8. Further Data Types

- Typical data types:
  - char
  - integer, shortint, longint, byte, unsigned …
  - real, longreal, (shortreal)
  - Arrays (homogeneous structure)
  - Records (heterogeneous structure)
  - Sets
  - Pointers
  - Procedure types (function types)
  - Classes (records with methods)
  - Subtypes (subtype polymorphism)
  - Type parameters (parametric polymorphism)

Real Number

- The floating point representation of a real number x is an approximation by a pair e and m such that:

  $x = B^{e-w} \times m \qquad 1 \leq m < B$

- The base B, the bias w, the exponent e, and the mantissa m usually follow one of the two IEEE standards. The standard also adds a bit s for the sign:

Type	B	w	bits for e	bits for m	total
real	2	127	8	23	32
longreal	2	1023	11	52	64

  More precisely, the number x represented is:

  $x = (-1)^s \times 2^{e-127} \times 1.m$ and $x = (-1)^s \times 2^{e-1023} \times 1.m$

Examples of Floating Point Representation

Decimal	s	e	1.m	Binary		
1.0	0	127	1.0	0	01111111	00000000000000000000000
0.5	0	126	1.0	0	01111110	00000000000000000000000
2.0	0	128	1.0	0	10000000	00000000000000000000000
10.0	0	130	1.25	0	10000010	01000000000000000000000
0.1	0	123	1.6	0	01111011	10011001100110011001101
–1.5	1	127	1.5	1	01111111	10000000000000000000000

- The value 0 is a special case, represented with all bits being zero.
- e = 0 with m ≠ 0 and e = 255 (resp. e = 1023) are invalid results, called NaN (not a number).

Compatibility Between Numeric Types …

- Consider following assignments where a:shortint, i:integer, and r: real:

  i := i + a   r := i + r   a := i   i := r

- Several possibilities for compatibility:
  - Operands of arithmetic operators and both sides of assignments must be of same type. If needed, conversion functions must be inserted explicitly by the programmer, as they affect efficiency.
  - Operands of arithmetic operators must be of same type, but on assignments implicit conversions may take place.
  - Operands of arithmetic operators may be of mixed integer type (short, normal, long) or of mixed real type, but not mixed integer and real type, since e.g. converting normal integer to long involves only a sign extension by integer to real conversion involves more.
  - For convenience, integer and real expressions may be freely mixed.

… Compatibility Between Numeric Types

- Further options for conversions:
  - implicit conversions take only place from 'smaller' to 'larger' types. Hence a notion of <u>type inclusion</u> in underlying:

    integer $\subseteq$ longint $\subseteq$ real $\subseteq$ longreal
  - implicit conversions take place from any numeric type to any other numeric type, with a check whether the result fits in the in the destination.
- Possibilities for mixed expressions:
  - the precision of an operator is the larger precision of the operands.
  - the precision of an operator is the largest possible precision; if needed, the result is converted to smaller precision on assignment.
  - the precision of an operator is the precision of the result (left hand side of an assignment).

Sets …

- Sets have been introduced (most notably in Pascal and Modula-2)
  - as a convenience for the programmer
  - to allow safe bit manipulations

- The type set of T, where T has n elements, is represented by n bits or $\lceil n / w \rceil$ words, if w is the word size. Usually n is restricted to be quite small, but at least 256 to allow set of char.

- Each set is represented by its characteristic function encoded as a bit vector. For example the set [3, 6] with n = 32 would be represented by

  00000000 00000000 00000000 01001000

… Sets

- The set operations of union, intersection, and complement are implemented by bitwise and, or, and negation.
- If a set is larger than a word, an instruction for and, or, negation has to be executed for each pair of words. For example, the union of two set of char takes 8 instructions.
- The membership test x in s is implemented by a bit test, possibly after first selecting the word to test. If such an instruction is not available, the bit sequence must be shifted appropriately with subsequent sign bit test (or zero bit test).

Pointers

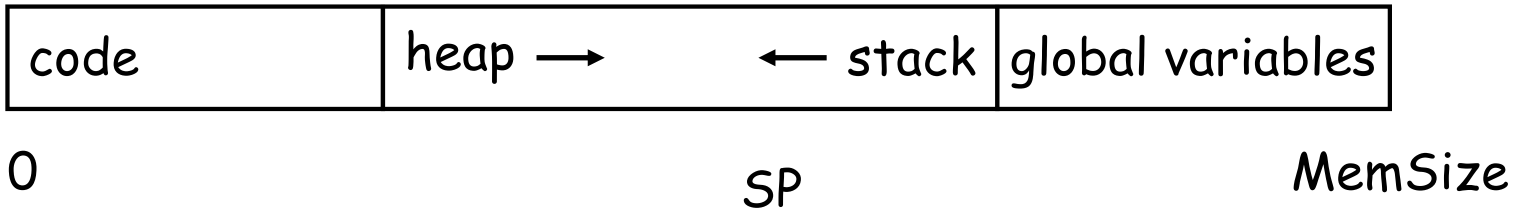
- In strongly typed programming languages, pointers are bound to a base type. For example:

```
T = record x,y : integer end;
P = ^T;
```

- A new object of type T with variable p of type P can be allocated on the heap by

```
new(p)
```

  Typically, the heap grows towards the stack:



- Given a pointer p, dereferencing it by p^ gives the object it points to, typically a record.

Nil

- Nil is a special pointer value meaning that a pointer does not point to an object. Nil is polymorphic since it can be of an arbitrary pointer type. For example:

  ```
  var p : ^integer;
  var q : ^char;
  p := nil; q := nil
  ```

  This example shows that the type of the right hand side of the assignment cannot be determined bottom up. Typically nil is implemented as zero.

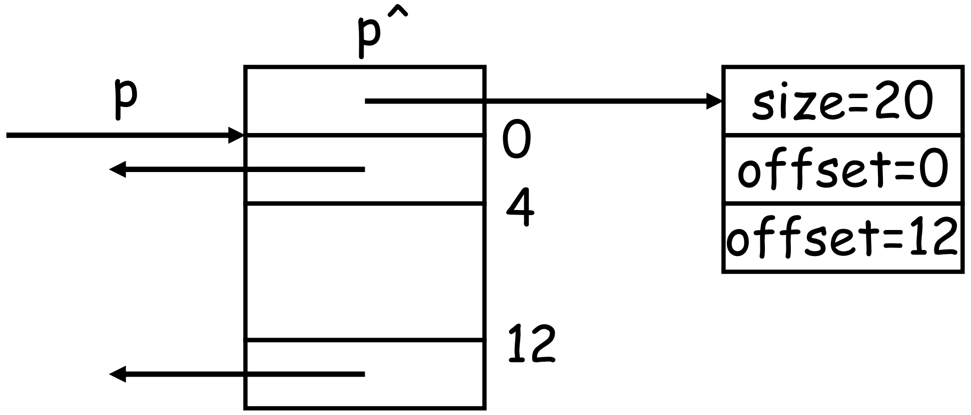
- Nil pointers must not be dereferenced. This can be easily checked by testing whether the pointer is zero or not. If this is too costly, the available memory protection mechanism can be used. If the first N bytes (for the code) are protected, then dereferencing nil will be caught. Dereferencing with subsequent indexing or field selection will be caught if the address is between 0 and N – 1.

Deallocation …

- Explicit deallocation, for example by dispose (p): The memory management reclaims the memory block occupied by p^. Explicit deallocation may lead to dangling references.

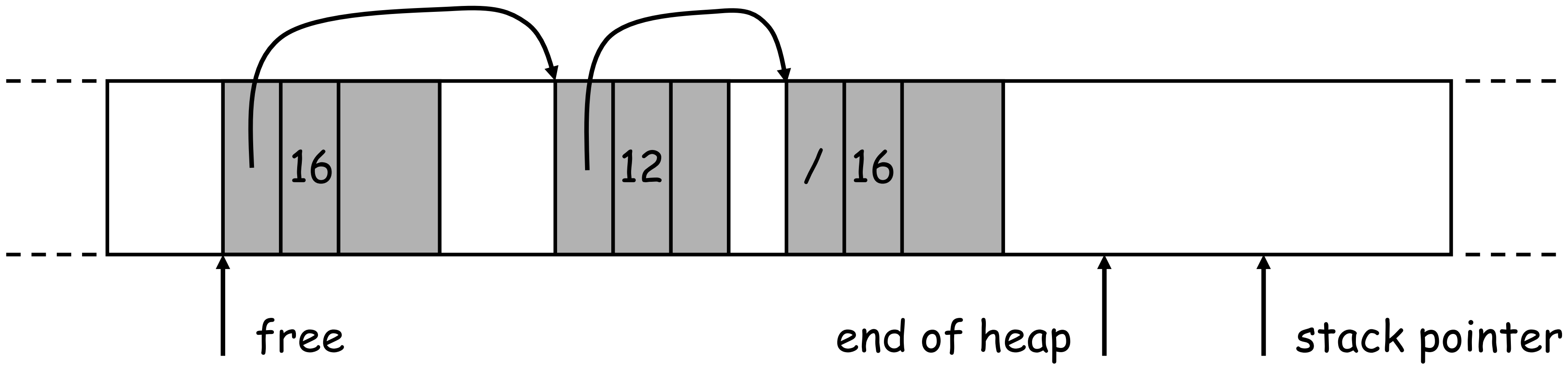
- Implicit deallocation by garbage collection: Objects on the heap which are not reachable are identified and the memory they occupy is reclaimed. For this, the garbage collector requires:
  - the addresses of all declared pointer variables,
  - the offsets of all pointer fields of objects on the heap, and
  - the size of every object on the heap.

  Every object on the heap contains a pointer, the type tag, to a type descriptor:



… Deallocation

- With both explicit and implicit deallocation, we need to keep a list of available memory blocks. These can be linked together, with each block containing its size and pointer to the next block:



- A call of dispose(p) would add p^ to the head of the list; garbage collection would similarly add blocks to the list.
- A call of new(p) would first look for a matching block in the list; if no block can be found, the heap is increased (towards the stack).

Dynamic Memory Allocation

- For the request new(p) to allocate a block alternatives exist:
  - First fit: the first fitting block is selected
  - Best fit: the best (or exact) fitting block is selected
  - Worst fit: the largest block is selected.

  If the fit is not exact, the remainder is kept in the list. The worst fit tries to avoid that the leftovers are too small to be useful.

- In Pascal the size of all data types are known at compile time – e.g. allocating an array of computed size is not possible! It is therefore possible to have one list for objects of each size, leading to constant allocation time.

- Constant allocation time can in general be achieved by standardizing the size of blocks, e.g. n, 2 x n, 4 x n, 8 x n, … bytes. At most half of the memory will be unused due to allocation of the next largest block.

Name vs. Structural Compatibility …

- Consider for example:

  ```
  type T = record i, j: integer end;
  type U = record i, j: integer end;
  var x, y: T; z: U;
  ```

  With name compatibility, the assignment x:=y is allowed by x:=z is not allowed. With structural compatibility, x:=z is also allowed.
  Restrictions of name compatibility apply to implicitly declared types:

  ```
  var x,y: record i, j: integer end; {creates a new type}
  ```

- Name compatibility is usually relaxed to allow "aliases":

  ```
  type T = integer;
  var i, j: T; {arithmetic operations on i, j are allowed}
  ```

- Pascal0 implements name compatibility: the types of two variables are compared by comparing the type pointers in the symbol table entries for those variables.

… Name vs. Structural Compatibility

- An argument for name compatibility is that programmers have better control of the operations possible of an introduced type since the name carries part of the meaning and intention. The other argument is ease of implementation.

- Comparing two types for structural compatibility requires comparing them recursively. In case of recursive types, which are introduced by pointers, care has to be taken:

```
type
   T = record x: integer; p: ^T end;
   U = record x: integer; p: ^U end;
```

  With structural compatibility, types T, U, and V as follows would also be compatible:

```
type
   T = record x: integer; p: ^T end;
   U = record x: integer; p: ^V end;
   V = record x: integer; p: ^U end;
```

Procedure Types

- Parameters can themselves be procedures. For example:

```
procedure F(x, y: real): real; …
procedure H(f: procedure (u, v: real): real);
begin a:= f(a + b, a – b)
end;
```

- With procedure types, structural compatibility is usually taken since otherwise procedure (u, v: real): real would have to be given a name first and F would have to be declared of that type.

- The type of a procedure consists of the types of the parameters plus possibly the result type, but not the name of the parameters. Usually names can be still given for readability.

- Passing a procedure parameter amounts to simply passing the address of that procedure.

Passing Procedures

- Consider following program:

```
        type T = procedure (var u: integer);
        var v: T; r: integer;
        procedure P(w: T);
        begin w(r) end;
        procedure Q;
            var a,b: integer;
            procedure R (var x: integer);
            begin x := a + b end;
        begin
            P(R); (* (1): ok *)
            v := R; (* (2): dangerous *)
            v(r); (* (3): ok *)
        end;
    ... Q; v(r) (* (4): not ok *) ...
```

Although (1) and (2) are allowed, local variables of Q are accessed outside Q. For this, besides the address of T a frame pointer to Q has to be passed as well to P.

Object-Oriented Concepts

- Object-oriented languages provide a combination of related features:
  - Classes
  - Inheritance
  - Subtyping (subtype polymorphism)
  - Object identities

- Object identities are just pointers, often in connection with garbage collection. The techniques discussed applied, except for relaxing the type disciple on pointers to allow subtying.

- Some languages feature also parametric polymorphism, which means allowing type parameters, e.g. Ada, Eiffel, C++. These are most useful if the type parameter has to be a subtype of a given type (e.g. Eiffel), leading to bounded parametric polymorphism. We do not discuss this further.

Classes

- Classes allow sets of encapsulated variables with operations on them to be grouped into a unit, very much like modules. However, unlike as with modules, several instances of a class – called objects – can exists. Hence, classes are more like records. In fact, Oberon, C++ use records for that purpose.

  A class definition specifies which of the fields - called attributes - and which of the operations - called methods - are private (encapsulated) and which are public.

- In most object-oriented languages private attributes and methods are visible only within the class. In Oberon, they are visible within the module in which the record is declared, hence facilitating mutually dependent classes.

  In both cases, the issue of visibility can be handled with simple extensions to the parser and possibly the symbol file.

Inheritance

- A <u>subclass</u> <u>inherits</u> all the methods and attributes of its <u>superclasses</u>. Examples in Oberon and Java:

```
type Point2D = record
              x,y: integer
         end;
     Point3D =  record (Point2D)
              z: integer
          end
…
var p: Point2D; q: Point3D;
begin p.x := 4; q.z := 9; (* ok *)
     p.z := 8 (* type error *)
end
```

```
class Point2D
{     public int x,y;
};
class Point3D extends Point2D
{     public int z;
};
      …
{Point2D p; Point3D q;
      p.x = 4; q.z = 9; // ok
      p.z = 8; // type error
}
```

Subtyping

- Subtyping allows an ordering $\subseteq$ among types. The subsumption rule states that:

  if $x{:}T$ and $T \subseteq U$ then $x{:}U$

  Thus variables can have several types, but there is always a unique least type. For example, with q : Point3D and p : Point2D:

  ```
  procedure move2D (var a: Point2D; dx,dy: integer);
  begin a.x := a.x+dx; a.y := a.y+dy end;
  … move2D (p, 3, 4); move2D (q, 5, 6); p := q; (* ok *)
  q := p; (* ??? *)
  ```

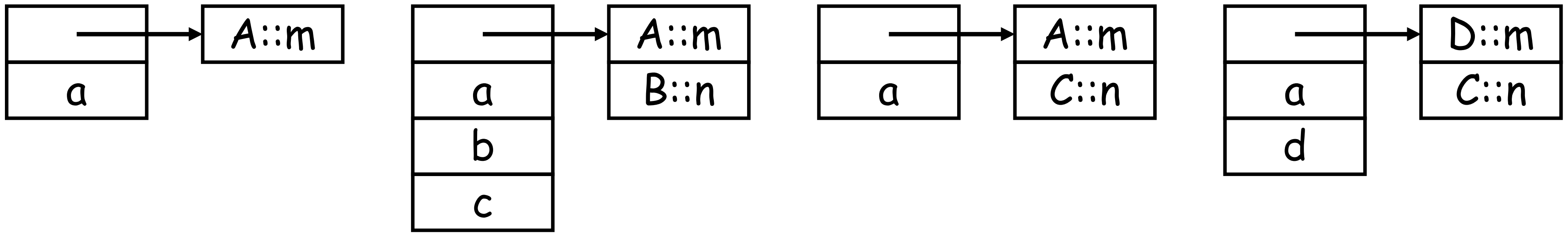
- Inheritance and subtyping are in principle independent:
  - It makes sense to inherit from a class (for reuse) without creating a subtype.
  - It is possible to create a subtype without inheriting, e.g. by having different private attributes.

  Most languages unify inheritance and subtyping.

Single Inheritance

- For single inheritance, the layout of attributes is determined by prefixing: Inherited attributes appear in same order as in the superclass before the attributes of the subclass.

```
class A              class B extends A    class C extends A    class D extends C
{int a;              {int b, c;           {n() {…};            {int d;
m() {…};             n() {…};             }                    m() {…};
}                    }                                         }
```

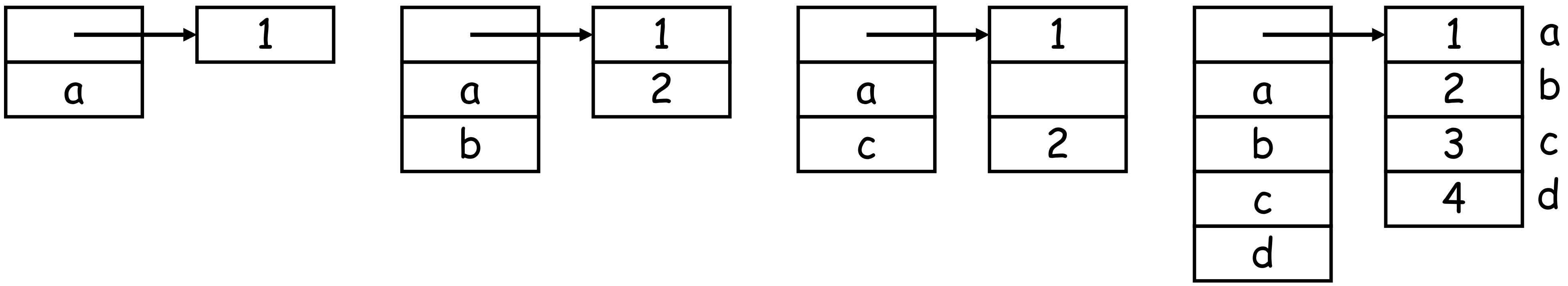


- For static methods, the declared type of the object x is used to determine the class and the method which is called on x.m().
  For dynamic methods – the usual case – the actual type of the object is used to determine the method. For this, each object has a type tag, a pointer to a class descriptor with the addresses of the methods.

Multiple Inheritance

- With multiple inheritance, the offsets of attributes and methods (rather method addresses) depend on how the class is constructed by inheritance. The class descriptors therefore also contain the field and method offsets. Example:

```
class A          class B extends A    class C extends A    class D extends B,C
{int a;}         {int b;}             {int c;}             {int d;}
```



- Compared to single inheritance, one further indirection is needed at each attribute access and method call. This can be avoided by decoupling inheritance form subtyping, e.g. as in Java.
- Also, the layout can only be determined when all classes are known, which means by the (special) linker. However, with this scheme new classes cannot be dynamically added.

Type Test

- Most languages allow to test whether the actual (dynamic) type of an object is a subtype of the declared (static) type:
  - Oberon: x is T
  - Java: x instanceof T

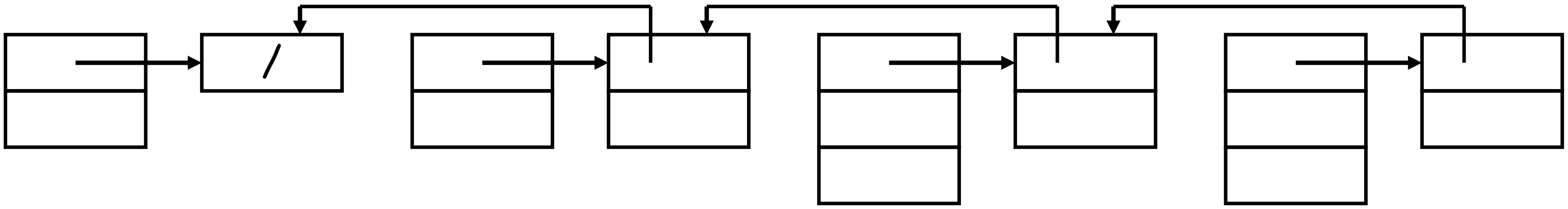
- At run-time, x contains a pointer to its type descriptor and T is represented by a pointer to the descriptor of T. If the type descriptors are augmented with pointers to their supertype, then the list starting at x has to be traversed until T or the end of the list is found. For example, if b:B, d:D, then d is C but not b is C.

class A
...

class B extends A
...

class C extends B
...

class D extends C
...



- However, this scheme works only for single inheritance.