Structures, Unions and Bit-fields in C

Objectives

Be able to use compound data structures in programs

Be able to pass compound data structures as function arguments, either by value or by reference

Be able to do simple bit-vector manipulations

Structures

- **Arrays require that all elements be of the same data type.**
- **Many times it is necessary to group information of different data types**
  - **An example is a materials list for a product (list typically includes a name for each item, a part number, dimensions, weight, and cost)**
- **C supports data structures that can store combinations of character, integer floating point and enumerated type data**
- **They are called a *struct*.**

Structures

Compound data:

A date is

- an `int month` **and**
- an `int day` **and**
- an `int year`

```
struct ADate {
   int  month;
   int  day;
   int  year;
};

int main(){
        struct ADate date;

        date.month = 1;
        date.day = 18;
        date.year = 2018;
        return 0;
}
```

Declaring structures

```
struct tag {
        member-list
} variable-list;
```

Any one of the three portions can be omitted

```
struct ADate{
   int  month;
   int  day;
   int  year;
}date;
int main(){
    date.month = 1;
    date.day = 18;
    date.year = 2018;
    return 0;
}
```

```
struct ADate {
   int  month;
   int  day;
   int  year;
};
int main(){
   struct ADate date;
   date.month = 1;
   date.day = 18;
   date.year = 2018;
   return 0;
}
```

```
struct {
   int  month;
   int  day;
   int  year;
}date;
int main(){
   date.month = 1;
   date.day = 18;
   date.year = 2018;
   return 0;
}
```

typedef

```
typedef struct {
        member-list
} type-name;
```

/* omit both tag and variables */

This creates a simple type named 'type-name' (more convenient than struct tag)

```
struct tag {
        member-list
} variable-list;
```

```
typedef struct {
   int  month;
   int  day;
   int  year;
}Date;
int main(){
        Date date;
        date.month = 1;
        date.day = 18;
        date.year = 2018;
        return 0;
}
```

Typedef

Mechanism for creating new type names

- **New names are an alias for some other type**
- ***May* improve clarity and/or portability of the program**

```
typedef int Length;
typedef struct ADate {
   int month;
   int day;
   int year;
} Date;

Length i = 100;
Date d = { 1, 18, 2018 };
```

Overload existing type names for clarity and portability

Simplify complex type names

Structure Representation & Size

`sizeof(struct …)` =

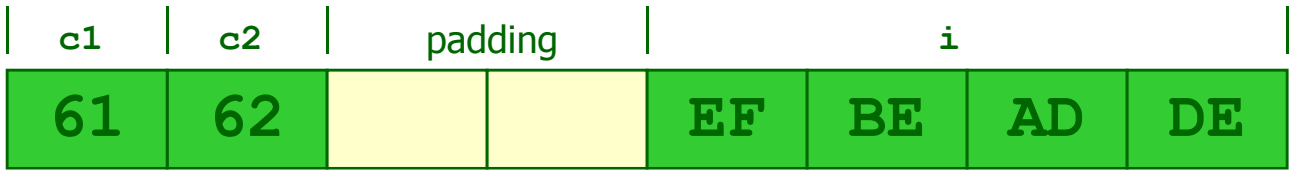
sum of `sizeof(field)`

\+ alignment padding

Processor- and compiler-specific

8 bytes

```
struct CharCharInt {
    char  c1;
    char  c2;
    int   i;
} foo;

foo.c1 = 'a';
foo.c2 = 'b';
foo.i  = 0xDEADBEEF;
```



x86 uses “little-endian” representation

Structure Representation & Size

`sizeof(struct …)` =

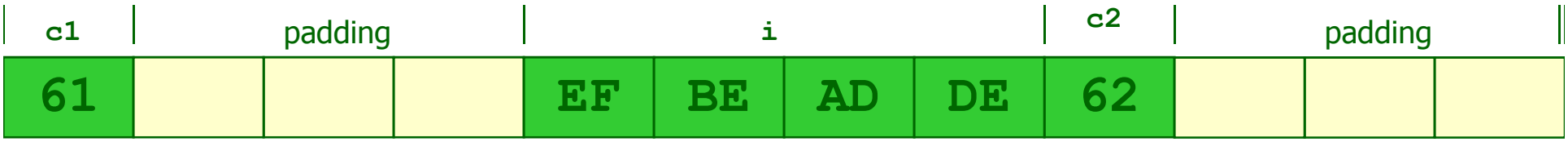
sum of `sizeof(field)`

+ alignment padding

Processor- and compiler-specific

12 bytes

```
struct CharIntChar {
   char  c1;
   int   i;
   char  c2;
} foo;

foo.c1 = 'a';
foo.c2 = 'b';
foo.i  = 0xDEADBEEF;
```

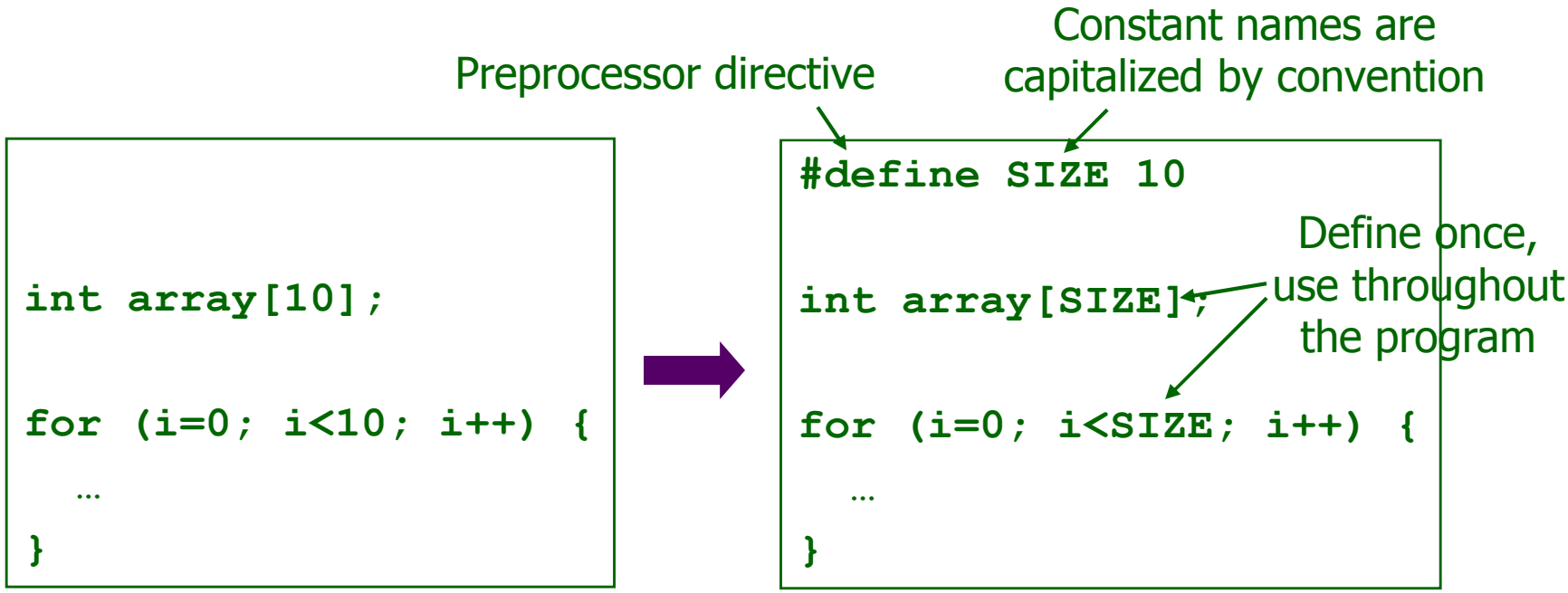


x86 uses “little-endian” representation

Constants

Allow consistent use of the same constant throughout the program

- Improves clarity of the program
- Reduces likelihood of simple errors
- Easier to update constants in the program



Arrays of Structures

```
typedef struct {
    int  month;
    int  day;
    int  year;
}Date;
```

Array declaration

Constant

```
Date birthdays[NFRIENDS];

int
check_birthday(Date birthdays[], int N, Date today)
{
  int i;

  for (i = 0; i < N; i++) {
    if ((today.month == birthdays[i].month) &&
        (today.day == birthdays[i].day))
      return i;

  return -1;
}
```

Array index, then structure field

Anonymous Structures

```
typedef struct {
   struct Point{
       int x;
       int y;
   }g;
   int r;
} Circle;
```

```
int main(){
        Circle c;
        Point p;
        c.g.x = 7;
        c.g.y = 8;
        c.r = 5;
        p.x = 9;
        p.y = 7;
}
```

```
dx = c.g.x - p.x;
dy = c.g.y - p.y;
d = dx*dx + dy*dy;
if(d <= r*r )
       printf("Inside");
else
       printf("Outside");
```

Anonymous Structures

```
typedef struct {
   struct {
       int x;
       int y;
   }g;
   int r;
} Circle;
```

```
int main(){
        Circle c;
        int x, y;
        c.g.x = 7;
        c.g.y = 8;
        c.r = 5;
        x = 9;
        y = 7;
}
```

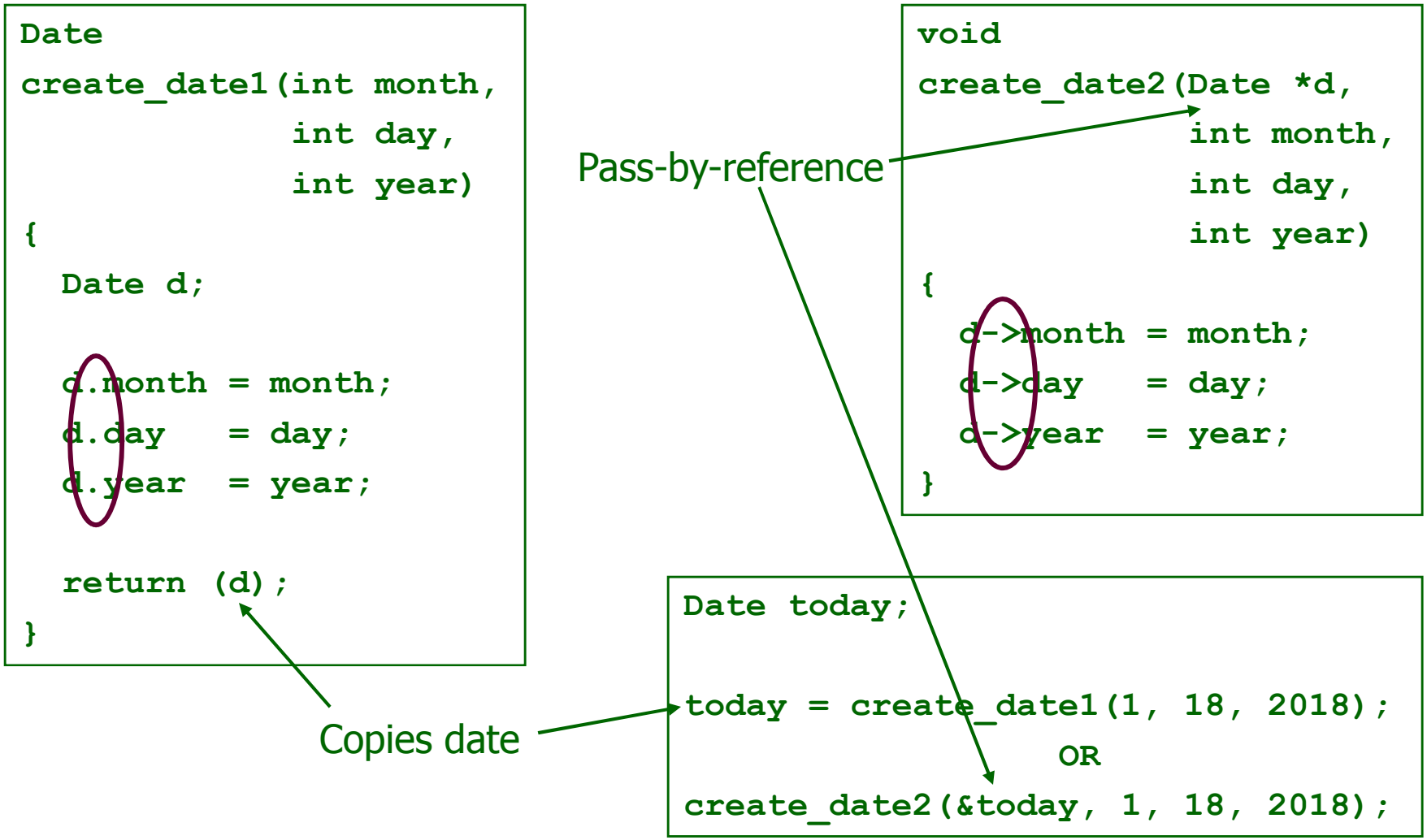
```
dx = c.g.x - x;
dy = c.g.y - y;
d = dx*dx + dy*dy;
if(d <= r*r )
        printf("Inside");
else
        printf("Outside");
```

Anonymous Structures

```
typedef struct {
   struct {
       int x;
       int y;
   };
   int r;
} Circle;
```

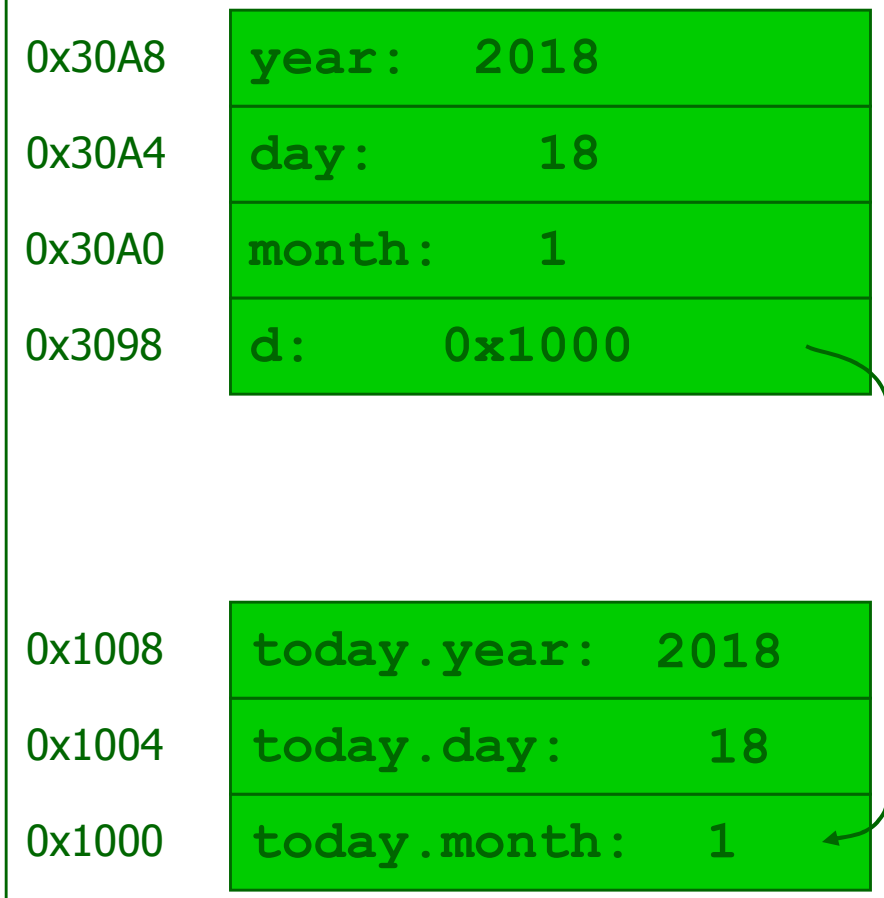
```
int main(){
        Circle c;
        c.x = 7;
        c.y = 8;
        c.r = 5;
}
```

Pointers to Structures



Pointers to Structures (cont.)

```
void
create_date2(Date *d,
             int month,
             int day,
             int year)
{
  d->month = month;
  d->day   = day;
  d->year  = year;
}

void
fun_with_dates(void)
{
  Date today;
  create_date2(&today, 1, 18, 2018);
}
```



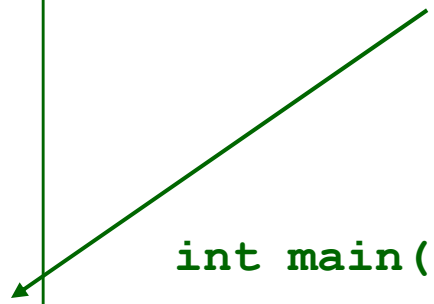
Pointers to Structures (cont.)

```
Date *
create_date3(int month,
             int day,
             int year)
{
  Date *d;

  d->month = month;
  d->day   = day;
  d->year  = year;

  return (d);
}
```

What is d pointing to?!?!
(more on this later)

Pointers to Structures (cont.)

```
Date *
create_date3(int month,
             int day,
             int year)
{
  Date *d;
  d = malloc(sizeof(Date));
  d->month = month;
  d->day   = day;
  d->year  = year;

  return (d);
}
```



```
int main( ){

Date *p;

p = create_date3(1, 18, 2018);

printf("%d %d %d", p->month,
        p->day, p->year);
free(p);

}
```

Unions

- **Like structures, but every member occupies the same region of memory!**
  - **Structures: members are "and"ed together**
  - **Unions: members are "xor"ed together**

```
union VALUE {
  float f;
  int i;
  char *s;
};
/* either a float xor an int xor a string */
```

Unions

- Up to programmer to determine how to interpret a union (i.e. which member to access)

- Storage
  - size of union is the size of its largest member
  - avoid unions with widely varying member sizes; for the larger data types, consider using pointers instead
- Initialization
  - Union may only be initialized to a value appropriate for the type of its first member

Unions

Choices:

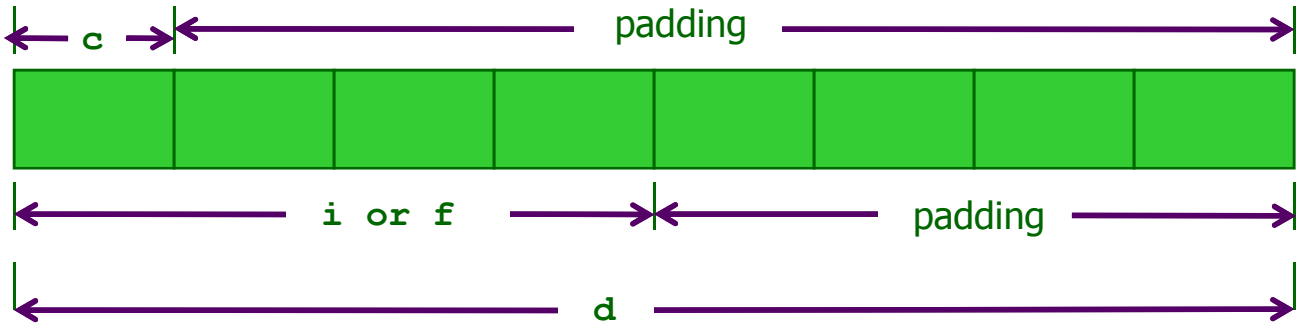
An element is

- an `int i` **or**
- a `char c` **or**
- a `float f` **or**
- a `double d`

`sizeof(union …)` =
maximum of `sizeof(`**field**`)`

```
union allType {
   int   i;
   char  c;
   float f;
   double d;
} u;
int main(){
        scanf("%d",&u.i);
         ...
        ...
        scanf("%c",&u.c);
        ...
        ...
        scanf("%f",&u.f);
        ...
        ...
        scanf("%lf",&u.d);
}
```

Unions



Unions (for inspecting bytes of int)

```
union intChar {
    int   i;
    unsigned char  c[4];
} n;
int main(){
      scanf("%d",&n.i);
      printf("%d %d %d %d",
              n.c[0],n.c[1],n.c[2],n.c[3]);

}
```

Unions (idea can be used for encryption)

```
union intChar encrypt(union intChar x){
      char t;
      t = x.c[0];
      x.c[0] = x.c[1];
      x.c[1] = t;
      t = x.c[2];
      x.c[2] = x.c[3];
      x.c[3] = t;
      return x;
}
```

```
union intChar {
    int   i;
    unsigned char c[4];
} n;
```

```
int main(){
      n.i = 0x00231115;
      n = encrypt(n);
      printf("%0x", n.i);
}
```

Bit-field Structures

Special syntax packs structure values more tightly

Padded to be an integral number of words

- **Placement is compiler-specific**
- **Bit0fileds can only be int or unsigned int**

```
struct StudentInfo {
   unsigned int  v:2;
   unsigned int  t:1;
   unsigned int  c:1;
   unsigned int  g:1;
} s;

s.v = 3;
s.t = 0;
s.c = 1;
s.g = 0;
printf("%d %d %d %d",s.v,s.t,s.c,s.g);
```

Bit-field Structures (be careful about signed extension)

Left most bit is treated as sign bit

- **Extended with sign bit**
- **Thus a '11' become**

1111 …. ….. … 1111

which is '-1' and not '3'

```
struct StudentInfo {
   int  v:2;
   int  t:1;
   int  c:1;
   int  g:1;
} s;

s.v = 3;
s.t = 0;
s.c = 1;
s.g = 0;
printf("%d %d %d %d",s.v,s.t,s.c,s.g);
```

Prints -1 0 -1 0

Restrictions applied to bit-field

The following restrictions apply to bit fields. You cannot,

- Define an array of bit fields.
- Take the address of a bit field.
- Have a pointer to a bit field.

structure + union + bit-field

```
union charToBinary{
    unsigned char n;
    struct {
        unsigned a:1;
        unsigned b:1;
        unsigned c:1;
        unsigned d:1;
        unsigned e:1;
        unsigned f:1;
        unsigned g:1;
        unsigned h:1;
    };
}x;
```

See bit pattern of a character ASCII code

```
int main( ){
    x.n = 'a';
    printf("%d%d%d%d%d%d%d%d", x.h, x.g,x.f,x.e,x.d,x.c,x.b,x.a);

}
```

structure + union + bit-field

Student ID at BUET

```
union StduentID{
    unsigned int n;
    struct {
        unsigned int r:12;
        unsigned int d:8;
        unsigned int y:8;
    };
}x;
```

```
int main( ){
        x.n = 0x1206003;
        printf("Year : %d, Department: %02d, Roll: %03d",x.y,x.d,x.r);
}
```