Data Types

There are five types of liar: pathological, chronic, ambitious, student and casual; the distinction of liar emeritus is reserved for those achieving lifetime contribution to the art through innovation and public service. -- anonymous

- Primitive Data Types 5.2
  - numeric, boolean, character, others
- Complex Data Types 5.3-5.10
  - strings, enumerated types, arrays, records, unions, sets, pointers
- Type Checking 4.5-4.7
  - strong typing
  - complex type compatibility

Data Types

We've already talked about the *type* attribute of variables. We defined *type* as:

- a restriction on the kinds of values a variable can store
- an indication of how much memory a variable requires
- a definition of what operations can be done to a variable

Of course, data types are nothing new to us programmers:

- integer, real, character, boolean, pointer, bit, byte, word
- array, record, union, string, enum, pointer, file

Primitive Data Types

Primitive data types are data types that are not composed of other types.

- they are "close to the hardware" in the sense that they are implemented directly, without using other types
- they are usually words, bytes, bits
- many of the operations on them are also "close to the hardware" in the sense that they are basic machine-level operations
  - add, subtract, move, test, rotate, etc.

So is there any data abstraction *with primitive types?*

Integers...

An integer data type is a finite approximation of the infinite set of integers: {0, 1, -1, 2, -2, ...}

- *long* vs. *short* vs. *byte*
  - long (a.k.a. *long integer*, *longint*)
    - usually 32 bits
    - -2,147,483,648 .. 2,147,483,647
  - short (a.k.a. *integer*, *word*, *short integer*, *short*)
    - usually 16 bits
    - -32768 .. 32767
  - byte (a.k.a. *shortint*, *char*)
    - usually 8 bits
    - -128 .. 127

More Integers

Since the integer type is only a finite approximation of the infinite set of integers, there is a limit on the range of expressible values.

- *signed* vs. *unsigned*
  - give up the negative numbers for more positive ones
  - unsigned
    - all bit patterns are positive integers
    - 0 .. 255, 0 .. 65535, 0 .. 4,294,967,296
  - signed
    - two's complement
    - -X = logicalcomplement(X) + 1
- the integer type usually admits all numeric operations (allowing mixing with *floats*)

117

Floating-Points

A floating-point data type is a finite approximation of the infinite, non-denumerable set of real numbers.

- exponent & mantissa
  - some bits reserved for value of exponent, some for mantissa
  - mantissa: the base *B* and exponent *E* of a real number B^E are adjusted to have no digits to the left of the decimal point in *B*
- approximation
  - not just of the *range* of the set (as with integers) but also of the precision of fractional numbers (e.g. π, 0.1, etc.)
- *float* vs. *double*
  - float
    - 1 sign bit, 8 exponent bits, 23 mantissa bits
  - double
    - 1 sign bit, 11 exponent bits, 52 mantissa bits

118

Booleans

A type with only *two* legal values: True and False

- operations include the standard two-valued logic operators
  - and, or, not, xor
- *bit* vs. *byte* implementation
  - bit
    - two values require only one bit
    - implemented as hardware flag? (unlikely)
  - byte
    - implemented as an integer
    - C and assembly allow full integers in logical operations

119

Characters

An *ordinal* type whose values are interpreted as alphabetic (and other) characters

- operations may include assignment, case conversion, etc.
- ASCII
  - 7-bit standard representation for 128 common English characters
  - 8-bit extended ASCII not standard
    - includes a few arbitrarily chosen accented characters (mostly for Germanic and Romance languages)
  - used in almost all the popular programming languages
- Unicode
  - 16-bit standard representation of many alphabets
  - used in Java

120

Other Primitive Types

- Bits
  - allow bitwise operations
    - and, or, not, xor, shift, rotate
  - PL/I
  - C bit fields

    -
      ```
      struct {
        unsigned read : 1;
        unsigned write : 1;
        unsigned exec : 1;
        unsigned : 4;
        unsigned dir : 1; };
      ```
- Pointers
  - primitive in C
    - independent of any other type
    - may be manipulated as a numeric type
  - more later

121

Complex Data Types

Complex data types are data types that are composed of other types.

- a.k.a. *non-primitive types*, *structured types*, *aggregates*
- they are "far from the hardware" in the sense that they are implemented using other types, not necessarily primitive
- standard programming language operators do not necessarily apply to complete complex data strucutres

So is there any data abstraction *with complex types?*

Is it better to have lots and lots of types or only a few?

122

Strings

A *string* is a sequence of characters.

- Implementation
  - a special data type (Fortran, Basic)
    - requires operators to decompose the string into characters
  - an array of characters (Ada, C, Pascal)
    - standard array operators supported (+ possibly some string operators)
    - characters directly accessible using array element notation
    - Pascal allows comparison of strings (packed arrays) directly
  - a (linked) list of characters (Prolog)
    - allows dynamic length strings (length can easily change)
  - usually a special notation
    - here is `"a string of characters"`
    - here is the character `'c'`

String Operations

Operation	Operands	Result	Example
concatenation	string, string	string	"Dennis "+"sinneD"→"Dennis sinneD"
substring	string, int, int	string	"malefaction":2:4 → "ale"
decompose	string	array	"puck" → ['p','u','c','k']
length	string	int	"a big P" → 7
is empty?	string	boolean	"" → True
equal?	string, string	boolean	"day" = "night" → False

- Other operations are (of course) possible
  - pattern matching
  - order comparison
  - etc.

Strings and Memory

- static length strings (Ada, Pascal, Fortran)
  - the length of a string variable is fixed at the declared size
  - shorter strings stored in a string variable are padded with blanks
- limited dynamic length strings (C, C++)
  - the *maximum* length of a string variable is fixed at the declared size
  - shorter strings are stored by terminating with a NULL character
- dynamic length strings (Perl, Prolog)
  - strings may be any length, any time
  - dynamic allocation
    - more memory allocated as needed
  - hidden maximum
    - e.g.: all strings are maximum 1,024 bytes
  - semi-dynamic
    - allocate *blocks* of memory (e.g. 128 bytes) as more chars needed

Enums...

An *enum* type is a data type in which the legal values are all named in the declaration. The enumerated values are treated as symbolic constants (not strings, not chars, not numbers).

- Pascal

  -
    ```
    type day = (mon, tues, wed, thurs, fri, sat, sun);
    …
    ```

  -
    ```
    var today : day;
    today := mon;
    …
    ```

  -
    ```
    if today < thurs
    …
    ```

  -
    ```
    succ(tues) = wed
    pred(sun) = sat
    …
    ```

  -
    ```
    ord(mon) = 0
    ord(thurs) = 3
    ```

More Enums...

- Ada

  -
    ```
    type day is (mon, tues, wed, thurs, fri, sat, sun);
    type weekday is (mon, tues, wed, thurs, fri);
    …
    ```

  -
    ```
    today: weekday := mon;
    …
    ```

  -
    ```
    today < thurs
    day’(mon) <> weekday’(mon)
    …
    ```

  -
    ```
    day’succ(today) = tues
    weekday’pred(today) = ???
    …
    ```

  -
    ```
    day’pos(mon) = 0
    weekday’val(3) = thurs
    day’first = mon
    weekday’last = fri
    ```

More Enums

The enumerated values in *enum* types are symbolic constants. But it is also possible to enumerate the values of integer types.

- Subrange types
  - Pascal
    - `type tinyint = 1..10;`
  - Ada
    - `subtype tinyint is integer range 1..10;`

    -
      ```
      type day is (mon, tues, wed, thurs, fri, sat, sun);
      subtype weekday is day range mon..fri;
      ```

Where is the data abstraction *in* enums*?*

Is it possible to print *the enum symbols from within a program?*

Arrays

An *array* is a complex type made of components that are all the same type.

- the components, or *elements*, are accessed by their position relative to the beginning of the array

An array can also be thought of as a *mapping*:

- index_type → element_type
- each element of the type index_type maps to some element of the type element_type
- index_type must be a discrete ordinal type (integer, character, enum, etc.)
- element_type can usually be anything

129

Referencing Array Elements

There are two common forms of notation for accessing array elements using the *array index* (a.k.a. *array subscript*):

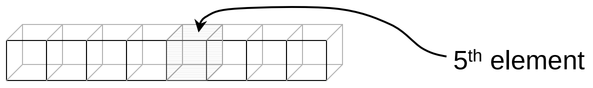
- Algol, Pascal, C:
  - `myarray[i]`
  - refers to the i^{th} element (Algol, Pascal usually) or $i+1^{th}$ element (C) of myarray
- Ada, Fortran:
  - `myarray(i)`
  - refers to the i^{th} element of myarray

Are there advantages or disadvantages to either notation?

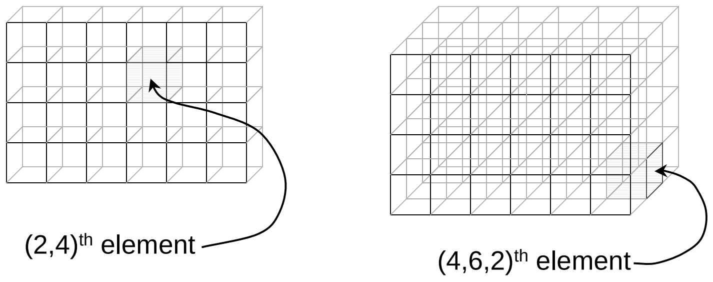
130

Multidimensional Arrays

A simple array can be thought of as a collection of elements accessed relative to the first element in the array:

- 

We also have multidimensional arrays with elements accessed by "row", "column" and higher-dimensional coordinates:

- 

Multidimensional Mapping

Just as a simple array can be thought of as a mapping:

- index_type → element_type

So can a multidimensional array be thought of as a mapping:

- $\text{index\_type}_1 \times \text{index\_type}_2 \rightarrow$ element_type
  - the two index types map on to the array element type
  - `myarray[i,j]`

or

- $\text{index\_type}_1 \rightarrow (\text{index\_type}_2 \rightarrow$ element_type)
  - index_type_1 maps on to an element which is itself an array
  - `myarray[i][j]`
    - the ith element of the array myarray is itself an array: `myarray[i]`
    - the jth element of the array `myarray[i]` is a single element

Operations on Arrays...

Simple operations on arrays:

- select an element (get its value):

  -
    ```
    int i, j, v1, v2, myvector[10], mymatrix[4][5];
    v1 = myarray[i];
    v2 = mymatrix[i][j];
    ```
- assign a whole array:

  -
    ```
    type vector is array (1..6) of real;
    vec1, vec2: vector;
    ...
    vec2 := vec1;     -- note the implicit loop
    ```
- select a slice of an array (get a subset of elements):

  -
    ```
    vec3: array (1..3) of integer;
    vec4: array (1..8) of integer;
    ...
    vec3 := vec1(4..6);
    ```

More Operations on Arrays

More complex operations on arrays are possible, but rarely built-in to the language. A couple of exceptions are:

- Fortran 90
  - `array1 + array2`
  - `array1 > array2`
  - etc.
- APL
  - row, column reversal
  - matrix transposition
  - matrix inversion
  - vector inner product

What might the semantics be for `array1 + array2`*?*

What can we say about the operators +, >, etc. in Fortran 90?

Array Index Binding...

Just as there is a binding between a primitive variable and its type, there is a binding between an array and its index type.

- *static*
  - array size fixed (∴ index range fixed at compile time)
  - static memory allocation
  - global static arrays
- *semistatic*
  - array size fixed (∴ index range fixed at compile time)
  - dynamic memory allocation
  - local (stack-dynamic) arrays

More Array Subscript Binding

- *semidynamic*
  - array size determined at runtime
  - index range fixed at runtime
  - dynamic memory allocation, but size doesn't change

  -
    ```
    get(InputVal);
    declare
      vec1: array (1..InputVal) of integer;
    ...
    ```
- *dynamic*
  - array size determined at runtime
  - index range variable
  - dynamic memory allocation
  - array size can change during execution

Implementing Arrays

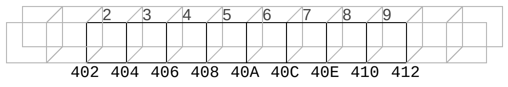
In order to use arrays, we need random access to each element in the array. That is, given an array variable, we need to be able to figure out the address of any given element in the array. To do this we need:

- the array address
  - elements in the array are stored in memory in some order starting at the address bound to the array
- the index type
  - in particular, the legal range of values and how these map to array elements
- the element type
  - the element type will give us the *size* of each element

137

Mapping One-dimensional Arrays to Memory

- `vec1: array (2..9) of integer;`
  - array address: `402h`
  - index type: `2..9`
    - first element: index = 2; second element: index = 3; ...
  - element type: `short integer`
    - each element is 2 bytes



138

Multidimensional Arrays; One-dimensional Memory

- Computer memory is one dimensional
  - that is, memory locations are numbered sequentially, with each location's address being one greater than the address of the preceding location
- Arrays can be "many-dimensional"
  - array elements can be numbered `1,1..1,N` then `2,1..2,N` etc. up to `M,1..M,N`
- In general, we may need to map an array of several dimensions down to one dimension to store it in memory

N-dimensions to One-dimension Mapping

Let's say we have a two-dimensional array of integers with four columns and three rows:

	1	2	3	4
1	23	14	6	70
2	4	11	23	33
3	16	5	13	99

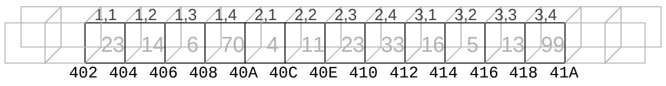
We have two choices for mapping the elements of the array to one-dimensional memory:

- *row-major*
  - increment the 2nd index from 1 to 4 for each value of the 1st index
- *column-major*
  - increment the 1st index from 1 to 3 for each value of the 2nd index

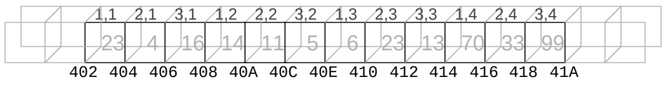
Row-Major and Column-Major

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 23 & 14 & 6 & 70 \\ 2 & 4 & 11 & 23 & 33 \\ 3 & 16 & 5 & 13 & 99 \end{array}$$

- *row-major*



- *column-major*



Row-Major Array Offsets

The address of a particular entry into an array is often called the *offset* of the element within the array.

-
  ```
  type matrix = array[6..10,3..11] of integer;
  var A: matrix;
  ```
- we have LO1 = `6`, LO2 = `3`, HI1 = `10`, HI2 = `11`
- the base address BASE is the address of `A[6,3]` or `A[`LO1`,`LO2`]`
- the number of elements in each row is NUMINROW = HI2 - LO2 + 1
- so the address of any given element `A[`I`,`J`]` is

$$\underbrace{\text{BASE}}_{\substack{\textit{starting address}\\ \textit{of the array}}} + \left\{ \underbrace{(\text{I} - \text{LO1})\times\text{NUMINROW}}_{\substack{\textit{elements in}\\ \textit{complete rows}}} + \underbrace{(\text{J} - \text{LO2})}_{\substack{\textit{elements in last}\\ \textit{(incomplete) row}}} \right\} \times \underbrace{\text{sizeof}(\texttt{integer})}_{\substack{\textit{number of bytes}\\ \textit{for each element}}}$$

Records

We defined an *array* as an *ordered* complex type made of components that are all the same type.

- a *homogeneous* aggregate

A *record* is an *unordered* complex type made of components that may be different types.

- a *heterogeneous* aggregate

Most high-level languages have some kind of *record* structure:

- Cobol, Pascal, C/C++, Ada, Java, Prolog, etc.

143

Record Examples...

Pascal

```
type bug_rec = record
  bug_num: integer;
  programmer: record
    name: packed array[1..30] of char;
    fired: boolean
  end;
  bug_date: record
    day: 1..31;
    month: (ja, fe, mr, ap,
            ma, jn, jl, au,
            se, oc, no, de);
    year: 80..99
  end
end;

...

var my_bug: bug_rec;
```

Ada

```
type bug_rec is record
  bug_num: integer;
  programmer: record
    name: string(1..30);
    fired: boolean;
  end record;
  bug_date: record
    day: 1..31;
    month: (ja, fe, mr, ap,
            ma, jn, jl, au,
            se, oc, no, de);
    year: 80..99;
  end record;
end record;

...

my_bug: bug_rec;
```

144

More Record Examples

C

```
typedef struct {
  int bug_num;
  struct {
    char name[31];
    char fired;
  } programmer;
  struct {
    char day;
    enum {ja, fe, mr, ap,
          ma, jn, jl, au,
          se, oc, no, de} month;
    int year;
  } bug_date;
} bug_rec;

...

bug_rec my_bug;
```

Cobol

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 BUG-RECORD.
   05 BUG-NUM   PICTURE IS 9999.
   05 PROGRAMMER.
      10 NAME   PICTURE IS X(30).
      10 FIRED  PICTURE IS X.
   05 BUG-DATE.
      10 DAY    PICTURE IS 99.
      10 MONTH  PICTURE IS XX.
      10 YEAR   PICTURE IS 99.
```

145

Referecing Record Fields

Whereas elements of arrays are ordered and referenced by an ordinal type (usually an integer), fields within records are unordered, and must be referenced by *name*.

- The notation for field referencing is almost universally the so-called *dot-notation*

  -
    ```
    bug_rec.bug_num = 0;
    bug_rec.bug_date.month = se;
    ```

  -
    ```
    bug_rec.bug_num := 0;
    bug_rec.bug_date.month := se;
    ```

  -
    ```
    MOVE ZEROES TO BUG-NUM IN BUG-REC.
    MOVE 'SE' TO MONTH IN BUG-DATE IN BUG-REC.
    ```

  -
    ```
    bug_rec.bug_num := 0;
    bug_rec.bug_date.month := se;
    ```

Oh no! Which language is which?

146

Omissions and Ellipsis

Some languages allow you to leave out part of the fully-qualified field names

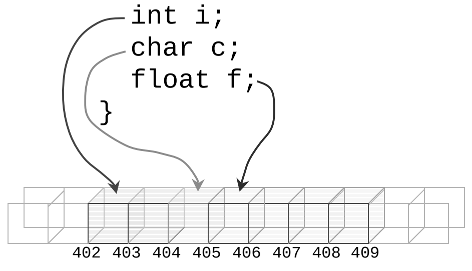
- *omission*
  - ▲ specify a common prefix to a block of statements
  - ▲
```
with bug_rec do
  bug_num := 0;
  bug_date.month := se;
end;
```
- *ellipsis*
  - ▲ don’t bother qualifying unambiguous names
  - ▲
```
MOVE ZEROES TO BUG-NUM.
MOVE ’SE’ TO MONTH.
```

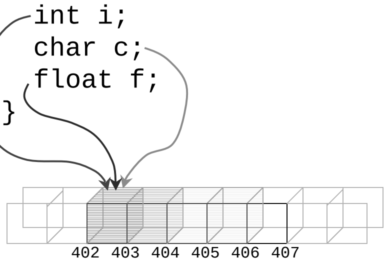
What are the advantages and disadvantages?

Unions

We already saw some examples of *unions* in our discussion of *aliasing* (♬ $_{96}$). A *union* is an unordered heterogeneous type whose elements share the same address space.

- a *record*’s fields are stored one after another in memory
- a *union*’s fields are all stored in the same place

▲
```
struct {
  int i;
  char c;
  float f;
}
```


▲
```
union {
  int i;
  char c;
  float f;
}
```


Discriminated Union

- A *tag* or *discriminant* is a variable used to keep track of which one of a union's fields is in use.

-
  ```
  type status = (married, single, divorced);
       date = record
         phnum: packed array[1..7] of char;
         birthday: record
           day: 1..31; month: 1..12; year: 60..70
         end;
         case risk: status of
           married: (spouse: packed array [1..30] of char);
           single: (nutcase: boolean);
           divorced: (money: longint)
       end;
  var my_date: date;
  ...
  if my_date.risk = divorced then
     if my_date.money < 1000000 then
        ...
  ```

149

Sets

A few languages (Pascal, Modula-2) have a built-in *set* type.

- a *set* is a *homogeneous* complex type interpreted as an *unordered* collection of values
- since a set is homogeneous, all of its elements are of the same type, called the *base type*.
  - ▲ the base type is usually restricted to some ordinal type
    ```
    type courses = (csi2111, csi2114, csi2115, csi2121, csi2131,
                    csi2165, csi2172, csi2173);
    var courses2A, courses2B: set of courses;
    ...
    courses2A := [csi2111, csi2114, csi2115, csi2165, csi2172];
    courses2B := [csi2121, csi2131, csi2165, csi2172, csi2173];
    ```
  - ▲ ordinal base types allow the set to be implemented as a *bit field*
    ```
    courses:   11111111
    courses2A: 11100110
    courses2B: 00011111
    ```

150

Operations on Sets

- union `11100110 OR 00011111`
  - `(courses2A + courses2B) = [csi2111, csi2114, csi2115, csi2121, csi2131, csi2165, csi2172, csi2173]`
- intersection `11100110 AND 00011111`
  - `(courses2A * courses2B) = [csi2165, csi2172]`
- difference `11100110 AND NOT 00011111`
  - `(courses2A - courses2B) = [csi2111, csi2114, csi2115]`
- membership `00000100 AND 11100110`
  - `(csi2165 in courses2A) = true`
- equality `00000110 AND (11100110 AND 00011111)`
  - `([csi2165, csi2172] = (courses2A * courses2B)) = true`
- inequality `NOT(00000110 AND (11100110 AND 00011111))`
  - `([csi2165, csi2172] <> (courses2A * courses2B)) = false`
- inclusion `10000110 AND 11100110`
  - `([csi2111, csi2165, csi2172] <= courses2A = true`

151

Pointers

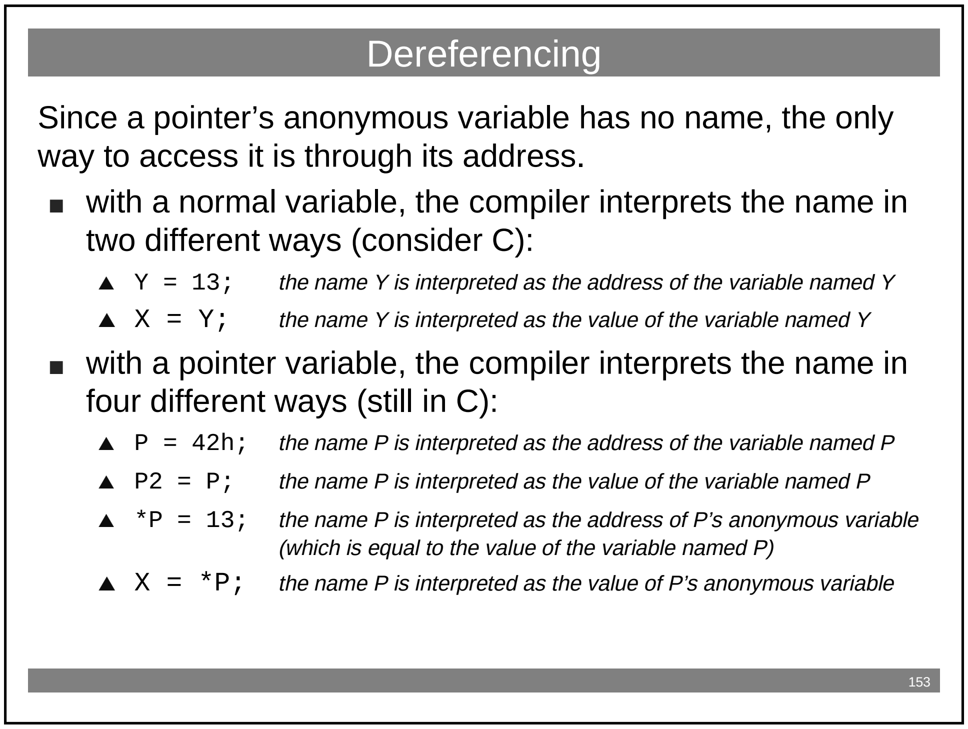
A *pointer* is usually considered a complex type consisting of two separate variables

- an *anonymous* variable
  - a variable with a type, value, address and lifetime
  - but *no name* (and therefore no scope)!
- a *pointer* variable
  - a regular old variable with a name, type, value, address, lifetime and scope
  - the value happens to be the address of its anonymous variable

This anonymous *variable is not the same as _ in Prolog!*

The primitive *pointer in C can point to anything (including existing variables, functions, whatever). When primitive, there is no anonymous variable.*

152

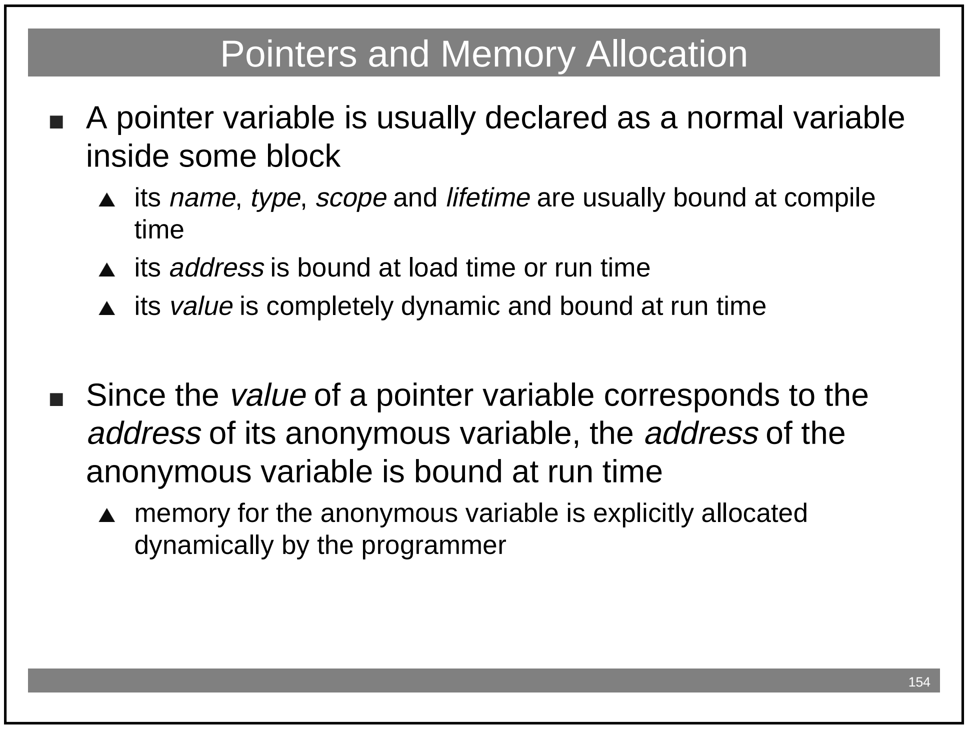


Dereferencing

Since a pointer’s anonymous variable has no name, the only way to access it is through its address.

- with a normal variable, the compiler interprets the name in two different ways (consider C):
  - `Y = 13;` *the name Y is interpreted as the address of the variable named Y*
  - `X = Y;` *the name Y is interpreted as the value of the variable named Y*
- with a pointer variable, the compiler interprets the name in four different ways (still in C):
  - `P = 42h;` *the name P is interpreted as the address of the variable named P*
  - `P2 = P;` *the name P is interpreted as the value of the variable named P*
  - `*P = 13;` *the name P is interpreted as the address of P’s anonymous variable (which is equal to the value of the variable named P)*
  - `X = *P;` *the name P is interpreted as the value of P’s anonymous variable*

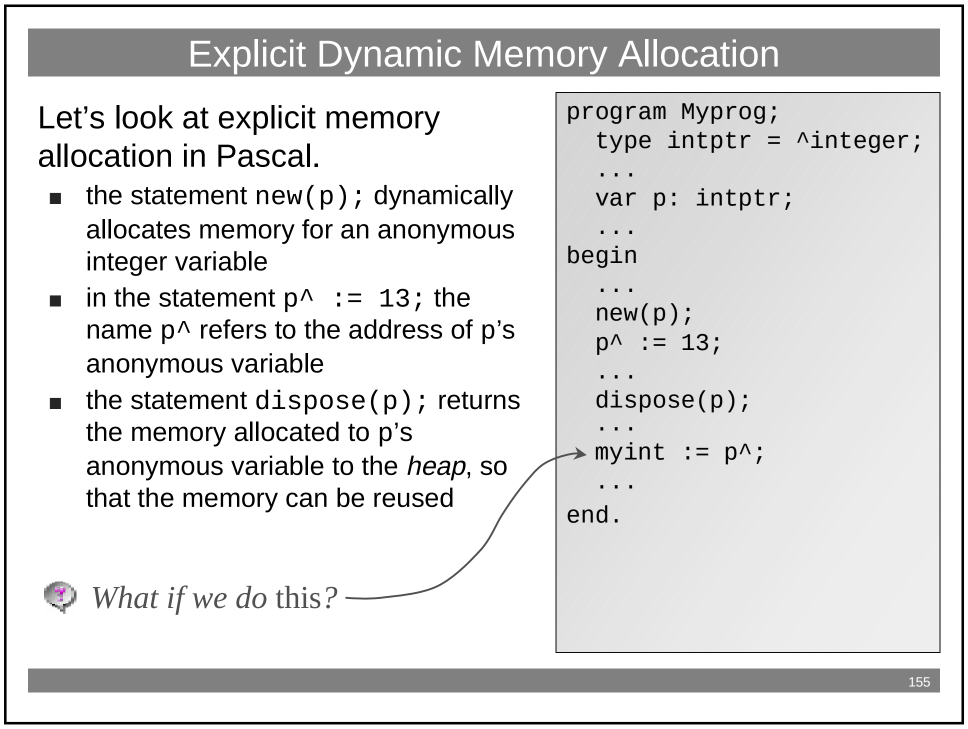
153



Pointers and Memory Allocation

- A pointer variable is usually declared as a normal variable inside some block
  - its *name*, *type*, *scope* and *lifetime* are usually bound at compile time
  - its *address* is bound at load time or run time
  - its *value* is completely dynamic and bound at run time
- Since the *value* of a pointer variable corresponds to the *address* of its anonymous variable, the *address* of the anonymous variable is bound at run time
  - memory for the anonymous variable is explicitly allocated dynamically by the programmer

154



Explicit Dynamic Memory Allocation

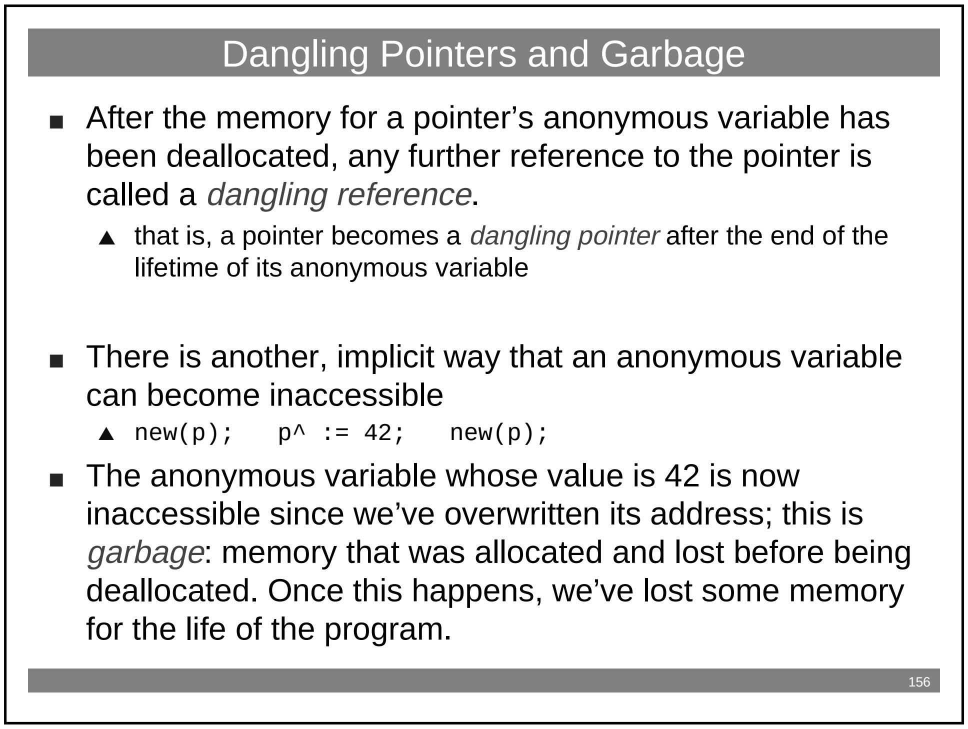
Let’s look at explicit memory allocation in Pascal.

- the statement `new(p);` dynamically allocates memory for an anonymous integer variable
- in the statement `p^ := 13;` the name `p^` refers to the address of p’s anonymous variable
- the statement `dispose(p);` returns the memory allocated to p’s anonymous variable to the *heap*, so that the memory can be reused

```
program Myprog;
  type intptr = ^integer;
  ...
  var p: intptr;
  ...
begin
  ...
  new(p);
  p^ := 13;
  ...
  dispose(p);
  ...
  myint := p^;
  ...
end.
```

What if we do this?

155



Dangling Pointers and Garbage

- After the memory for a pointer’s anonymous variable has been deallocated, any further reference to the pointer is called a *dangling reference*.
  - that is, a pointer becomes a *dangling pointer* after the end of the lifetime of its anonymous variable
- There is another, implicit way that an anonymous variable can become inaccessible
  - `new(p);   p^ := 42;   new(p);`
- The anonymous variable whose value is 42 is now inaccessible since we’ve overwritten its address; this is *garbage*: memory that was allocated and lost before being deallocated. Once this happens, we’ve lost some memory for the life of the program.

156

Garbage Collection

Garbage can be created in different ways:

```
var
  p: ^integer;
begin
  new(p);
  ...
  p^ := 5;
  ...
  new(p);
  ...
```

```
void myfunc()
{
 int *p, i;
 ...
 i = sizeof(int);
 p = malloc(i);
 ...

}
```

```
grow(L1, L1) :-
  length(L1, Len),
  Len >= 1000.

grow(L1, L2) :-
  length(L1, Len),
  Len < 1000,
  append(L1, L1, LTemp),
  grow(LTemp, L2).
```

The process of automatically reclaiming inaccessible storage space is called *garbage collection*. 5.10.10

Type Checking

Programming can be thought of largely as the application of *operators* to *operands*:

- boolean operators
  - `=`, `<>`, `<`, `>`, `and`, `or`, `not`, etc.
- arithmetic operators
  - `+`, `-`, `*`, `/`, `mod`, `**`, `sin`, `cos`, `abs`, etc.
- string operators
  - `cat`, `subs`, `length`, etc.
- also... assignment, subprogram calls, etc.

Type checking ensures that the *operands* involved in an operation are of an appropriate type.

Automatic Type Checking

In general, it is considered a "bad thing" to attempt to use an operator with inappropriate operand types. To help the programmer avoid bad behaviour, the compiler attempts to assist.

- if the variable → type binding is static (occurs at compile time), it should be possible for the compiler to check the appropriateness of operand types during compilation
- if the variable → type binding is dynamic (occurs at run time), then we need a sophisticated run time environment to check types at run time
  - the compiler may have to generate extra code to do this checking

What type checking can Pascal not *do at compile time?*

Strong Typing

A programming language has *strong typing* if all type errors can be *detected*

- preferably at compile time
- acceptably at run time

Note that not all type errors are equally serious:

```
var x: real;
    a: (c, d, g, rw, lw);
begin
  x := sqrt(37.5);
  a := x * 3.1;
  ...
```

```
var x: real;
    y: integer;
begin
  y := 5;
  x := y * 3.1;
  ...
```

Type Casting and Coercion

In a strictly strongly typed programming language, both of the examples would result in type errors. To avoid a compile error, the programmer would have to convert one of the values:

-
  ```
  var x: real;
      y: integer;
  begin
    y := 5;
    x := real(y) * 3.1;
  ```

Explicitly converting a type to another type is called *casting*.

Most programming languages recognize that some type incompatibilities are forgivable and relieve the programmer of having to cast by doing type casting automatically. Automatic type casting by the compiler is called *type coercion*.

Type Compatibility

Type casting and coercion allow us to compare two different primitive types when appropriate. Two types are *compatible* if one type can be converted to the other type.

- *integer* and *floating-point* types are an obvious example of compatible types

Complex types may also be compatible, but the compatibility is much less obvious

- when are two array types compatible?
- when are two record types compatible?
- are subranges compatible?
- are subtypes compatible?
- what about user-defined types?

Compatibility by Name...

In each of these Pascal examples, the variables `X` and `Y` are compatible.

-
  ```
  type vector = array[1..10] of integer;
  var X, Y: vector;
  ```

-
  ```
  type vector = array[1..10] of integer;
  var X: vector;
      Y: vector;
  ```

-
  ```
  var X, Y: array[1..10] of integer;
  ```

What about this one?

-
  ```
  var X: array[1..10] of integer;
  var Y: array[1..10] of integer;
  ```

More Compatibility by Name

The previous examples show *compatibility by name*: two types are compatible if they have the exact same type name.

- the Pascal example:

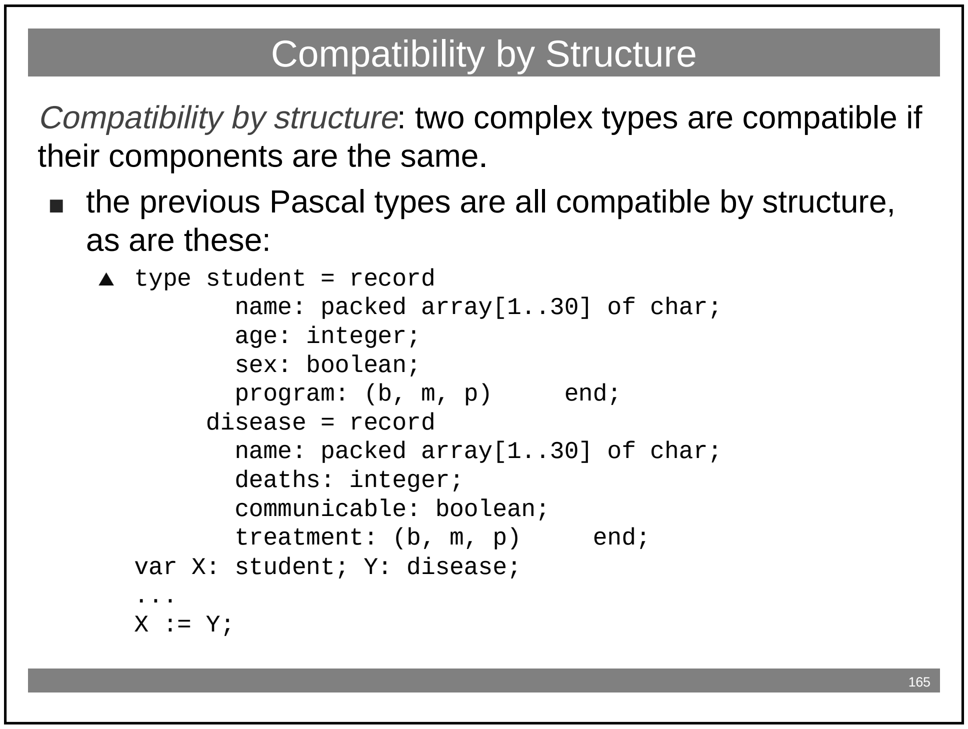
  -
    ```
    var X: array[1..10] of integer;
    var Y: array[1..10] of integer;
    ```

  is just shorthand for:

  -
    ```
    type t1 = array[1..10] of integer;
         t2 = array[1..10] of integer;
    var X: t1;
        Y: t2;
    ```
- other Pascal incompatibilities (`X` not compatible with `Y`):

  -
    ```
    type days = (mon, tue, wed, thu, fri);
    var X: days;  Y: (mon, tue, wed, thu, fri);
    ```

  -
    ```
    type myint = -32768..32767;
    var X: myint; Y: integer;
    ```



Compatibility by Structure

Compatibility by structure: two complex types are compatible if their components are the same.

- the previous Pascal types are all compatible by structure, as are these:

  -
    ```
    type student = record
           name: packed array[1..30] of char;
           age: integer;
           sex: boolean;
           program: (b, m, p)     end;
         disease = record
           name: packed array[1..30] of char;
           deaths: integer;
           communicable: boolean;
           treatment: (b, m, p)     end;
    var X: student; Y: disease;
    ...
    X := Y;
    ```

165