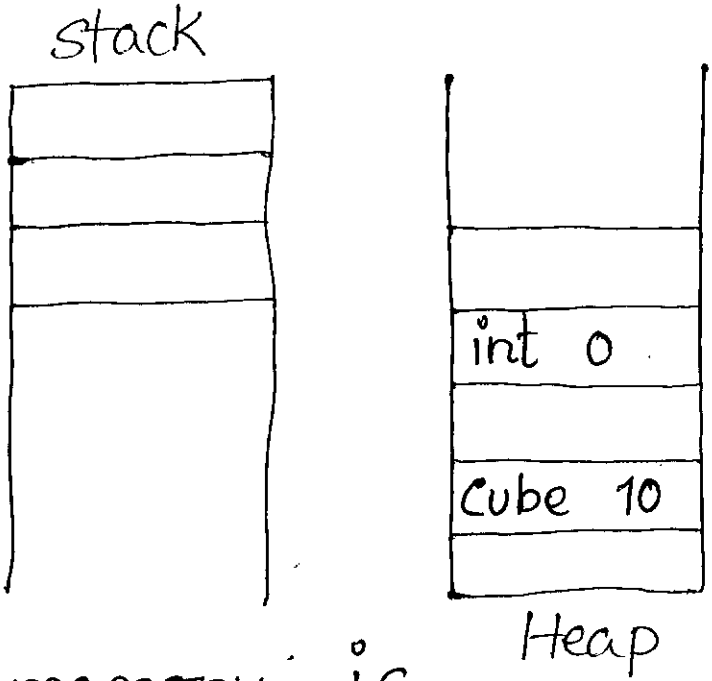


5

Method setLength was called on C2. This call changes the width of the Cube ϕ pointed by C1 and C2

6

Stack memory is deallocated, and return 0



As we see, stack memory is erased / freed. But, the heap memory is not freed.

Memory is freed only if it is done by the programmer.

known as memory leak

Important: As functions can not return pointers of stack variables, the values can be put at HEAP [more | permanent] than stack memory.

Explicit heap-dynamic

- Allocated and de-allocated by explicit directives as specified by the programmers
- It takes effect during execution
- The variables that are allocated and deallocated from heap, can only be referenced through pointer or reference value.
- Example: For instance in C++, the operator "new" uses a type as its operand and when executed,

An explicit heap-dynamic variable is created of the operand type and its address is returned

These variables are bound to type at compilation time

Binding is static