Variables, records, and pointers

Lubomír Bulej
KDSS MFF UK

Variable = named storage location

Values stored as sequences of bytes

- Type determines storage size and layout
  - Also the set of legal values and operations
- Location in memory = address of the first byte
  - Compiler determines where to store values
- Variables provide symbolic names to addresses

```
var
  i : integer;
  d : double;
  a : array [1..5]
      of word;
```

A	A+4	...	A+12	A+14	A+16	A+18	A+20
i	d		a[1]	a[2]	a[3]	a[4]	a[5]
4 B	8 B		2 B	2 B	2 B	2 B	2 B

Alignment on modern processors

Certain memory accesses may be inefficient/illegal

- Depends on address and access size
  - Exact criteria depend on processor architecture
  - Access size typically powers of 2, related to register size
- Memory can be efficiently (sometimes only) accessed at addresses aligned to access size
- Affects layout of variables in memory!
  - Ensure that a value in memory can be read or written to efficiently (single memory access).

```
var
  i : integer;
  d : double;
  a : array [1..5]
      of word;
```

A	A+4	A+8	...	A+16	A+18	A+20	A+22	A+24
i		d		a[1]	a[2]	a[3]	a[4]	a[5]
4 B	4 B	8 B		2 B	2 B	2 B	2 B	2 B

Records/structures = composite values

Group of related variables

- Access to variables (fields) inside a record through the name of the record variable
- Laid out together in memory
- Each field has a fixed **offset** from the **base address** of the record

```
type Person : record
  name, surname : string [15];
  age : integer;
  sex : char;
end;

var
  child, adult: Person;

begin
  ...
  child.age := 5;
  adult.age := 21;
end.
```

A	A+16	A+32	A+36
name	surname	age	sex
1 + 15 B	1 + 15 B	4 B	1 B

Example: FAT directory entry

Describes file in a directory

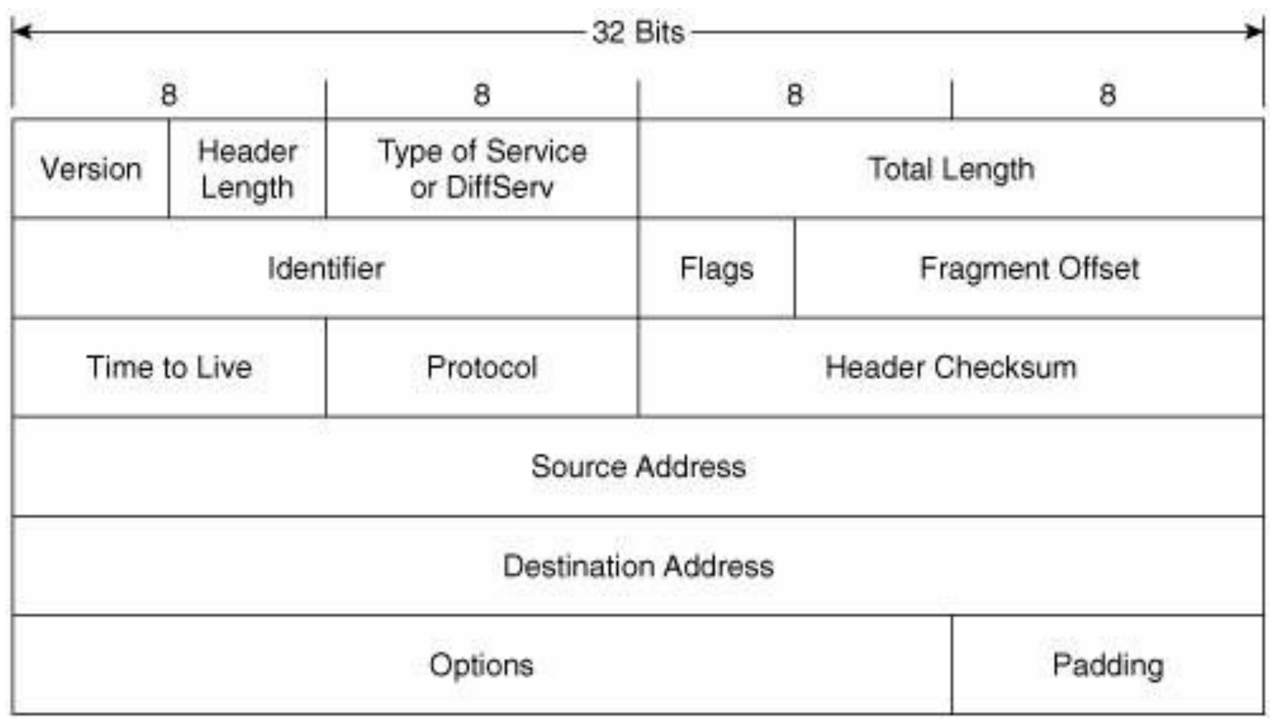
- File name and extension
- Special file attributes
- Time and date of creation
- Date of last access
- Date and time of last modification
- Location on disk (cluster number)
- File size in bytes

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	File name								Extension			Atr		Create T		
10	Cre D		Acc D				Update DT				Cluster		File size			

Example: IP packet header

Describes IP packet

- Version and length of the header
- Total length of the packet
- Source and destination addresses
- …

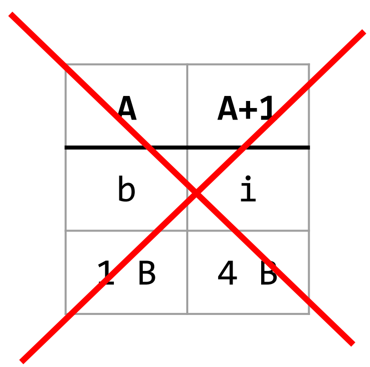


Alignment within record

Fields within a record are aligned too

- Typically the closest power of 2 greater than or equal to field size (alignment size)
  - Free Pascal: word-aligned (2 bytes) by default
  - Free Pascal: packed records are byte-aligned
- Record size is not necessarily the sum of field sizes
  - Due to field alignment within records

```
type R : record
  b : byte;
  i : integer;
end;
```



A	...	A+4
b		i
1 B	3 B	4 B

The size of things

The `SizeOf()` function

- Returns the size of a type in bytes
- Predefined for base (primitive) types
- Computed for arrays
  - array [LB .. UB] of T → (UB - LB + 1) × SizeOf (T)
- Computed for records
  - $\sum_i$ SizeOf (f_i) is only lower bound due to alignment
  - Sum of the record's last field's *offset* and this field's size, rounded up to a multiple of the record's *required alignment* (the alignment of the record's field with the largest alignment size)

Example: alignment within a record

Type declaration

```
type TItem = record
   field0 : Byte;
   field1 : array [1 .. 3] of Word;
   field2 : Single;
   field3 : Byte;
   field4 : QWord;
end;
```

Primitive type sizes

```
SizeOf (Byte) = 1
SizeOf (Word) = 2
SizeOf (Single) = 4
SizeOf (QWord) = 8
```

What are the field offsets?

What are the field alignments?

What is the record alignment?

What is the result of SizeOf (TItem)?

Abstraction of an address

Pointer

- Type providing an abstraction of an address
  - We don’t need to know the address to use it
- Pointer variable stores an address of a value
  - Typed pointer points to a value of specific type

Pascal

- Pointer type declaration `type PInteger = ^integer;`
- Pointer variable definition `var pi : PInteger;`
- Dereferencing a pointer to access the pointed-to value `i := pi^;`
- Taking an address of a variable `pi := @i;`

Basic pointer example

```
type
  PInteger = ^integer;

var
  i, j : integer;
  pi, pj, p : PInteger;

begin
  i := 1; WriteLn (i);
  pi := @i; WriteLn (pi^);
  pi^ := 2; WriteLn (i);

  j := 42; WriteLn (j);
  pj := @j; WriteLn (pj^);
  j := 84; WriteLn (pj^);

  p := pi; WriteLn (p^);
  p^ := 0; WriteLn (i);

  p := pj; WriteLn (p^);
  p^ := -1; WriteLn (j);
end.
```

Address	Contents
0x3000	i
0x3004	j
0x3008	pi = 0x3000
0x300C	pj = 0x3004
0x3010	p

Basic pointer example

```
type
  PInteger = ^integer;

var
  i, j : integer;
  pi, pj, p : PInteger;

begin
  i := 1; WriteLn (i);
  pi := @i; WriteLn (pi^);
  pi^ := 2; WriteLn (i);

  j := 42; WriteLn (j);
  pj := @j; WriteLn (pj^);
  j := 84; WriteLn (pj^);

  p := pi; WriteLn (p^);
  p^ := 0; WriteLn (i);

  p := pj; WriteLn (p^);
  p^ := -1; WriteLn (j);
end.
```

Address	Contents
0x3000	i
0x3004	j
0x3008	pi = 0x3000
0x300C	pj = 0x3004
0x3010	p = 0x3000

Basic pointer example

```
type
  PInteger = ^integer;

var
  i, j : integer;
  pi, pj, p : PInteger;

begin
  i := 1; WriteLn (i);
  pi := @i; WriteLn (pi^);
  pi^ := 2; WriteLn (i);

  j := 42; WriteLn (j);
  pj := @j; WriteLn (pj^);
  j := 84; WriteLn (pj^);

  p := pi; WriteLn (p^);
  p^ := 0; WriteLn (i);

  p := pj; WriteLn (p^);
  p^ := -1; WriteLn (j);
end.
```

Address	Contents
0x3000	i
0x3004	j
0x3008	pi = 0x3000
0x300C	pj = 0x3004
0x3010	p = 0x3004

Example: linked list

```
type
  TNode = record;
  PNode = ^TNode;

  TNode = record
    value : integer;
    next : PNode;
  end;

var
  list : PNode;
```

Physical layout

- One of several possible...

Address	Contents
0x100	value = 1
0x104	next = 0x400
	...
0x200	list = 0x100
	...
0x300	value = 3
0x304	next = 0x0 (NIL)
	...
0x400	value = 2
0x404	next = 0x300

Logical view

- “Chain” of records