Structure in C language

Variable and Data Types

int x = 9;

- In programming we need something for holding data, and *variables* is the way to do that.
- A *data type* in a programming language is a set of data with predefined values. Examples of data types are: integer, float, string,…
- There are two kind of data types in programming languages:
  - System-defined data types (also called *Primitive* data types)
  - User-defined data types

System-defined data types

- The data types are defined by languages
- The number of bits are allocated to each data type is fully depend on the language and compiler
- For example: in C language:
  - int: 2 bytes (actual value depends on compiler)
  - float: 4 bytes
  - …

User-defined data types

- The user-defined data types are defined by the user himself.
- In C we can create **structure**. In C++/Java, we can create **class.**
- For example:

struct student{
  - char[20] name;
  - int id;

};

struct student x;

Data structure

- *Data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- General data structure: Array, Linked List, Queue, Stack,…
- Data structure is divided into two types:
  - Linear data structure
  - Non-linear data structure

Type Definition

- Syntax: `typedef` *`type`* *`Name`* `;`
- Name becomes a name you can use for the type
- Examples:

```
typedef struct student Student;
Student x;  /* x is an struct student */
Student* PtrToStudent
```

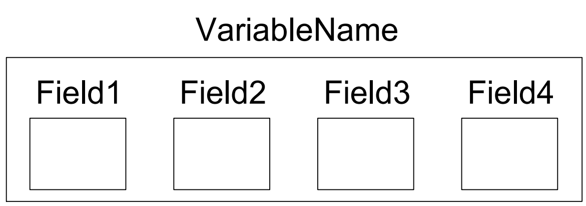
↔

```
typedef struct {
    char[20] name;
    int id;
} Student;
```

```
typedef char *STRING;
STRING sarray[10];
/* sarray is an array of char *'s, equivalent to declaring:
   char *sarray[10]; */
```

Structured Variables

- Group of related values (but unlike array, where the values are not necessarily of the same type)
- Each part of structure is a *field*, with an associated value
- The group is treated as a single unit (a single variable with parts)



Structured Variable

- Declaration

```
struct {
  Type1 FieldName1;
  Type2 FieldName2;
  Type3 FieldName3;
  /* as needed */
} VarName;
```

- Variable consists of parts corresponding to the fields
- Memory set aside corresponding to the total size of the parts
- Each part is an individual variable of the appropriate type

Structure Types

Tag declaration:

```
struct TagName {
  Type1 Field1;
  Type2 Field2;
  Type3 Field3;
  /* any more */
};
```

Variable declaration:

```
struct TagName VarN;
```

Type definition:

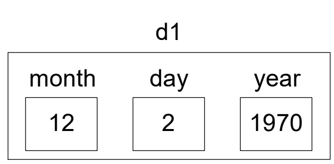
```
typedef struct {
  Type1 Field1;
  Type2 Field2;
  Type3 Field3;
  /* any more */
} TypeName;
```

Variable declaration:

```
TypeName VarN;
```

Field Selection

- Dot (.) form to refer to field of structured var
- Syntax: *VarName*.*FieldName*
- Each field treated as an individual variable of that type



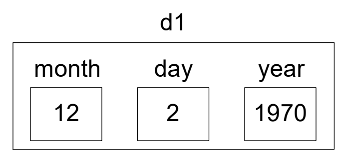
- Example:

```
typedef struct {
  int month, day, year;
} DATE;

void main() {
  DATE d1;

  d1.month = 12;
  d1.day = 2;
  d1.year = 1970;
}
```

Structure Initialization

- Can initialize structured variable by giving value for each field (in the order fields are declared)
- Syntax:
  STYPE Svar = { *FVal1*, *FVal2*, *FVal3*, … };
- Example:

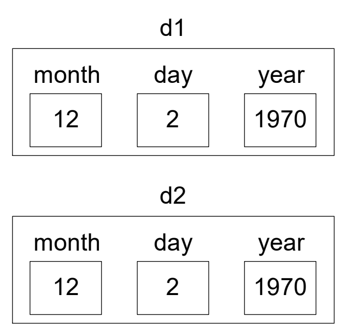
```
typedef struct {
  int month, day, year;
} DATE;

DATE d1 = { 12, 2, 1970 };
```



Structure Assignment

- Can assign value of one structured var to another
  - variables must be of same type (same name)
  - values are copied one at a time from field to corresponding field
- Example:

```
typedef struct {
  int month, day, year;
} DATE;

DATE d1 = { 12, 2, 1970 };
DATE d2;

d2 = d1; /* Assignment */
```



Structure Comparison

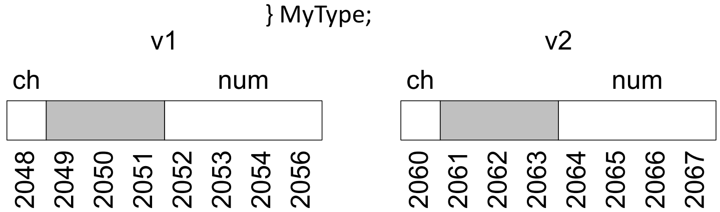
- Should not use == or != to compare structured variables
  - compares byte by byte
  - structured variable may include unused (garbage) bytes that are not equal (even if the rest is equal)
  - unused bytes are referred to as slack bytes
  - to compare two structured variables, should compare each of the fields of the structure one at a time

Slack Bytes

- Many compilers require vars to start on even numbered (or divisible by 4) boundaries, unused bytes called slack bytes
- Example:

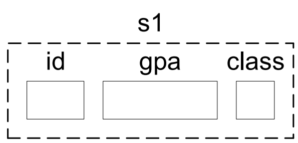
```
typedef struct {
  char ch;
  int num;
} MyType;
```

```
MyType v1;
MyType v2;
```



Structure Example

```
#include <stdio.h>

typedef struct {
  int id;
  float gpa;
  char class;
} Student;

void main() {
  Student s1;
```



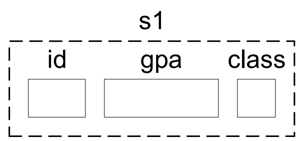

```
  printf(”Enter:\n”);
  printf("  ID#: ");
  scanf("%d",&s1.id);
  printf("  GPA: ");
  scanf("%f",&s1.gpa);
  printf("  Class: ");
  scanf(" %c",&s1.class);

  printf(”S#%d (%c) gpa = 
%.3f\n",s1.id,s1.class,s1
.gpa);
}
```

Passing Structures as Parameters

- A field of a structure may be passed as a parameter (of the type of that field)
- An advantage of structures is that the group of values may be passed as a single structure parameter (rather than passing individual vars)
- Structures can be used as
  - value parameter: fields copied (as in assignment stmt)
  - reference parameter: address of structure passed
  - return value (resulting structure used in statement) -- not all versions of C allow structured return value
  - best to use type-defined structures

Structure as Value Parameter

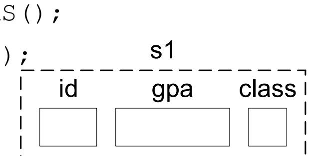
```
#include <stdio.h>

typedef struct {
  int id;
  float gpa;
  char class;
} Student;
```




```
void printS(Student s) {
/* Struc. Param. named s
   created, fields of arg
   (s1) copied to s */
  printf("S#%d (%c) gpa =
%.3f\n",s.id,s.class,s.gpa);
}

void main() {
  Student s1;

  s1 = readS();
  printS(s1);
}
```

Structure as Return Value

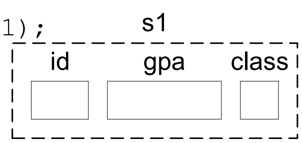
```
typedef struct {
  int id;
  float gpa;
  char class;
} Student;
void main() {
  Student s1;

  s1 = readS();
  printS(s1);
}
```




```
Student readS() {
  Student s; /* local */

  printf(”Enter:\n");
  printf("  ID#: ");
  scanf("%d",&s.id);
  printf("  GPA: ");
  scanf("%f",&s.gpa);
  printf("  Class: ");
  scanf(" %c",&s.class);

  return s; /* local as
               return val */
}
```

Structure as Reference Parameter

```
typedef struct {
  int id;
  float gpa;
  char class;
} Student;
void main() {
  Student s1;

  readS(&s1);
  printS(s1);
}
```



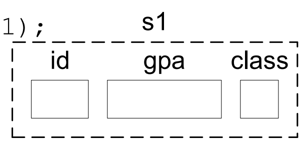

```
void readS(Student *s) {
  printf(”Enter:\n");
  printf("  ID#: ");
  scanf("%d",&((*s).id));
  printf("  GPA: ");
  scanf("%f",&((*s).gpa));
  printf("  Class: ");
  scanf(" %c",&((*s).class));
}
/*
  s - address of structure
  *s - structure at address
  (*s).id - id field of struc
          at address */
```

The Pointer Selection Operator

- Passing a pointer to a structure rather than the entire structure saves time (need not copy structure)
- Therefore, it is often the case that in functions we have structure pointers and wish to refer to a field:
  (**StrucPtr*).*Field*
- C provides an operator to make this more readable (the pointer selection operator)
  StrucPtr->*Field* /* equivalent to (**StrucPtr*).*Field* */
  - StrucPtr must be a pointer to a structure
  - Field must be a name of a field of that type of structure

Pointer Selection Example

```
typedef struct {
  int id;
  float gpa;
  char class;
} Student;
void main() {
  Student s1;

  readS(&s1);
  printS(s1);
}
```



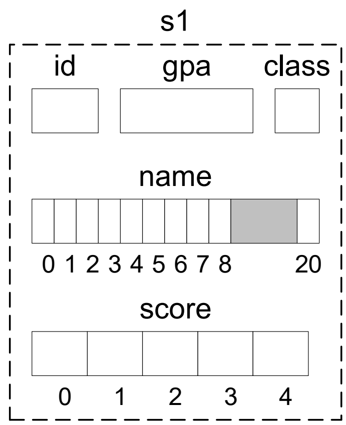

```
void readS(Student *s) {
  printf(”Enter:\n");
  printf("  ID#: ");
  scanf("%d",&(s->id));
  printf("  GPA: ");
  scanf("%f",&(s->gpa));
  printf("  Class: ");
  scanf(" %c",&(s->class));

  printf(“Id is %d”,s->id);
}
```

Derived Types as Fields

- The fields of a structure may be any type, including derived types such as arrays, structures, enumerations, etc.
- An array within a structure is given a field name, to refer to individual elements of the array we give the field name and then the array ref ([*x*])
- A structure within a structure is referred to as a nested structure -- there are a couple of ways to declare such structures

Array Within Structure

```
typedef struct {
  int id;
  float gpa;
  char class;
  char name[20];
  int score[5];
} Student;

Student s1;
/* With large structure,
 more efficient to pass as
 pointer */
```



Array Within Structure (cont)

```
void readS(Student *s) {
  int i;

  printf(”Enter:\n");
  printf("  Name: ");
  scanf("%20s",s->name);
  printf("  ID#: ");
  scanf("%d",&(s->id));
  printf("  GPA: ");
  scanf("%f",&(s->gpa));
  printf("  Class: ");
  scanf(" %c",&(s->class));
  printf("  5 grades: ");
  for (i = 0; i < 5; i++)
    scanf("%d",
          &(s->score[i]));
}
```

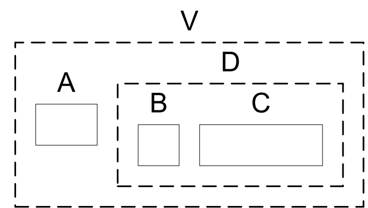
```
void printS(Student *s) {
  int i;

  printf("%s id=#%d (%c)
    gpa = %.3f\n",s->name,
    s->id,s->class,s->gpa);
  for (i = 0; i < 5; i++)
    printf("%d ",s->score[i]);
  printf("\n");
}

void main() {
  Student s1;

  readS(&s1);
  printS(&s1);
}
```

Nested Structure

One mechanism, declare nested structure directly within type definition:

```
typedef struct {
  int A;
  struct {
    char B;
    float C;
  } D; /* struc field */
} MyType;

MyType V;
```

- Fields of V:
  - V.A /* int field */
  - V.D /* structure field */
  - V.D.B /* char field */
  - V.D.C /* float field */

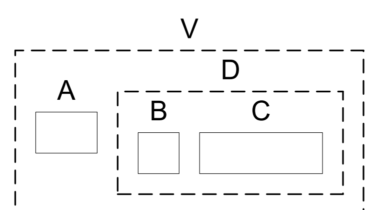


Nested Structure (cont)

Alternate mechanism (preferred):

```
typedef struct {
  char B;
  float C;
} MyDType;
typedef struct {
  int A;
  MyDType D;
} MyType;

MyType V;
```

Fields of V:
- V.A /* int field */
- V.D /* structure field */
- V.D.B /* char field */
- V.D.C /* float field */



Initializing Nested Structures

- To initialize a nested structure we give as the value of the structure a list of values for the substructure:

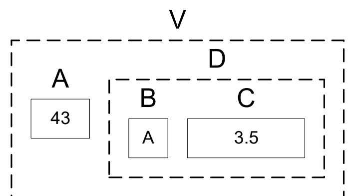
  StrucType V = { *Field1Val*, *Field2Val*, *Field3Val*, … }

  where *FieldXVal* is an item of the form

  { *SubField1Val*, *SubField2Val*, *SubField3Val*, … }

  if Field *X* is a structured field
- Previous example (MyType)

  MyType V = { 43, { 'A', 3.5 } };



Representing Table Data

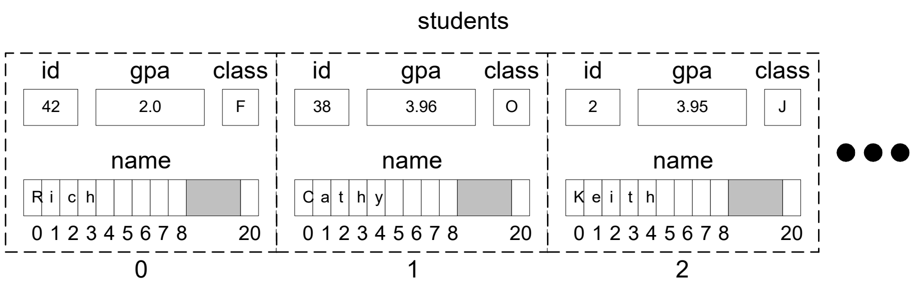
- For many programs it is appropriate to keep track of a *table* of information where we know several things about each entry in the table:

Name	ID	Class	GPA
Rich	42	F	2.00
Cathy	38	O	3.96
Keith	2	J	3.95
Karen	1	S	4.00

- We would like to keep the values for each entry in the table together as one unit

Table: Array of Structures

- One mechanism for representing a table of information is to use an array where each member of the array is a structure representing the info about a line of the table:



Array of Structures

- Define type corresponding to individual element (structured type)

```
typedef struct {
  /* Fields */
} Student;
```

- Declare a named array of that type

```
Student Ss[100];
```

- Often use an integer variable to keep track of how many of the array elements are in use:

```
int numS = 0;
```

Array of Structures Example

```
#include <stdio.h>

#define MAXSTUDENT      100

typedef struct {
  int id;
  float gpa;
  char class;
  char name[20];
} Student;

void main() {
  Student Ss[MAXSTUDENT];
  int numS = 0;
  int option;
```

```
 readSFile(Ss,&numS,"stu.dat");
 do {
  option = select();
  switch (option) {
    case 'I': case 'i':
      insS(Ss,&numS); break;
    case 'R': case 'r':
      remS(Ss,&numS); break;
    case 'P': case 'p':
      prntS(Ss,numS); break;
    case 'S': case 's':
      sort(Ss,numS); break;
    case 'Q': case 'q': break;
  }
  printf("\n");
 } while ((option != 'Q') && (option != 'q'));
 prntSFile(Ss,numS,"stu.dat");
}
```