SOM2IF15 – Compilation

Concepts of Programming Languages

Frédéric Loulergue

2022

Chapters

Reference

- Maurizio Gabbrielli, Simone Martini, *Programming Languages: Principles and Paradigms*, Springer, 2010
- Thereafter mentioned as the “textbook”

Outline

Reference

- Chapter 4 of the textbook

Outline

Name

Name

A name is a sequence of characters used to **represent**, or **denote**, another object ("object" is intended in a wide sense, not in the technical sense of object-oriented languages)

Abstraction

- `int x`: data abstraction
  - `x` is a symbolic identifier for a memory location
  - abstracting from the low-level details of memory addresses
- `void incr(int * x){ *x = *x + 1; }`: control abstraction
  - name associated with a set of commands
  - visibility: name and parameters form an interface

Denotable Objects

Denotable Object

- A programming language element that can be given a name
- Many differences between programming languages

Examples

- User defined objects: variables, formal parameters, procedures (in the broad sense: procedures, functions, methods, subprograms, ...), user defined types, labels, modules, constants, exceptions, ...
- Programming language defined objects: primitive types, primitive operations, predefined constants, ...

Binding

Definition

A binding is an association between a name and an object it denotes

Name and Object are Different

"*The variable* `x` *has type* `int`"
is an abbreviation for
"*The value referenced with name* `x` *has type with name* `int`"

Names and Objects

- An object can have different names:

```
List<Integer> l1 = new ArrayList();
List<Integer> l2 = l1;
```

- A name can be bound to different objects during execution

```
(define name 42)
(set! name "Forty Two")
```

Binding Time

- **Design of language**: bindings between primitive constants, types and operations of the language are defined

  For example, + indicates addition, and int denotes the type of integers, ...
- **Program writing** Given that the programmer chooses names when they write a program, we can consider this phase as one with the partial definition of some bindings, *later to be completed*

  For example, the binding of an identifier to a variable is defined in the program but is effectively created only when the space for the variable is allocated in memory

Binding Time

- **Compile time** The compiler, translating the constructs of the high-level language into machine code, allocates memory space for some of the data structures that can be *statically processed*.

  For example, the global variables of a program
- **Runtime** This term denotes the entire period of time between starting and termination of a program. All the associations that have not previously been created must be formed at runtime.

  For example, for bindings of variable identifiers to memory locations for the local variables in a recursive procedure, or for pointer variables whose memory is allocated dynamically

Linking and Loading

- What is exactly done in these phases depends on the programming language
- The goal is to obtain an executable program from different modules
- One task related to names is to *resolve references* to externally defined objects: add information in the caller code about where to find the object externally defined

Binding Time related to Linking and Loading

Depending on the language and operating system, or even configurable:

- Static: prior to execution
- Dynamic: during execution

Outline

Environment

Definition (Environment)

The set of associations between names and denotable objects which exist at runtime at a specific point in the program and at a specific time during execution, is called the **(referencing) environment**

Definition (Declaration)

A declaration is a construct that allows the introduction of an association in the environment.

Example

```
int x;
int f (){ return 0; }
type T = int;
```

Definition (Alias)

An *alias* is a name for a denotable object already named

Example

```
int  *x,  *y                          //  x, y:  pointers  to  integers
x   = (int *)  malloc(sizeof(int));   //  allocate  heap memory
*x  = 5;                              //  *  dereference
y   = x;                              //  y  points  to  the  same object as x
*y  = 10;
printf ("x=%d\n", *x);
```

Blocks: Definitions

Definition (Block)

A block is a textual region of the program, identified by a start sign and an end sign, which can contain declarations local to that region, that is, which appear within the region

Definition (Procedure block)

A block associated with a procedure is a block associated with declarations local to a procedure. It corresponds textually to the body of the procedure itself, extended with the declarations of formal parameters.

Definition (In-line block)

An In-line block is a block which does not correspond to a declaration of procedure and which can appear (in general) in any position where a command can appear

Block: Nesting

Nested Block

- In most language, blocks can be nested
- Opening/closing of block should always be well parenthesised

Counter Example (not allowed)

```
open block A;
    open block B;
close block A;
    close block B;
```

Visibility

Definition (Visibility)

A declaration local to a block is visible in that block and in all blocks listed within it, unless there is a new declaration of the same name in that same block. In this case, in the block which contains the redefinition the new declaration hides the previous one.

Blocks: Examples

Java

```
class Block
{
  static public void main(String [] a)
  {
    int x = 10;
    {
      double y = Math.PI;
      System.out.println("y = "+y);
    }
    System.out.println("x = "+x);
  }
}
```

C

```
#include "stdlib.h"
#include "stdio.h"
int main(void)
{
  int x = 10;
  {
    double x = 42.25;
    printf ("x = %f\n",x);
  }
  printf ("x = %d\n",x);
  return EXIT_SUCCESS;
}
```

Types of Environments

Definition (Type of Environment)

The environment associated with a block is formed of the following components:

- **local environment**: composed of the set of associations for names declared locally to the block (for procedure blocks include the formal parameters)
- **Non-local environment:** the environment formed from the associations for names which are visible from inside a block but which have not been declared locally
- **Global environment:** formed from associations created when the program's execution began. It contains the associations for names which can be used in all blocks forming the program

Types of Environment

```
A: {  //  We assume A is the global block
   int  a = 1;
   B: {
      int  b = 2;
      int  c = 2;
      C: {
         int  c = 3;
         int  d;
         d = a+b+c;
         printf ("d = %d\n", d);
      }
      D:{
         int  e;
         e = a+b+c;
         printf ("e = %d\n", e);
      }
   }
}
```

Operations on Environments

- **Creation of associations between name and denoted object (naming):** elaboration of a declaration or connection of a formal to an actual parameter when a new block containing the declaration is entered
- **Reference to a denoted object via its name** This is the use of the name in an expression, in a command, or in any other context. The name is used to access the denoted object.
- **Deactivation of association between name and denoted object**: when entering a block in which a new association for that name is created locally. The old association is not destroyed but remains inactive. It will be usable again when the block containing the new association is left.
- **Reactivation of an association between name and denoted object:** When leaving block in which a new association for that name is created locally, reactivation occurs. The previous association can now be used.
- **Destruction of an association between name and denoted object (unnaming):** on local associations when the block in which these associations were created is exited. The association is removed from environment and can no longer be used.

Operations on Denotable Objects

- **Creation of a denotable object** This operation is performed while allocating the storage necessary to contain the object. Sometimes, creation includes also the initialisation of the object
- **Access to a denotable object** Using the name, and hence the environment, we can access the denotable object and thus access its value. At a given point in the program and during a given execution, there is a one-to-one correspondence.
- **Modification of a denotable object:** for languages that allow mutability it is possible to access the denotable object via a name and then modify its value
- **Destruction of a denotable object** An object can be destroyed by reallocating the memory reserved for it (explicitly or automatically)

Outline

Scoping Rules: An Example

What will be printed?

```
A: {
  int x = 0;
  void fie(){
    x = 1;
  }
  B: {
    int x;
    fie();
  }
  print(x);
}
```

Static Scoping

Rule 1

The declarations local to a block define the local environment of that block.

The local declarations of a block include only those present in the block (usually at the start of the block itself) and not those possibly present in blocks nested inside the block in question

Static Scoping

Rule 2

- If a name is used inside a block, the valid association for this name is the one present in the environment local to the block, if it exists.
- If no association for the name exists in the environment local to the block, the associations existing in the environment local to the block immediately containing the starting block are considered. If the association is found in this block, it is the valid one, otherwise the search continues with the blocks containing the one with which we started, from the nearest to the furthest.
- If, during this search, the outermost block is reached and it contains no association for the name, then this association must be looked up in the language's predefined environment.
- If no association exists here, there is an error.

Static Scoping

Rule 3

A block can be assigned a name, in which case the name is part of the local environment of the block which immediately includes the block to which the name has been assigned. This is the case also for blocks associated with procedures.

Pros and Cons

+ Environment present in a program by reading the text
+ Static verifications at compile time
+ Efficient compilation
- More complex to implement

Languages: most of them

Static Scoping

Example

```
{
  int x = 0;
  void fie (int n){
    x = n+1;
  }
  fie (3);
  write(x);
  {
    int x = 0;
    fie (3);
    write(x);
  }
  write(x);
}
```

Dynamic Scoping

Definition (Dynamic Scope)

According to the rule of dynamic scope, the valid association for a name X, at any point P of a program, is the most recent (in the temporal sense) association created for X which is still active when control flow arrives at P.

Pros and Cons

+ Simplifies runtime environment management
+ Useful for some very specific applications
- No static information for verification or optimization

Languages: APL, Lisp (some versions), Perl, ...

Dynamic Scoping

Typo in the Textbook (Fig 4.5, page 81)

```
{
  const x = 0;
  void fie (){
    write(x);
  }
  void foo(){
    const x = 1;
    {
      const x = 2;
    }
    fie();
  }
  foo();
}
```

- Prints 1

Outline

Summary

- **Denotable objects** are the objects to which names can be given. Denotable objects vary according to the language under consideration.
- **Environment**: set of associations existing at runtime between names and denotable objects
- **Blocks** In-line or associated with procedures, these are the fundamental construct for structuring the environment and for the definition of visibility rules
- **Environment Types**: local environment, global environment and non-local environment.
- **Operations on Environments**: Associations present in the environment in addition to being created and destroyed, can also be deactivated, and re-activated, and can be used.
- **Scope Rules** are rules which, in every language, determine the visibility of names.
- **Static Scope** is typically used by the most important programming languages.
- **Dynamic Scope**: easiest to implement. Used today in few languages.

Outline

Reference

- Textbook Chapter 6

Outline

Expressions and Commands: Textbook Definitions

Definition (Expressions)

An expressions is a syntactic entity whose evaluation either produces a value or fails to terminate, in which case the expression is undefined

Definition (Commands)

A command is a syntactic entity whose evaluation does not necessarily return a value but can have a side effect

Expressions and Commands: Formal Semantics View

Semantics of Expressions (W^0, W^1, W)

- $\mathcal{A} : (aexpr \times State) \to \mathbb{Z}$
- General form: $\mathcal{A}[\![e]\!]\sigma = n$

Semantics of Commands (W^0, W^1, W)

- Relation on $(command \times State) \times (command \times State)$
- General form: $\langle com,\ \sigma \rangle \to \langle com',\ \sigma' \rangle$

Expressions and Commands

What does this program print?

```
#include "stdio.h"
int x = 0;
int f(int y){ x = y + 1; return x; }
void main(void){
   printf ("expr = %d\nx = %d\n", f(1)+f(2), x);
   printf ("x = %d\n", x);
}
```

Expressions and Commands: Experiments

What does this program print?

- Compiler: gcc 5.5.0 version 9.1.0 (on MacOS)

```
expr = 5
x = 0
x = 3
```

- Compiler: clang version (on MacOS)

```
expr = 5
x = 3
x = 3
```

6.5 Expressions

- An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof.
- Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.
- The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call.

6.5 Expressions

- An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof.
- Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.
- The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call.

The Previous Program

- is undefined (why?)

Expressions

Order of Evaluation Matters Because

- of side-effects
- of finite arithmetic: $a + b - c$ with $a = \texttt{MAX\_INT}$, $b < c$

Order of Evaluation of Expressions

- in most languages and for most operations is undefined (to allow for compiler optimization)
- is fixed for conditional expressions and at least some boolean operations (in C: `&&` and `||` for e.g.)

Advices for Writing Correct Programs

- know the semantics of your language!
- being as explicit as possible using parenthesis, ...
- avoid side effects in expressions

Outline

Sequence Control Commands

Explicit Sequence Control

- Sequential Command: `;`
- Composite Command: block of lists of commands
  - C-like: `{ ... }`
  - Pascal-like: `begin ... end`
- `goto`
- Other: `break`, `continue`, `return`

Conditional and Iterative Commands

- Conditional Commands: Textbook section 6.3.2
- Iterative Commands: Textbook section 6.3.3

Structured Programming

Debate in the 70s

- Related to the rejection of the `goto` statement:
  Edsger W. Dijkstra. 1968. Letters to the editor: go to statement considered harmful. Commun. ACM 11, 3 (March 1968), 147-148.
- Prescribes programming language features and a programming methodology

Structured Programming

- Top-down or hierarchical design of programs
- Code modularisation:
  - procedures and functions
  - modules
- Meaningful names and comments
- Use of structured data types (e.g. records)
- Use of structured control constructs:
  - a single entry point
  - a single exit point

Outline

Recursion

Definition (Informal)

- A recursive procedure is a procedure whose body contains a call to itself
- Recursion can be indirect: two (or more) procedures can be mutually recursive

Mathematical Definitions

No Total Function $f : \mathbb{N} \to \mathbb{N}$ Defined by

$$\begin{cases} f(0) & = & 0 \\ f(n) & = & f(n) + 1 \text{ for } n > 0 \end{cases}$$

No Unique Function Defined by

$$f(1) = f(1)$$

Valid Recursive Procedure Definitions

Non Terminating

```
int f(int n){ return (n == 0)? 1 : (f(n)+1); }
```

Non Terminating

```
int f(int n) {
  if (n==1) return f(1);
}
```

Execution

factorial

```
int fact(int n){
  if (n≤1)
    return 1;
  else
    return n*fact(n−1);
}
```

factorial version 2

```
int f(int n, int res){
  if (n≤1)
    return res;
  else
    return f(n−1, n*res);
}
int fact(int n){
  return f(n, 1);
}
```

Tail Recursion

Definition (Tail Recursion)

- Let f be a function which, in its body, contains a call to a function g (different from f or equal to f).
- The call of g is said to be a tail call if the function f returns the value returned by g without having to perform any other computation.
- We say that the function f is tail recursive if all the recursive calls present in f are tail calls.

Iteration and Recursion

Tail Recursion

- Memory occupation can be optimized
- But is not always optimized
  To check: a non terminating tail recursive function

Iteration or Recursion

- Iteration more natural for arrays, matrices, tables
- Recursion more natural for symbolic structures (lists, trees)
- Tail recursive functions can be as efficient as loops

Outline

Summary

- Distinction expression / command (or instruction)
- Expressions: problems related to evaluation order
- Commands/Instructions:
  - overview of existing commands
  - structured programming
  - recursion

Outline

Reference

- Textbook Chapter 7

Outline

Control Abstractions

Subprograms

- Provide control abstraction: functions and procedures
- Higher-Order Functions:
  - Functions as parameter
  - Functions as result
  - Textbook section 7.2

Exceptions

- Textbook section 7.3

Outline

Procedures and Functions

Vocabulary

```
int r;
int fact(int n){
  return (n≤0)?1:(n*fact(n-1));
}
void main(void){
  r = fact(6);
  printf("fact(6) = \%d", r);
}
```

- Header
- Formal parameters
- Actual parameters
- Return value
- Non local environment

Functional Abstraction

Software component:

- provides services to its environment
- clients are not interested in how they are implemented
- clients are interested in how to use them

Subprograms as components:

- clients not interested by the body
- clients interested by the header
- real functional abstraction:
  possible to substitute a function by another one with the same header and semantics
- procedures/functions in PL only provide partial support for functional abstraction

Outline

```c
void write_a(const int t[], int size){
  assert(size >=0);
  printf("[ ");
  for(int i=0; i<size; i++)
    printf("%d ", t[i]);
  printf("]\n");
}
void to_zero(int t[], int size){
  assert(size >=0);
  for(int i=0; i<size; i++)
    t[i] = 0;
}
void swap(int x, int y)
{ int tmp = x; x=y; y=tmp; }
int main(char ** argv, int argc)
{
  int t[] = { 1, 2, 3, 4, 5 };
  swap(t[0], t[1]); write_a(t, 5);
  to_zero(t, 5); write_a(t, 5);
}
```

```
program parameters;
const size = 5;
type arr5 = array [1..size] of integer;
procedure write_a (const t: arr5);
var i : 1..size;
begin
  write('[ ');
  for i:=1 to size do
    write(t[i],' ');
  writeln(']')
end;
procedure to_zero(t : arr5);
var i : 1..size;
begin
  for i := 1 to size do
    t[i] := 0;
end;
procedure swap(var x : integer;
               var y : integer);
var tmp : integer;
begin tmp:=x; x:=y; y:=tmp end;
var t: arr5 = (1, 2, 3, 4, 5);
begin
  swap(t[1],t[2]); write_a(t);
  to_zero(t); write_a(t);
end.
```

```
def to_zero(a):
    for i in range(0,len(a)):
        a[i] = 0;

def swap(x, y):
    tmp=x;
    x=y;
    y=tmp;

a = [ 1, 2, 3, 4, 5];
swap(a[0], a[1]); print(a);
to_zero(a); print(a);
```

The examples

- C: `[1,2,3,4,5]` and `[0,0,0,0,0]`
  Pass-by-Value but the value of an array is a reference to its first cell
- Pascal : `[2,1,3,4,5]` and `[2,1,3,4,5]`
  Pass-by-Value by default, Pass-by-Reference with `var`, and the value of an array is the sequence of the values of its cells
- Python: `[1,2,3,4,5]` and `[0,0,0,0,0]`
  Pass-by-Value but the value of an array is a reference to its first cell

```
#include <iostream>
using namespace std;
void swap(int* x, int* y)
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main()
{
  int a = 0, b = 42;
  cout << "a = " << a
       << " b = " << b << "\n";
  swap(&a, &b);
  cout << "a = " << a
       << " b = " << b << "\n";
}
```

```
#include <iostream>
using namespace std;
void swap(int & x, int & y)
{
  int tmp = x;
  x = y;
  y = tmp;
}

int main()
{
  int a = 0, b = 42;
  cout << "a = " << a
       << " b = " << b << "\n";
  swap(a, b);
  cout << "a = " << a
       << " b = " << b << "\n";
}
```

In both examples, the values are swapped: left, pass-by-value with the value is an address, right, pass-by-reference

Parameter Passing Modes

Pass by Value

- Actual parameters are evaluated
- The values are used to initialize the formal parameters
- Formal parameters are local variables
- Advantages: fast for scalar, protection
- Disadvantages: copy, additional memory

Parameter Passing Modes

Pass by Reference

- A formal parameter is an alias for its corresponding actual parameter
- Advantage: no copy, no additional memory
- Disadvantage: indirection, no protection

Parameter Passing Modes

Pass by Value and Pointers/References

In C, Java, C++: it is the value of the pointer/reference (i.e. an abstraction of a memory address) that is copied, not the value pointed to.

Parameter Passing Modes

Pass by Constant

- Like call by value
- The PL implementation checks that no assignment is made
- It no assignment made, passing the reference is safe

Parameter Passing Modes

Pass by Result

- Only for output parameters
- The parameter should be a l-value (something that can be assigned)
- The body of the subprograms computes the result in a local variable that is then copied back to the actual parameter

Parameter Passing Modes

Pass by Value-Result

- Pass by Value for the entry
- Pass be Results for the exit

Pass by Value-Result and Pass by Reference

- Are they equivalent?

Parameter Passing Modes

Pass by Name

- In functional programming: Haskell
- In imperative languages too: Algol
- No current imperative PL supports CbN

Outline

Summary

- the concept of procedure
- parameter passing methods:
  - by value
  - by reference
  - by constant
  - by result
  - by value/result
  - by name

Outline

Reference

- Textbook Chapter 8

Outline

Types

Definition (Data Type)

A data type is a homogeneous collection of values, effectively presented, equipped with a set of operations which manipulate these values

Types

Data types are used

1. At the design level, as support for the conceptual organisation
2. At the program level, as support for correctness
3. At the translation level, as support for the implementation.

Type Systems

Definition (Type System)

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

B. Pierce, *Types and Programming Languages*. MIT Press, 2002

Type Systems

Definition (Type System)

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

B. Pierce, *Types and Programming Languages*. MIT Press, 2002

A type system consists of

- Predefined types of the language
- Mechanisms to define new types
- Mechanisms to control the use of types:
  - Equivalence rules: when are two types equal?
  - Compatibility
  - (Type inference)
- How are types checked: statically or dynamically?

Type Systems

Type Safety

A type system (and its associated language) is **type safe** (or strongly typed) when no program during its execution can generate a non signaled error caused by a type violation

Kind of Values

Denotable : they can be given a name

Expressible : they can be the result of a complex expression

Storable : they can be stored in a variable

Dynamic Checking

Pros:

- some checks can only (simply) be done at runtime
- compilation is faster

Cons:

- Execution is slower
- Additional memory is required

Static Checking

Pros:

- type errors are detected before execution
- better memory usage
- faster execution

Cons:

- compilation is slower
- good programs may be rejected

Outline

Scalar Types

Booleans

- Not always expressible (no boolean type in C)
- If expressible, not always denotable or storable (e.g. W^0 and W^1)

Characters

- The character set may depend on the language: ASCII, Unicode
- The operations strongly depend on the langauge
- Usually denotable, expressible, storable

Integers

- Values: a finite number, between $[-2^t, 2^t - 1]$ for typical values of t (usually 8, 16, 32, 64, sometimes 31, e.g. OCaml)
- Sometimes unbounded integers: Scheme

Scalar Type: Floating Point Numbers

What does this program prints?

```
class Floating{
    public static void main(String [] a)
    {
        double x = 0;
        for(int i = 0; i <8; i ++){
            x += 0.1;
            System.out.println(x);
        }
    }
}
```

Scalar Type: Floating Point Numbers

What does this program prints?

```
class Floating{
    public static void main(String [] a)
    {
        double x = 0;
        for(int i = 0; i <8; i ++){
            x += 0.1;
            System.out.println(x);
        }
    }
}
```

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
```

Scalar Type: void/unit

Values

- In some languages, an empty type `void`): there is no expressible values of this type, values of this type are not denotable, not storable: C, Java
- Type `unit`: only one value, usually written '`()`'. Usually denotable, expressible, storable.
  - OCaml: `unit`, `()`
  - Racket: the value is `#<void>`, the result of function `void` that doesn't take any argument. Procedures such as `display` return `#<void>`

Outline

Composite Types

Definition (Composite Type)

A composite type is a type obtained by combining other types.

Composite Types

Definition (Composite Type)

A composite type is a type obtained by combining other types.

Common Composite Types

- array
- structure
- object/class
- pointers/references

Records

Definition (Record)

- A record is a (generally ordered) finite collection of named types called fields
- In imperative languages each field behaves like a variable of the same type
- In functional languages, records are cartesian products but where names can be used to access components of tuples

Records: Example in Scheme

```
(define-record-type point (fields x y))
```

The following procedures are automatically defined:

```
(make-point x y)     ; constructor
(point? obj)         ; predicate
(point-x p)          ; accessor for field x
(point-y p)          ; accessor for field y
```

Records: Examples

What does this program print?

```
typedef struct { float x; float y; } point;

int main(void)
{
  point p1, p2;
  p1.x = 0.0; p1.y = 1.0;
  p2.x = 0.0; p2.y = 1.0;
  printf("%d \n", p1==p2);
}
```

Variants Records and Unions

Definition (Variant Record and Union)

- A particular form of record is that in which some fields are mutually exclusive. We talk of variant record in this case.
- In C a union is a collection of fields that share the same area of storage and such that only one is active at a time

Variants and Unions

Example

```
#include "stdio.h"
typedef enum { red, green, blue } color;
typedef enum { two_D, colored, three_D } kind;
typedef struct {
  float x; float y;
  kind k;
  union { color c; float z; };
} point;
void main(int argv, char ** argc)
{
  point p;
  p.k = colored;
  p.x = 0; p.y = 0; p.c = red;
  printf ("( x=%f, y=%f, z=%f)\n", p.x, p.y, p.z);
}
```

Variants and Unions

Example

```
#include "stdio.h"
typedef enum { red, green, blue } color;
typedef enum { two_D, colored, three_D } kind;
typedef struct {
  float x; float y;
  kind k;
  union { color c; float z; };
} point;
void main(int argv, char ** argc)
{
  point p;
  p.k = colored;
  p.x = 0; p.y = 0; p.c = red;
  printf ("( x=%f, y=%f, z=%f)\n", p.x, p.y, p.z);
}
```

Variant/Union can break type safety

- In C, C++, Pascal, ...
- Type safety preserved in Ada, OCaml, Reason, ...

Arrays

Definition (Array)

- An array is an ordered homogeneous collection of data elements
- Each element is identified by its position in the collection
- Usually the elements should be of the same type"

Design Issues

- What are the types for positions?
- Are ranges checked?
- When does allocation take place?
- Multidimensional arrays?
- Non rectangular arrays?
- Initialization?

Reference to an array element

Arrays

Reference to an array element

- name of the collection + subscript(s)/indice(s) for position(s)
- C-like languages: `a[i][j]`
- Ada: `a(i,j)`
- OCaml: `a.(i).(j)`
- Perl: for array `a`, first element: `$a[0]`

Types for positions

Arrays

Reference to an array element

- name of the collection + subscript(s)/indice(s) for position(s)
- C-like languages: `a[i][j]`
- Ada: `a(i,j)`
- OCaml: `a.(i).(j)`
- Perl: for array `a`, first element: `$a[0]`

Types for positions

- C-like, OCaml-like: integer numbers (from 0)
- Pascal/Ada: integer ranges and enumeration types

Memory Allocation

Arrays

Memory Allocation

- Static array: static storage, static position range
- Fixed stack-dynamic array: allocation at declaration elaboration, static position range
- Stack-dynamic array: allocation and position range at declaration elaboration
- Fixed heap-dynamic array: same as previous but allocated in the heap
- Heap-dynamic array: size can change during the lifetime of the array

Arrays

Examples

- Static:
- Fixed stack-dynamic:
- Stack-dynamic array:
- Fixed heap-dynamic:
- Heap-dynamic:

Arrays

Shapes

- Regular/rectangular array: the rows have the same size, the columns have the same size
- Jagged array: the row may have a different size
  - C: rectangular (jagged with pointers and malloc)
  - C#/Java/ML: jagged arrays (basically array of arrays rather than multidimensional)
  - C++, Ada: both

Operations

Arrays

Operations

In most languages:

- access to an array element
- (length)

Arrays

Operations

In most languages:

- access to an array element
- (length)

Higher-level language:

- Operations on array as a whole
- Example: APL, Reason with higher-order functions
- Example: libraries for C++ using overloading
- Much easier to grasp quickly the global structure of an algorithm

Arrays

Bound Checks

- At runtime:
  - Java, Ada, C#, Python, Scheme, ...
- No check: C, C++
- Compiler switch: OCaml

Pointers Types and References

Definition (l-value)

A *l-value* is a value that represents a location in memory

Definition (Reference)

- In some languages, a variable does not contain directly a value but a l-value. The value is usual in the heap at the location denoted by the l-value.
- In Java:
  - variables for basic types are containers
  - variables for objects are references

Definition (Pointer)

- Some language do not have references, but can explicity use values of a *pointer type*
- Values of a pointer type are l-values

Pointers Types and Reference

Design Issues

- References or pointer types?
- Typed pointers or untyped pointers?
- Pointer operations:
  - deferencing
  - pointer arithmetic
- Related memory operations:
  - memory allocation
  - memory deallocation
- Main problem: pointer to unallocated memory:
  - to `null`
  - to memory non longer allocated (dangling pointer)

Outline

Type Equivalence

When two formally different types are equal?

- Equivalence by name: only when they have the same name
  Variant: weak equivalence by name (aliases are considered equal)
  Example: Pascal

```
type T1 = 1..10;
type T2 = 1..10;
type T3 = int;
type T4 = int;
```

- Equivalence by structure

Structural Equivalence

Definition (Structural Equivalence)

Structural equivalence of types is the (least) equivalence relation satisfying the following properties:

- The name of a type is equivalent to itself
- If a type T is introduced with the definition `type T = expression` (or equivalent definition in other syntax), `T` is equivalent to `expression`
- If two types are constructed by applying the same type constructor to equivalent types, then the two types are equivalent.

Outline

Summary

- Definition of type as a set of values and operations and the role of types in design, implementation and execution of programs
- Type systems as the set of constructs and mechanisms that regulate and define the use of types in a programming language
- The distinction between dynamic and static type checking
- The concept of type-safe systems, that is safe with respect to types
- The primary scalar types, some of which are discrete types
- The primary composite types, among which we have discussed records, variant records and unions, arrays and pointers: for each of these types, we have also presented the primary storage techniques
- The concept of type equivalence, distinguishing between equivalence by name and structural equivalence

Outline

Outline

Static and Dynamic Memory Management

Low Level Languages

- Simple
- Static: program + data loaded into memory before execution begins

High-Level Languages

- More complicated
- Constraints depend on the language features

Recursion and Memory Allocation

Recursion

```
int fib (int n) {
  if (n == 0)
    return 1;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) +
           fib(n-2);
}
```

- Number of active procedures depends on values known only at runtime
- Each call requires memory space for:
  - parameters
  - intermediate results
  - return addresses
  - ...
- Block: Last In First Out
  $\Rightarrow$ block activation stack

Memory Allocation

- Explicit memory allocation/deallocation (for e.g. C malloc/free)
- Calls can alternate in any order $\Rightarrow$ not possible to use a stack
- Use of a structure called *heap*

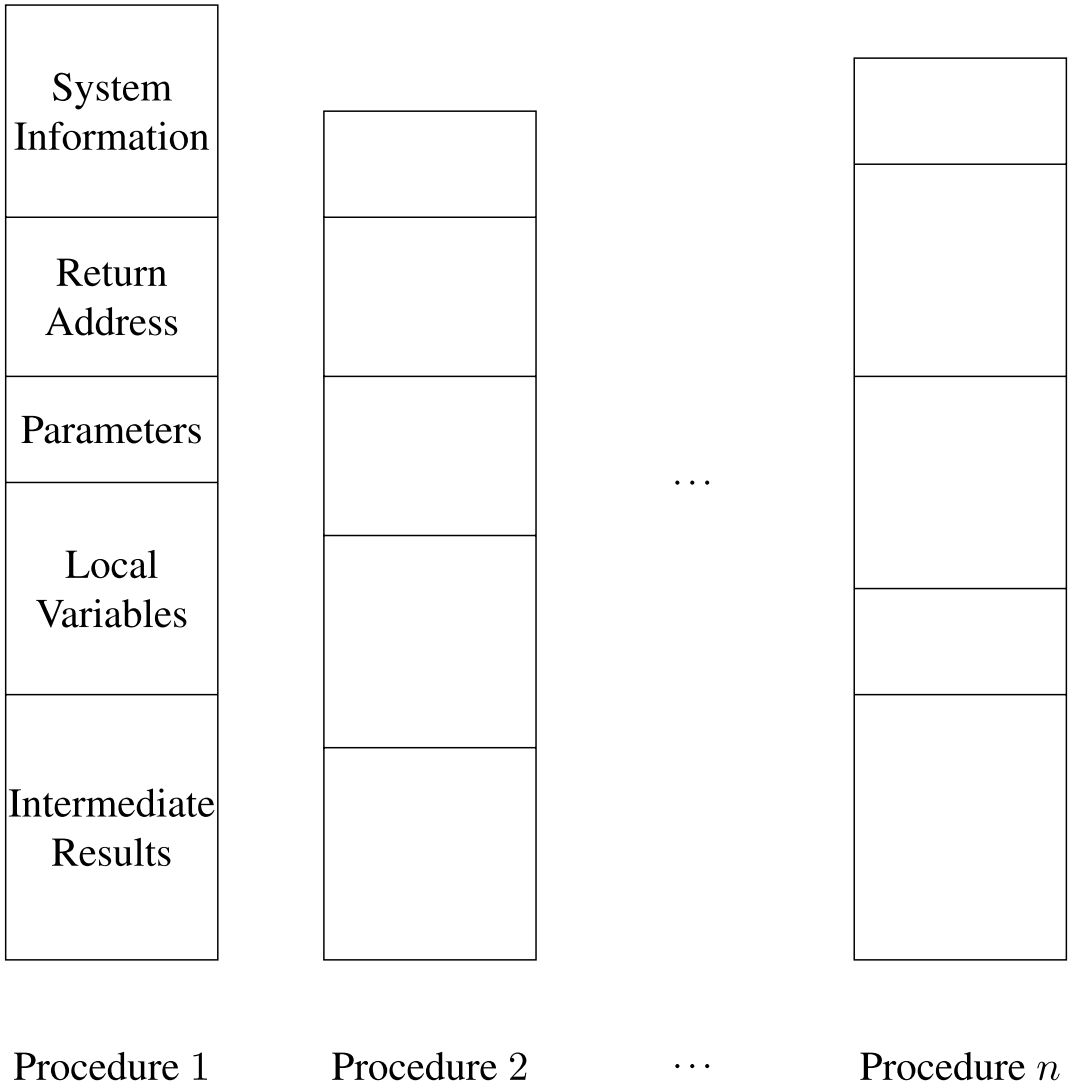
Outline

Static Memory Management

Elements that can be statically allocated

- Global variables
- Procedures instructions
- Constants (if non dependent on values known at runtime)
- Compiler generated tables for runtime support:
  - name handling
  - type checking
  - garbage collection
  - …
- Language without recursion:
  - memory for blocks and procedure (sub-routines) calls
  - it works because without recursion and without parallelism, only one call per procedure can be active at a given time

Static Memory Management



Outline

Dynamic Memory Management using Stacks

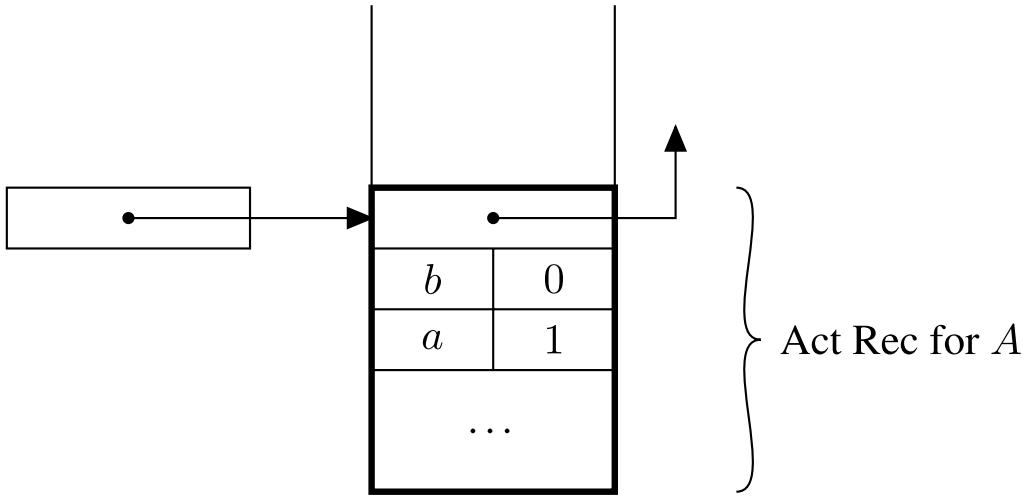
Example of Activation Record/Frame

```
A:{
   int  a = 1;
   int  b = 0;
   B:{
      int  c = 3;
      int  b = 3;
   }
   b=a+1;
}
```

Dynamic Memory Management using Stacks

Example of Activation Record/Frame

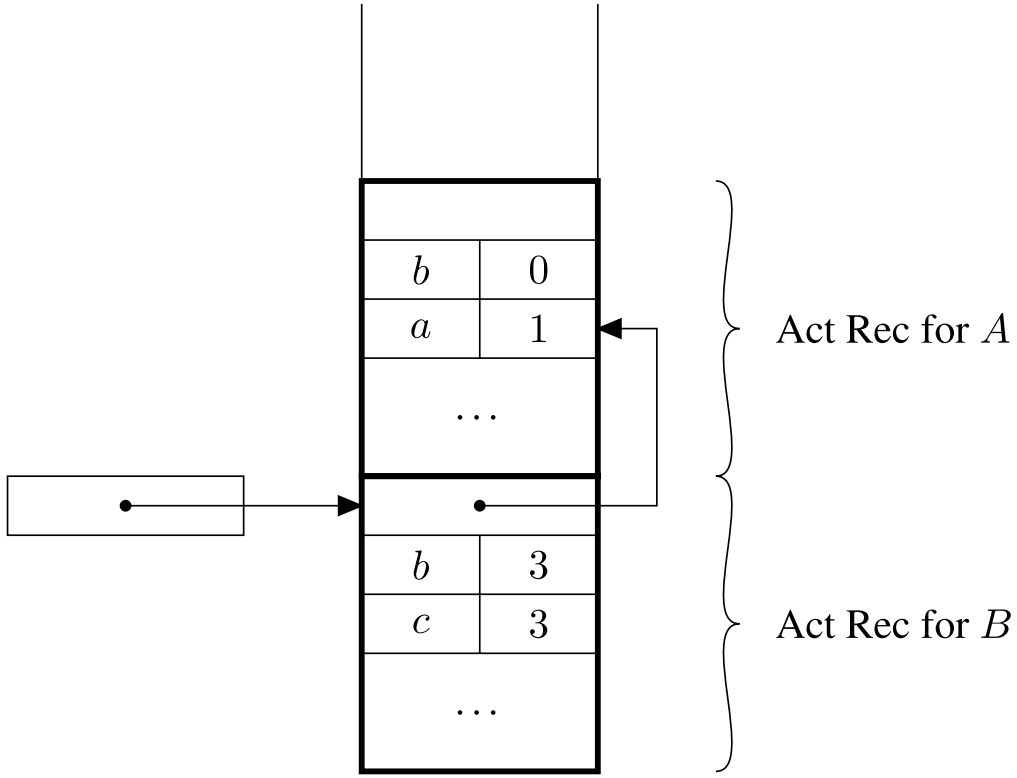
```
A:{
   int  a = 1;
   int  b = 0;
   B:{
      int  c = 3;
      int  b = 3;
   }
   b=a+1;
}
```



Dynamic Memory Management using Stacks

Example of Activation Record/Frame

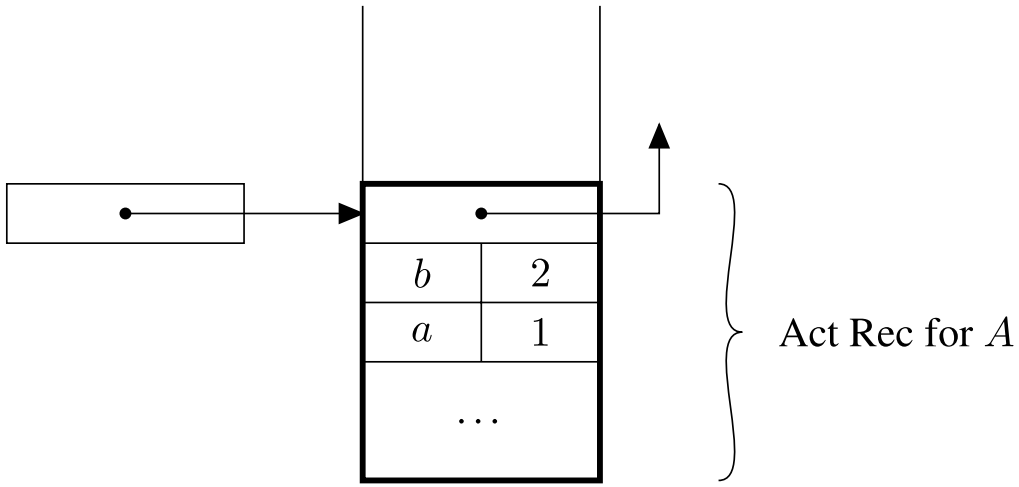
```
A:{
   int  a = 1;
   int  b = 0;
   B:{
      int  c = 3;
      int  b = 3;
   }
   b=a+1;
}
```



Dynamic Memory Management using Stacks

Example of Activation Record/Frame

```
A:{
   int  a = 1;
   int  b = 0;
   B:{
      int  c = 3;
      int  b = 3;
   }
   b=a+1;
}
```



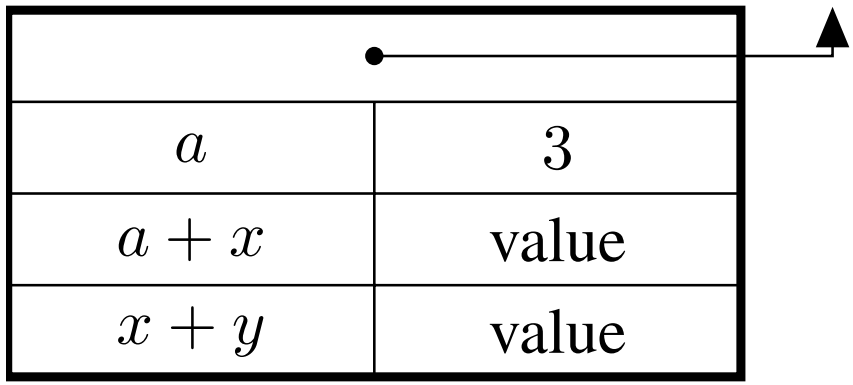
Activation Records for In-line Blocks

Activation Record

- **Intermediate results** When calculations must be performed, it can be necessary to store intermediate results, even if the programmer does not assign an explicit name to them

Example

```
{
  int a = 3;
  b = (a+x) / (x+y);
}
```



Activation Records for In-line Blocks

Activation Record

- **Local Variables**
  - Memory size: depends on the type and number of variables
  - Size information: in general determined at compiled time
  - Size information: but in some cases at runtime
    (for example dynamic arrays)
- **Dynamic chain pointer:** pointer to the *previous* activation record on the stack (or the last activation record created)

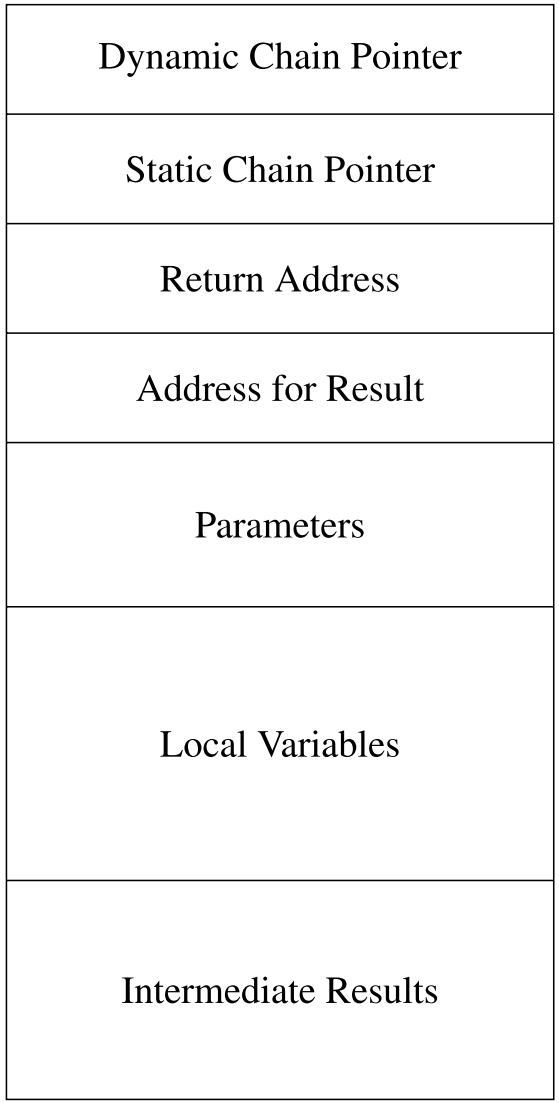
Remark: compiled languages

- both local variables and intermediate results are stored in *registers* instead of the stack, for improving performances
- there is a limited number of registers, so both the stack and registers may be used
- this phase of compilation is named **register allocation** and is based on an algorithm of graph coloring

Activation Records for Procedure Blocks

Activation Record

- **Intermediate results, local variables, dynamic chain pointer**: as in-line blocks
- **Return address:** contains the address of the first instruction to execute after the call to the current procedure/function has terminated execution
- **Returned result:** (only for functions) contains the address (inside the caller's activation record) of the memory location where the subprogram stores the value to be returned when the function terminates
- **Parameters:** the values of actual parameters used to call the procedure or function are stored here



Recursion and Activation Records

factorial

```
int fact(int n){
  if (n≤1)
    return 1;
  else
    return n*fact(n−1);
}
```

Activation Record

Dynamic Chain Ptr	
Address for Result	
`n`	
Intermediate Result	

Recursion and Activation Records

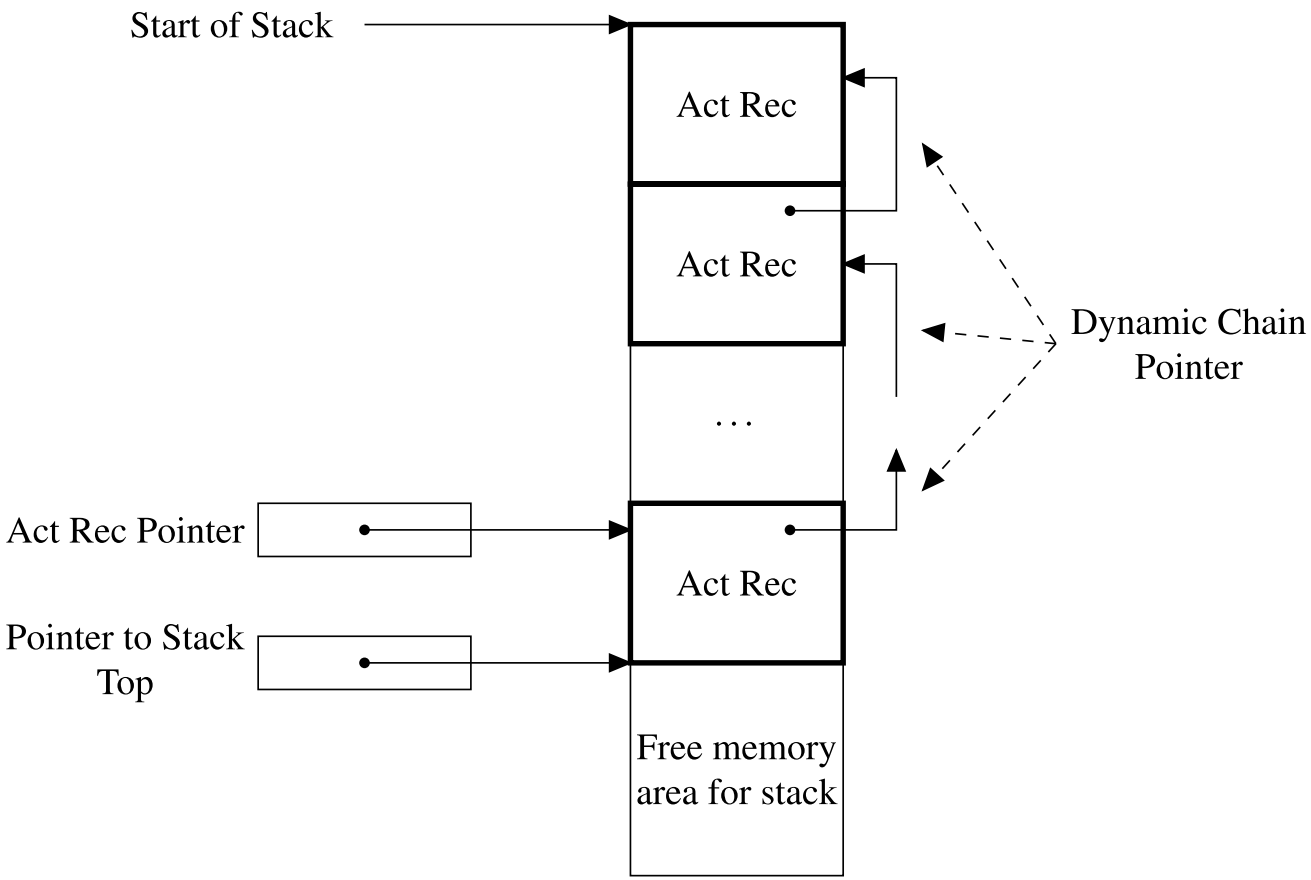
factorial version 2

```
int f(int n, int res){
  if (n≤1)
    return res;
  else
    return f(n−1, n*res);
}
int fact(int n){
  return f(n, 1);
}
```

Activation Record for f

Dynamic Chain Ptr	
Address for Result	
`n`	
`res`	

Stack Management



Activation Record Stack

- `Act Rec Pointer`: the current frame/environment
- `Ptr Stack Top`: optional if predefined size of activation records
- stack management: code fragments inserted before and after a procedure call or start/end of inline block

Stack Management

Caller/Callee

- Caller: program/procedure that performs a call
- Callee: procedure that has been called
- Both perform part of the stack management
- In the caller: *calling sequence* include the call itself and code immediately before and after
- In the callee:
  - *prologue*: to be executed just after the call
  - *epilogue*: when the procedure ends execution
- Exact distribution of code: depends on the implementation, for optimization the callee should have most of the code

Stack Management

Tasks at the Start of the Call

- **Allocation of stack space** to store the new activation record
- **Modification of program counter** to give the control to the callee, the incremented old value should be saved as the **return address**
- **Modification of activation record pointer** to set the current environment
- **Parameter passing**: done by the caller
- **Register save**, for e.g. the old activation record pointer

Stack Management

Tasks at the End of the Call

- **Update of program counter** to return the control to the caller
- **Value return**: usually should be stored in the activation record of the caller (address accessible from the activation record of the callee)
- **Return of registers**: previsouly saved registers are restored
- **Deallocation of stack space**

Remark

We omitted the data structures necessary for scope rules
(see textbook section 5.5)

Outline

Example of C Program

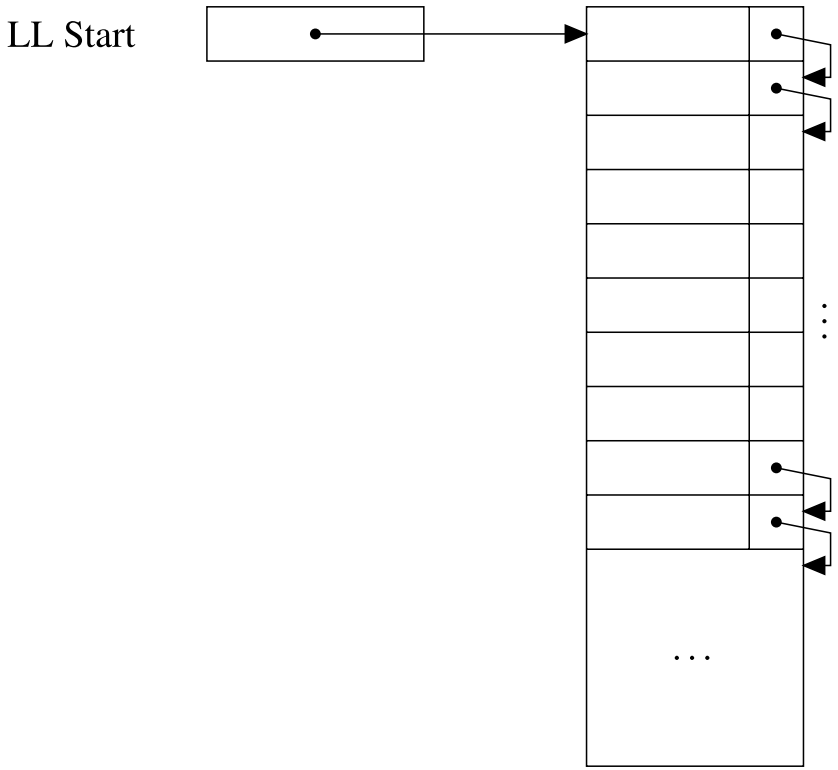
```
1 int *p, *q; /* p,q NULL pointers to integers */
2 p = malloc (sizeof (int));
3             /* allocates the memory pointed to by p */
4 q = malloc (sizeof (int));
5             /* allocates the memory pointed to by  q */
6 *p = 0;     /* dereferences and assigns */
7 *q = 1;     /* dereferences and assigns */
8 free(p);    /* deallocates the memory pointed to by p */
9 free(q);    /* deallocates the memory pointed to by q */
```

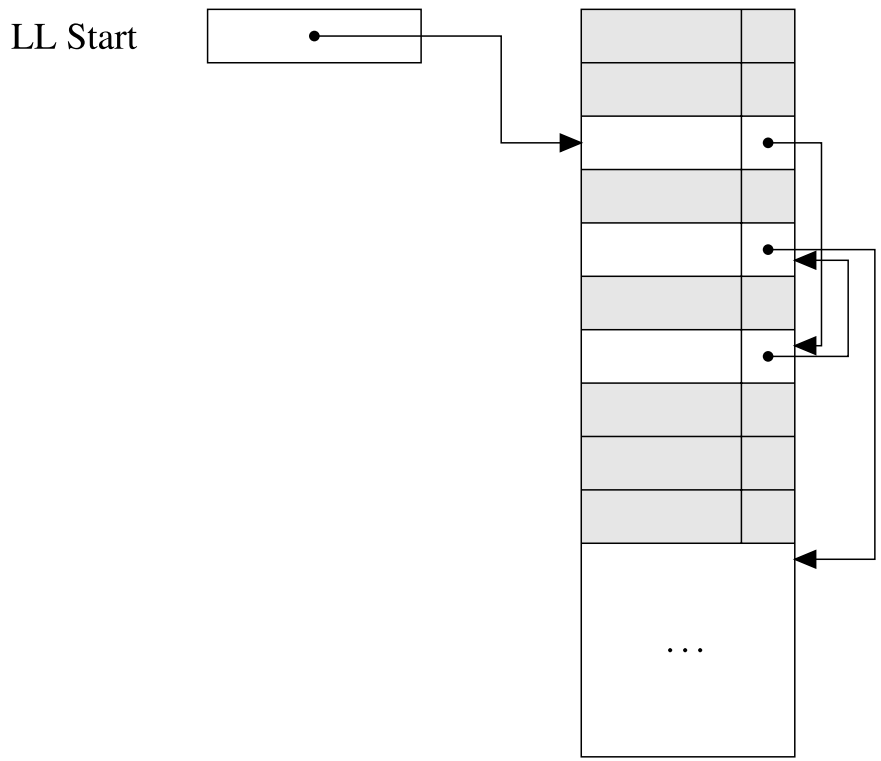
Fixed Sized Blocks

Free List

- Linked list of addresses of free blocks
- Allocation: removes the first element of the list and return the address
- Deallocation: stores back the deallocated address at the beginning of the list

Fixed Sized Blocks





Free List for Heap of Fixed Size Block

In Gray: block in use

Variable-Length Blocks

Techniques

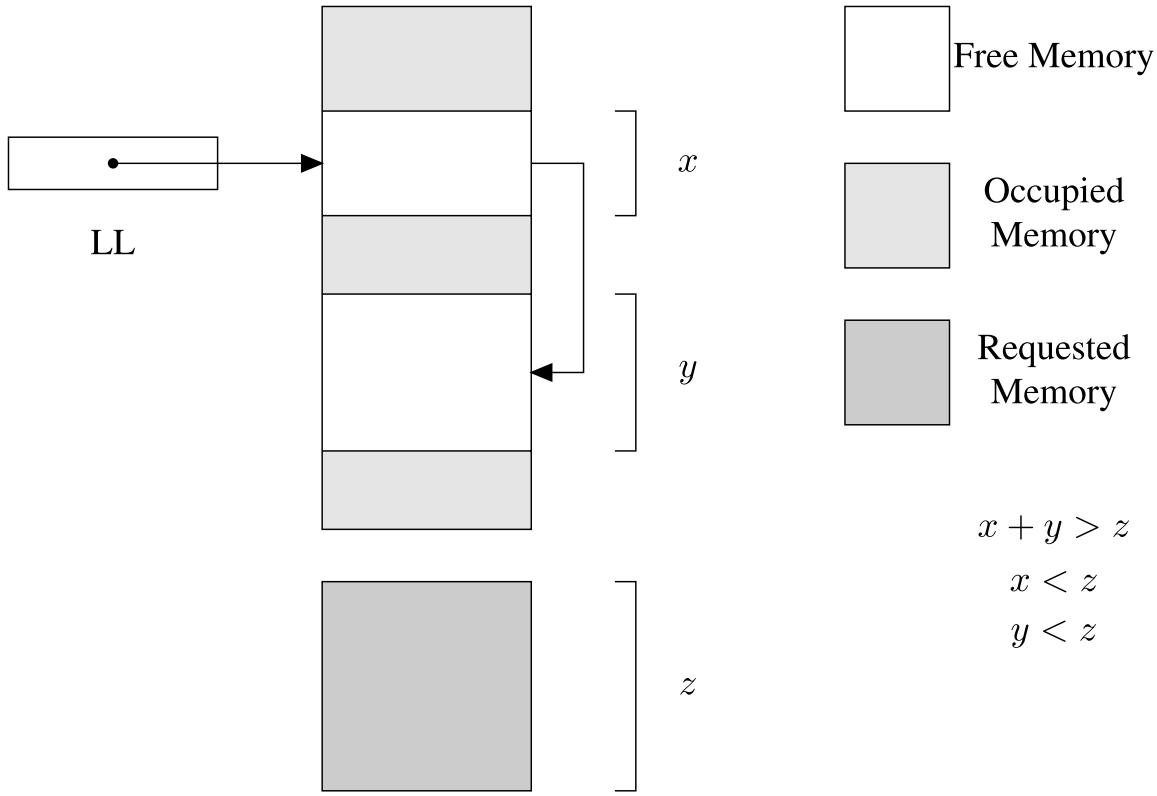
- An array requires a contiguous region
- Goal of techniques for variable-length blocks:
  - Good memory occupation
  - Good execution speed

$\Rightarrow$ rational trade-off

Variable-Length Blocks

Problems

- Internal fragmentation: the allocated memory is larger than the requested memory (for e.g. basic blocks 16 bytes, requested memory 24 bytes)
- External fragmentation: because of allocation/deallocation not all free memory can be occupied



Variable-Length Blocks

Single Free List

- List of blocks of variable size
- Threshold size: s
- Requested memory of size n: if available block of size $k \geq n$, the block can be used:
  - if $k - n \leq s$: internal fragmentation
  - otherwise a new block of size $k - n$ is added to the free list
- Search for available blocks:
  - First fit: first block of sufficient size (fast)
  - Best fit: block with smallest sufficient size (memory occupation)
  - Data structure: list ordered by block size (search faster, insertion slower)
- Deallocation: if adjacent blocks are free too, they are merged to avoid external fragmentation (partial compaction)

Variable-Length Blocks

Multiple Free Lists

- Different free lists for different sizes
- Sizes: static or dynamic
- Dynamic:
  - Buddy system (powers of 2)
  - Fibonacci heap
- If no block of a given size is available, a bigger one is split
- When two “buddy” blocks are free they are merged

Variable-Length Blocks

Buddy System

- Allocated block with size a power of 2
- 2^s is the smallest possible size
- If 2^m is the memory size there are $m - s + 1$ free lists
- Free list for blocks of size 2^k is at level/index $m - k$
- Request for allocation of a block of size n:
  - find k such that $2^{k-1} < n \leq 2^k$
  - look for the first free block in the free list of index $m - k$
  - if there is no free block, look for a free block in the list of the previous level $m - (k + 1)$, and split the obtained free block in two blocks of size 2^k
  - repeat if there is still no free block at level $m - (k + 1)$
- Request for deallocation of a block: if the "buddy" of the deallocated block is free, then they are merged

Pros and Cons

Buddy System

+ less external fragmentation than one free list
+ search for a block that fits very efficient
+ allocation/deallocation cost is low
- internal fragmentation

- Variants (mostly block size)
- Implementation details are important in practice

Memory Management: Implicit or Explicit?

In C: Explicit

- Memory allocation in the heap with `malloc`
- Memory deallocation in the heap with `free`

In Java: Implicit

- Memory allocation in the heap with `new`
- No explicit memory deallocation
- Need for automatic memory management: a GC
  - GC = Garbage Collector
  - GC = Glaneur de cellules

Outline

Summary

- Memory management depends on the language features
- Recursion requires dynamic memory management
- Static only memory management possible without recursion
- Usually a mix:
  - static memory management for some features (e.g. global variables)
  - dynamic memory management for others (for e.g. blocks)
- Stack-Based Memory Management:
  - Activation records for in-line and procedure blocks
  - Stack management by code fragments in the caller and callee
- Heap-Based Memory Management:
  - Common techniques for fixed-size and variable-size blocks
  - Fragmentation problem and how to limit it