More about structures

ESC101: Fundamentals of Computing

Nisheeth

Passing Struct to Functions

- When a struct is passed directly, it is passed by copying its contents
  - Any changes made inside the called function are lost on return
  - This is same as that for simple variables
- When a struct is passed using pointer
  - Change made to the contents using pointer dereference are visible outside the called function

Functions Returning Structures

```
struct point make_pt (int x, int y) {
        struct point temp;
        temp.x = x;
        temp.y = y;
        return temp;    }

void print_pt (struct point pt) {
        printf(“%d  %d\n”, pt.x, pt.y); }

int main() {
        int x, y;
        struct point pt;
        scanf(“%d%d”, &x,&y);
        pt = make_pt(x,y);
        print_pt (pt);
        return 0;  }
```

```
struct point {
        int x; int y;
};
```

Even though not returning anything, make_pt is still able to do the job using pointers

Functions Returning Structures

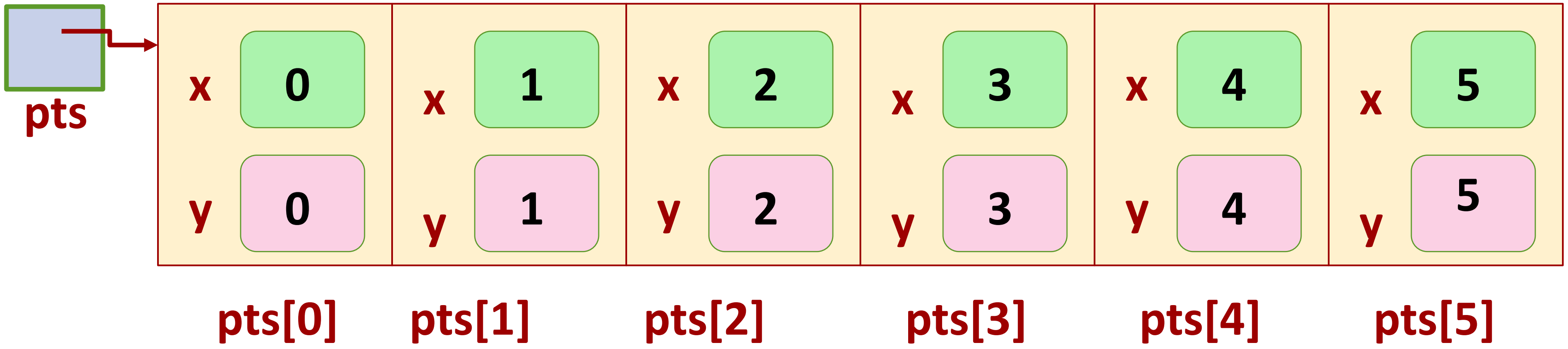
```
void make_pt(int x, int y, struct point *temp) {
        temp->x = x;
        temp->y = y;
}

void print_pt(struct point *pt) {
        printf("%d  %d\n", pt->x, pt->y);
}

int main() {
        int x, y;
        struct point pt;
        scanf("%d%d", &x,&y);
        make_pt(x,y, &pt);
        print_pt(&pt);
      return 0;
}
```

```
struct point {
             int x; int y;
};
```

Dynamic Allocation of Struct

- Similar to other data types
- sizeof(…) works for struct-s too

```
struct point* pts;
int i;
pts = (struct point*) malloc(6 * sizeof(struct point));
for (i = 0; i < 6; i++)
        pts[i] = make_point(i, i);
```



Self-Referential Structures

A field within a structure can even be a pointer to another variable of that structure type

```
struct Node{
    float x;
    struct Node *next;   // The next node in the list
};
```

Note: Invalid to have a structure with a field that is another variable of that structure type

Self-referential structures can useful in many programs, such as linked lists and trees
(will look at linked-lists in later lectures)

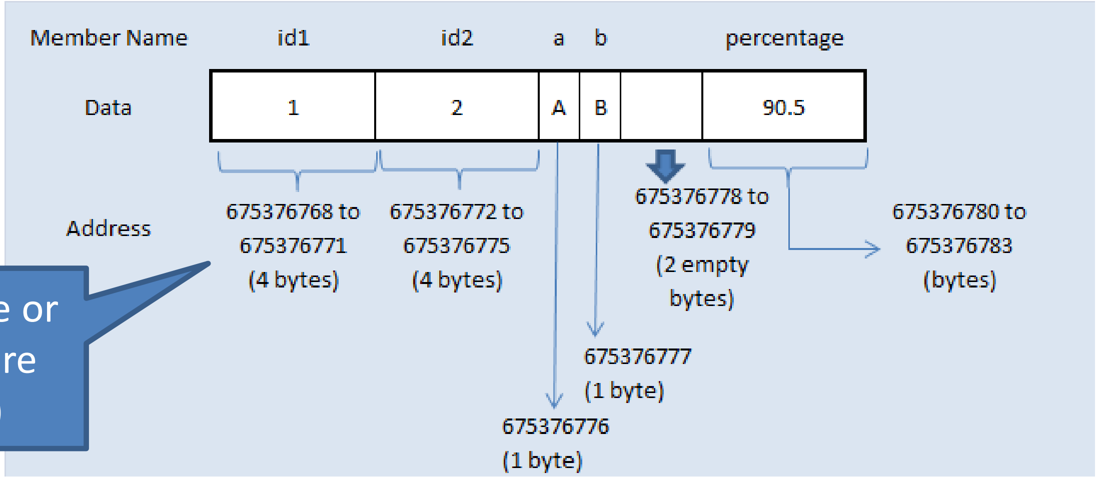
Structures: Storage in memory

```
struct student
{
    int id1;
    int id2;
    char a;
    char b;
    float percentage;
};
```

```
int main()
{
   int i;
   struct student record1 = {1, 2, 'A', 'B', 90.5};

   printf("size of structure in bytes : %d\n", sizeof(record1));

   printf("\nAddress of id1        = %u", &record1.id1 );
   printf("\nAddress of id2        = %u", &record1.id2 );
   printf("\nAddress of a          = %u", &record1.a );
   printf("\nAddress of b          = %u", &record1.b );
   printf("\nAddress of percentage = %u",&record1.percentage);

   return 0;
}
```

```
size of structure in bytes : 16
Address of id1 = 675376768
Address of id2 = 675376772
Address of a = 675376776
Address of b = 675376777
Address of percentage = 675376780
```

Note: fields may be stored starting with the lowest address (as shown below) or highest address



Member Name	id1	id2	a	b		percentage
Data	1	2	A	B		90.5
Address	675376768 to 675376771 (4 bytes)	675376772 to 675376775 (4 bytes)	675376776 (1 byte)	675376777 (1 byte)	675376778 to 675376779 (2 empty bytes)	675376780 to 675376783 (bytes)

Can avoid it by explicitly telling Mr. C not to do padding (using #pragma pack(1))

To align the data in memory, one or more empty bytes (addresses) are inserted ("structure padding")

(Re)defining a Type - typedef

- When using a structure data type, it gets a bit cumbersome to write struct followed by the structure name every time.

- Alternatively, we can use the typedef command to set an alias (or shortcut).

```
struct point {
        int x; int y;
};
typedef struct point Point;
struct rect {
  Point leftbot;
  Point righttop;
};
```

- We can also merge struct definition and typedef:

```
typedef struct point {
        int x; int y;
} Point;
```

More on typedef

- **`typedef`** may be used to rename *any* type
  - – Convenience in naming
  - – Clarifies purpose of the type (typedef char* string;)
  - – Cleaner, more readable code
- Syntax

```
typedef Existing-Type NewName;
```

  - – **Existing type** is a base type or compound type
  - – **NewName** must be an identifier (same rules as variable/function name)

More on typedef

```
typedef char* String;
// String: a new name to char pointer

typedef unsigned int size_t; // Improved
                           //Readability

typedef struct point* PointPtr;

typedef long long int int64;
```

Bit Fields

Sometimes, not all fields in a struct need the same amount of storage even if they are of the same data type

```
// a struct to store date
struct date {
    unsigned int d;
    unsigned int m;
    unsigned int y;
};
```

In the above, d ranges from 1-31, m ranges from 1-12, and y is a 4 digit integer
But the above will use 4 bytes for each of them. Wasteful.

Bit Fields

The idea of bit fields is to specify how many bits we want to use for storing each field. The definition looks like this

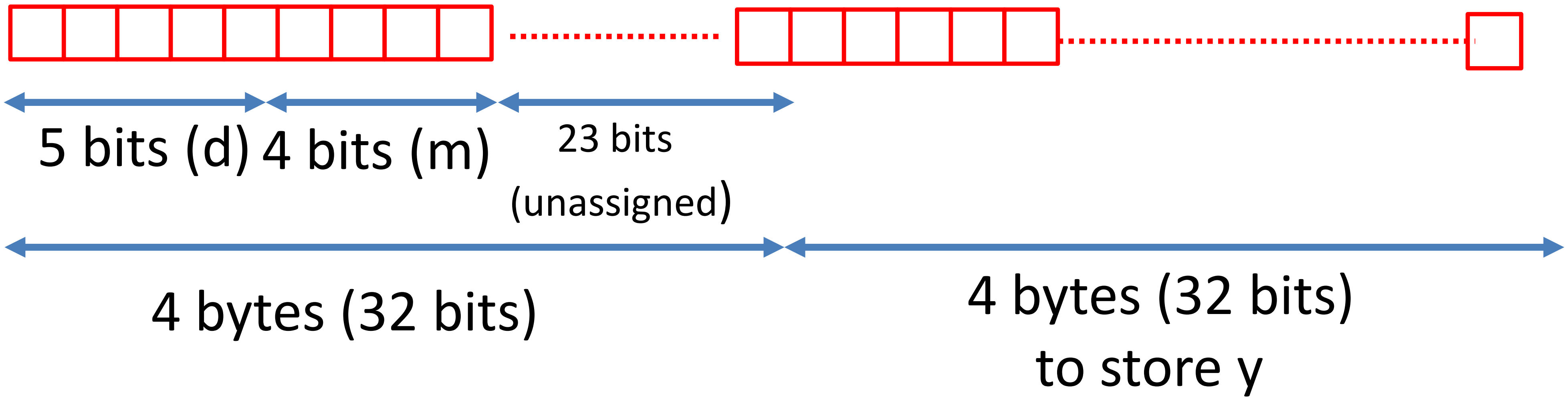
```
// a struct to store date
struct date {
    unsigned int d : 5;      // d will now use only 5 bits
    unsigned int m : 4;     // m will now use only 4 bits
    unsigned int y;           // y will use all 4 bytes (as an unsigned int)
};
```

Still saved 4 bytes ☺

Total storage required will be 8 bytes, not 4 bytes + 9 bits?
4 bytes for y + a total of 4 bytes for d and m (even though d and m together need only 9 bits, one full unsigned int will be allotted to store them)

Bit Fields

```
// a struct to store date
struct date {
    unsigned int d : 5;       // d will now use only 5 bits
    unsigned int m : 4;     // m will now use only 4 bits
    unsigned int y;           // y will use all 4 bytes (as an unsigned int)
};
```



5 bits (d)

4 bits (m)

23 bits (unassigned)

4 bytes (32 bits)

4 bytes (32 bits) to store y

Important note: Can’t get the address of individual fields if using bit fields. Can only get the address (pointer) of the whole structure variable and then access each field using that pointer

Enumerated Type

- Collecting data about bank accounts
  - Need a variable for account type: Checking, Saving, …

- Dealing with the color of a traffic light
  - A variable that can hold only three values: red, yellow, green

- One option is to use numbers (0,1,2,3,…) but numbers not very meaningful

- Enumerated type provides a better way of storing such information

Enumerated Types

- Enumerated type allows us to create our own symbolic name for a list of related things.
  - The key word for an enumerated type is **enum**.

- Here is the C statement to create an enumerated type to represent various “account types”

```
enum account_type {savings, current, fixedDeposit, minor};
```

- In the above, savings means 0, current means 1, fixDeposit means 2, and so on (the first symbolic name maps to 0 by default). Internally, each possible value will be an integer

Example: Enumerated Types

- Account type via Enumerated Types

```
enum account_type { savings, current, fixedDeposit, minor  };

enum account_type a;
a = current;

if (a==savings)
        printf("Savings account\n");

if (a==current)
        printf("Current account\n");
```

Can use typedef here as well to shorten it ☺

Enumerated types provide a symbol to represent one state out of several constant states.

- The default values (0,1,2,…) can be changed, e.g.,

```
enum account_type { savings = 2, current = 1, fixedDeposit = 3, minor = 6  };
```