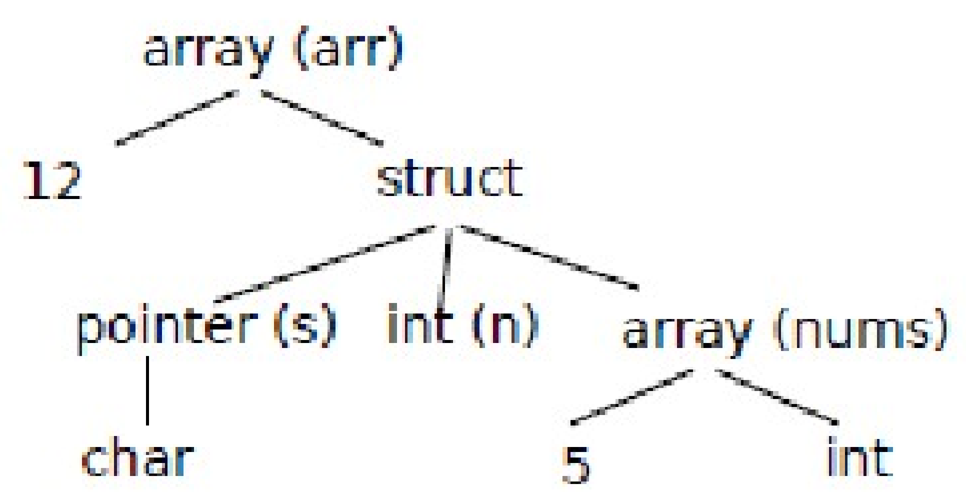
# *Lecture #24*

## *Semantic Analysis & Run-time System*

# *Equivalence of Compound Types*

- The equivalence of base types is usually very easy to establish:
  - an **int is equivalent** only to another **int**
- **Compound Types**

```
struct {
    char *s;
    int n;
    int nums[5];
} arr[12];
```



# *Equivalence of Compound Types*

- Rules for building type trees
  - Arrays: two subtrees, one for number of elements and one for the base type
  - Structs: one subtree for each field
  - Pointers: one subtree that is the type being referenced by the pointer

# *Equivalence of Compound Types*

- A recursive test for structural equivalence

```
bool AreEquivalent(struct typenode *tree1, struct typenode *tree2) {
   if (tree1 == tree2)  // if same type pointer, must be equiv!
      return true;
   if (tree1->type != tree2->type)      // check types first
      return false;
   switch (tree1->type) {
      case T_INT: case T_DOUBLE: ...
         return true;      // same base type
      case T_PTR:
         return (AreEquivalent(tree1->child[0], tree2->child[0]);
      CASE T_ARRAY:
         return (AreEquivalent(tree1->child[0], tree2->child[0]) &&
                (AreEquivalent(tree1->child[1], tree2->child[1]);
      ...
```

# *Run-time Systems*

- We have completed the entire front-end of a compiler:
  - Scanning
  - Parsing
  - Semantic analysis

- These stages depend only on the properties of the source language.

- They are completely independent of :
  - The target (machine or assembly) language
  - The properties of the target machine
  - Operating system

# *Run-time Systems*

- The front-end:
  - Enforces the language definition
  - Builds data structures that are needed to do code generation

- If the front-end has not generated any error:
  - We have a valid program in the source language that we are compiling
  - We are ready to produce code which is a valid translation of the source code and that can executed on a given target architecture

# *Run-time Systems*

- The Back-end:
  - Optimization
  - Code generation

- Runtime Systems:
  - What the target program looks like and how is it organized. Why?
    - We have to know what we need to generate before knowing how we generate it and how such a generation strategy makes sense.

-

# *Run-time Support*

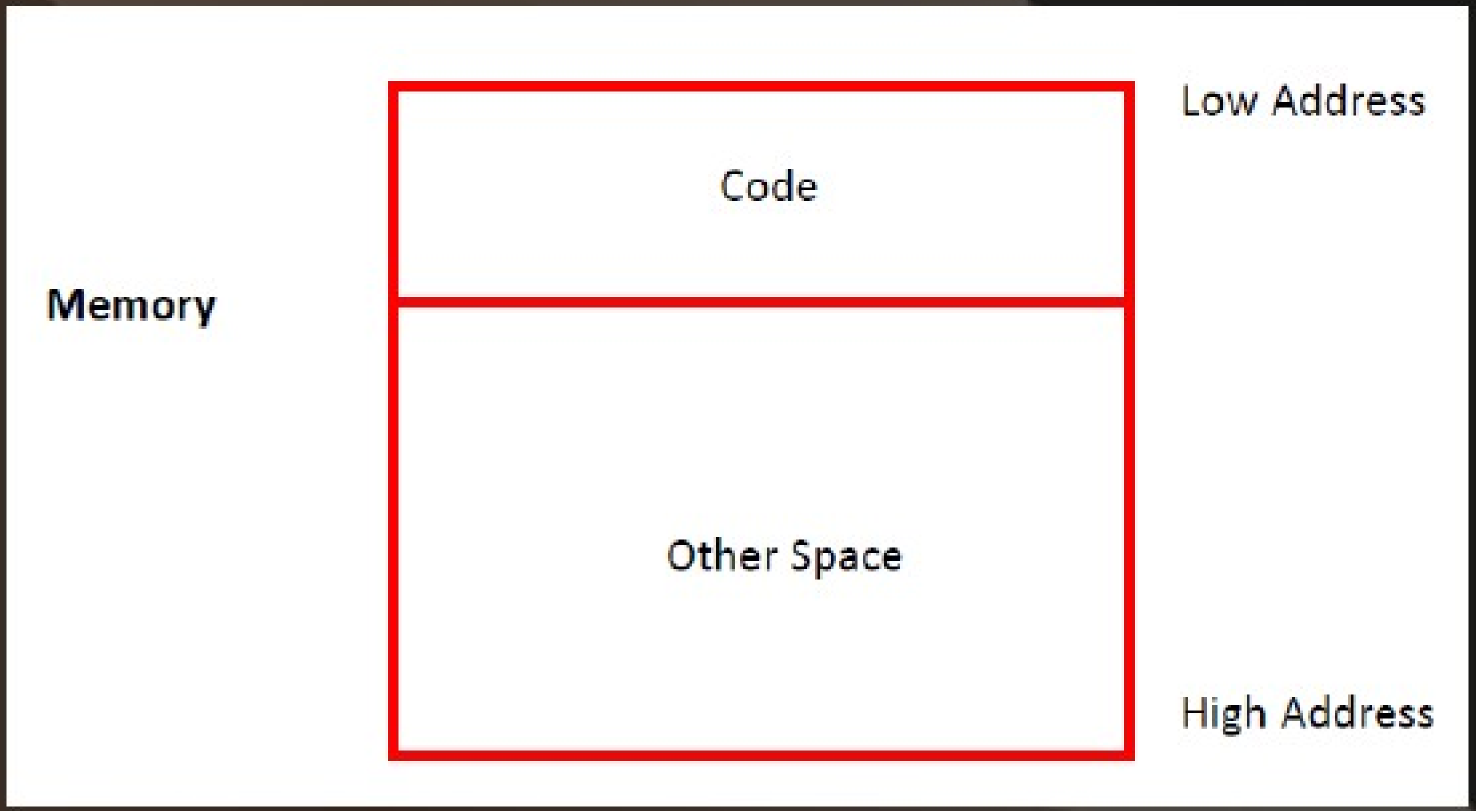
- The target program interacts with system resources.
- There is a need to manage memory when a program is running
  - This memory management must connect to the data objects of programs
  - Programs request for memory blocks and release memory blocks
  - Passing parameters to functions needs attention
- Other resources such as printers, file systems, etc., also need to be accessed

# *Runtime Support*

- Execution of a program is initially under the control of the operating system
- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., "main")

# *Management of Run-time Resources*

- The compiler is not only responsible for generating code but also handling the associated data
- Compiler needs to decide what the layout of data is going to be and then generate code that correctly manipulates the data
  - References data from with in code
  - Code and layout of data needs to be designed together
- Storage Organization



# *Procedure Activations*

- Two goals in code generation:
  - Correctness
  - Speed
- Fast as well as correct – Difficult

- Two assumptions of Activation:
  - Execution is sequential; control moves from one point in a program to another in a well-defined order
  - When a procedure is called, control always returns to the point immediately after the call

# *Procedure Activations*

- An invocation of procedure **P** is an *activation of P*

- The *lifetime of an activation of P is*
  - All the steps to execute **P**
  - Including all the steps in procedures **P** calls

- The *lifetime of a variable x is the portion of execution in which x is defined*

- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept

# *Procedure Activations*

- Observation
  - When **P** calls **Q**, then **Q** returns before **P** returns

- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree

```
class Main {
    int g() { return 1; };
    int f()  { return g(); };
    int main() {{ g(); f(); }};
}
```