Software Tools

C, Unix (Linux), and tools

Structures

- Similar to *objects* in languages like java or C++
  - But much simpler
- They are collections of data that naturally fit together
- The data can be (and usually are) disimilar
  - Intgers, floats, strings, functions or other structures
- Can be copied, assigned, passed as arguments and returned by functions

Defining Structures

- A couple of examples

```
struct point {
   int x;
   int y;
   char *name;
}; *declares a data type*
```

```
struct {
   int x;
   int y;
   char *name;
} pt; /*defines
a variable*/
```

```
struct point {
   int x;
   int y;
   char *name;
} pt; /*defines a data
type and a variable*/
```

```
struct point pt;
/*defines a variable*/
```

Terminology

- A *tag* is an optional name for the structure.
- The members are `x` and `y`.
  - AKA fields
- The structure member operator is the dot (.)
  - Or the arrow (->) for pointers

```
struct point {
   int x;
   int y;
};
struct point pt;
pt.x=3;
pt.y=pt.x+1;
```

Nesting Structures

- Structures can go practically anywhere any other data type can go, including inside another structure.

```
struct point {
  int x;
  int y;
};
struct rect {
  struct point pt1;
  struct point pt2;
};
struct rect screen;
screen.pt1.x=3;
```

Copying-Assigning Strucures

- Structures behave like other data types
  - They are just bigger
- Can be passed as arguments to functions
- Can be returned by functions

```
struct point pt1, pt2;
pt1.x=3;
pt1.y=pt.x+1;
Pt2 = pt1;
```

```
struct point ptadd(struct point pt1,
                   struct point pt2)
{
  struct point pt;
  pt.x = pt1.x+pt2.x;
  pt.y = pt1.y+pt2.y;
  return pt;
}
struct point pt1, pt2, pt3;
pt3 = ptadd(pt1,pt2);
```

Copying-Assigning Strucures

- Copying and assigning structures is OK for small structures
- For bigger ones it is faster to pass around pointers

```
struct point *ptadd(struct point *ppt1,
                    struct point *ppt2)
{
  struct point *ppt;
  ppt = pointalloc();
  ppt->x = ppt1->x+ppt2->x;
  ppt->y = ppt1->y+ppt2->y;
  return ppt;
}
struct point pt1, pt2, *ppt3;
ppt3 = ptadd(&pt1,&pt2);
```

Allocating Structures

- Very often we need to create more structures
- We have to allocate space for them

```
struct point *pointalloc()
{
  struct point *ppt;
  ppt = (struct point *)malloc(sizeof(struct point));
  return ppt;
}
```

Structures containing Pointers to Themsleves

- Very often needed for lists, trees, etc
- Sometimes two structures have pointers to each other

```
struct point {
  int x;
  int y;
  struct point *next;
} *pointlist;
```

```
struct s {
  int x;
  int y;
  struct t *sister;
};
struct t {
  int u;
  int v;
  struct s *sister;
};
```

Unions

- Very much like structures but all members occupy the same space
  - No it is not a joke

```
typedef enum {NUM,OP,NIL} ttag;
struct enodestruct
{
  ttag tag;
  union
  {
    int num;
    struct
    {
      otag optype;
      enode l, r;
    } opstruct;
  } data;
};
```

Typedef

- It is too much work to type thingsl like
  - This: `struct enodestruct *e;`.
- We can use

```
typedef enum {NUM, OP, NIL} ttag;
typedef struct enodestruct *enode;
enode e;
```

Bit-fields

- We often need to manipulate bitfields
- The obvious way is easy but awkward.
- Bitfields are like other elements
- Mostly...
- We cannot:
  - Take address
  - Treat them as arrays
- The are not very portable
  - But with careful coding, they can be.

```
struct flagstruct {
  unsigned int key:2;
  unsigned int ext:1;
  unsigned int sta:1;
} flags;
```