Content of programming languages
Names, bindings and scopes

Part of 3.1-3.3

Names

- Can we have a program without names?
  - In fact, in our daily life, how many sentences we use that do not contain names?
- Names are basic units in programming
- What are names?
  - identifiers are names in most programming languages.
  - Symbols like “+” are names

Bindings

- In programming languages, a non-trivial question is what objects do names refer to. E.g., what are the outputs of the following C++ program

```
 int i = 10;
   { int i = 5;
     cout << "first cout" << i << endl;
   }
cout << "second cout" << i;
```

- Different occurrences of the same *name* may mean (or associate with) different objects!
- Why should we allow this?
  - For a program written by many people, it is hard to require people to use all different names. It helps to allow people use names freely (i.e., the same name used by different people may mean different objects)
  - In fact, for a large program, even written by one person, it is hard to use all different names!
  - So, allowing this will facilitate the *ease of programming.*

- For example, when we write a square function

```
int square (int x) {
  int y;
  y = x * x;
  return y;
}
```

clearly we don’t want to forbid the user of this function to use `y` in other parts of his program.

- A *binding* is an association of a name to an object. It defines what object a name means.
- For your understanding purpose, it may help in the rest of the notes if you can replace name by “variable name.”

Scope

- Since the binding of a name could change we introduce the concept of *scope of bindings.*
- The *scope* of a binding is the maximal textual part of the program in which the binding holds.
- The *scope* of a variable is the maximal textual part of the program in which binding of this variable holds.

Binding

- *Binding time* can be, among many others (see the book)
  - *compile time*: when the program is compiled into an executable program
  - *run time*: when the program runs on a computer
- *Static binding* informally refers to the binding before run time; while *dynamic binding* the binding during run time.
- In general, early binding times implies greater efficiency while later binding times implies greater flexibility

Scope rules

- Since the binding of a name may change in different parts of a program, we need *scope rules* to define precisely how the binding changes
- Since binding associates names to *objects*, let’s talk more about objects

Lifetime of objects and bindings

- Creation of objects, destruction of objects
- creation / destruction of bindings
- reactivation / (temporary) deactivation of bindings
- The period of time from creation to destruction of a binding (or object) is called the *lifetime* of the binding (or object)
  - If an object outlives binding, it's *garbage*
  - If a binding outlives the associated object, the name becomes *dangling*.

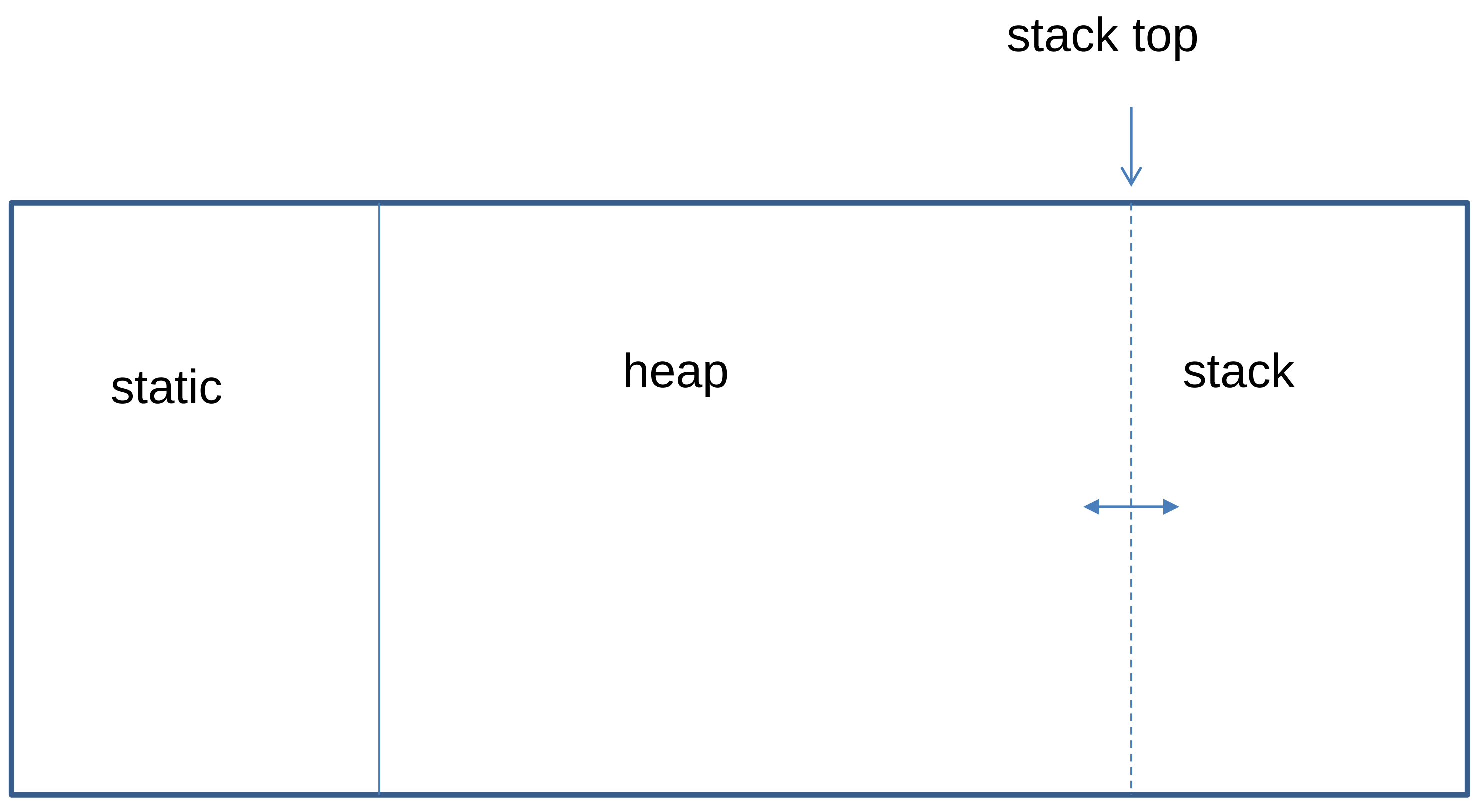
Objects and Memory Model

- To create an object is to reserve a block in the memory
- Question: how to organize the memory to facilitate the creation/destruction of objects and creation, destruction, deactivation, and reactivation of bindings?

Memory Model

- The memory organized into three parts
  - static part: an object can be created anywhere here, and once created, it will never be destructed.
  - stack based part: this part is a stack. To create an object here, one has to use the memory block at the top of the stack. To destruct an object, one can only just pop out the memory block at the top.
  - heap: an object can be created anywhere here, and can be destructed by releasing the occupied space.

Memory model



Scope rules and implementation

- Examples of scope rules and the memory model
  - Global variables:
    - Scope rule: there is a permanent binding which may be deactivated by local variables
    - Implementation of this rule: for each such variable, create (by the compiler) a block (i.e., an object) in the static part and a binding between this variable and this block.

- Local variables in a block
    - Scope rule: the lifetime of the binding of these variables is from the declaration of this variable to end of the block (a different rule: the same as lifetime of the block.)
    - Implementation. For each such variable
        - At the beginning of the subroutine, create an object in the *stack* part and a binding between it and the variable
        - Deactivate the binding of the global variable with the same name as the local variable
        - At the end of the subroutine, destruct the object and then the binding for this variable

Static scope rule

- With *static (lexical) scope rule,* a scope is defined in terms of the textual structure of the program
- In a block structured language like C
  - Static scope rule: for any name, use the closest (in the textual program) binding of this variable.
  - Implementation: using the stack part too …

Dynamic scope rule

- With *dynamic scope rule,* bindings depend on the current state of program execution
- As an example of dynamic scope rule, for any variable, always use the *most recent* (in time) binding.
- As an example of static scope rule, for any variable, always use the *closest* (in space of the textual program) binding.

Static vs dynamic rule

-
  ```
  program scopes (input, output );
  var a : integer;
  procedure first;
     begin a := 1; end;
  procedure second;
     var a : integer;
     begin first; end;
  begin
     a := 2; second; write(a);
  end.
  ```

More about names

- Names in a program can be more than just variable names: you have names for constants, operations, types etc.
- For example, consider the name + in the following program (it means different functions -- overloading)
  - `int x, y;`
  - `float z, w;`
  - `x = y+3; // +: on integer`
  - `Z = W+Z; // +: on float numbers`

Summary

- Name, objects, binding, scope