Derived Types

- C allows a number of derived types:
  - Array - group of similar elements (same type)
  - Pointer - location of a variable
  - Enumerated - set of named values
  - Structure - group of related elements (different types)
  - Union - single type allowing different types of values
- Provides a mechanism (type definition) to give names to these types

Outline

- Type Definition
- Enumerated Type
  - definition, representation
- Structured Type
  - definition, initialization, assignment, comparison
  - passing as parameter (value, ref), return value
  - pointer selection operator
  - arrays within structures
  - structures within structures (nested structures)
  - arrays of structures (tables)
    - reading group, writing group, insert, remove, sort, print
- Union Type
  - definition, use within structures

Type Definition

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

```
typedef int INTEGER;
INTEGER x;  /* x is an integer */

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

Enumerated Types

- New types where the set of possible values is *enumerated* (listed)
- Used to make code more readable
- Declaring an enumerated variable:

  enum { Constant_List } VarName;
- List of constants is a set of identifiers separated by commas
- Example:

  enum { Sun, Mon, Tue, Wed, Thu, Fri, Sat } Day;

Enumerated Type Use

```
enum { Sun, Mon, Tue, Wed, Thu, Fri, Sat } day;

day = Tue;

for (day = Sun; day <= Sat; day++)
  if ((day == Sun) || (day == Sat))
    printf("Weekend\n");
  else
    printf("Weekday\n");

printf("%d\n",Wed); /* Would print 3 */
printf("%s\n",Tue); /* Would print garbage */
```

Enumerated Type Details

- The names in an enumerated type are replaced by integer values by the compiler

  enum { Sun, Mon, Tue, Wed, Thu, Fri, Sat } Day;

  Sun is 0, Mon is 1, Tue is 2, etc.
- You can cause compiler to use different values:
  - give different initial value (C numbers from there):

    enum { Sun = 1, Mon, Tue, Wed, Thu, Fri, Sat } Day;

    Sun is 1, Mon is 2, Tue is 3, etc.
  - give a value for each

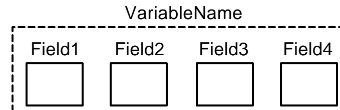
    enum { First = 1, Second = 2, Third = 4, Fourth = 8 } X;
  - important to use different values (C does not check)

Building Enumerated Types

- Declaring type: (outside of function)
  Tag: enum *Tag* { *ConstList* };
  Defined type: typedef enum { *ConstList* } *TypeName*;
  Examples:
    enum DayType1 { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
    typedef enum { Sun, Mon, Tue, Wed, Thu, Fri, Sat } DayType2;
- Variable declarations:
  Tag: enum *Tag VarName* ;
  Defined type: *TypeName VarName*;
  Examples:
    enum DayType1 DayVar1;
    DayType2 DayVar2;

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:

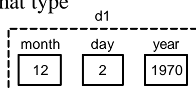
```
typedef struct {
  Type1 Field1;
  Type2 Field2;
  Type3 Field3;
  /* any more */
} TypeName;
```

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



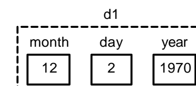
- Example:

```
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:

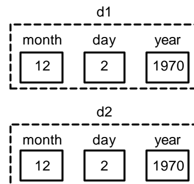
```
typedef struct {
  int month, day, year;
} DATE;

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 */
```



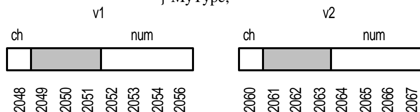
Structure Comparison

- 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

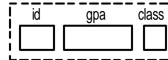
- Many compilers require vars to start on even numbered (or divisible by 4) boundaries, unused bytes called slack bytes
- Example:

```
typedef struct {          MyType v1;
  char ch;                MyType v2;
  int num;
} MyType;
```



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);
}
```

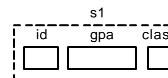


Passing Structures as Parameters

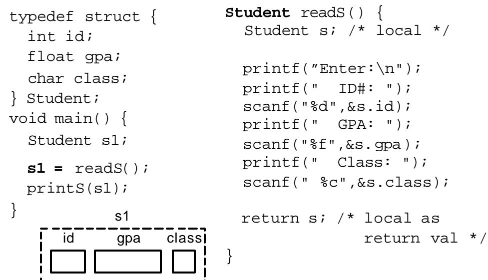
- A field of a structure may be passed as a parameter (of the type of that field)
- An advantage of structures is that the group of values may be passed as a single structure parameter (rather than passing individual vars)
- Structures can be used as
  - value parameter: fields copied (as in assignment 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);
}
```



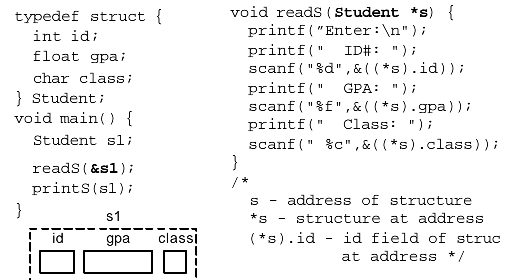
Structure as Return Value



```
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



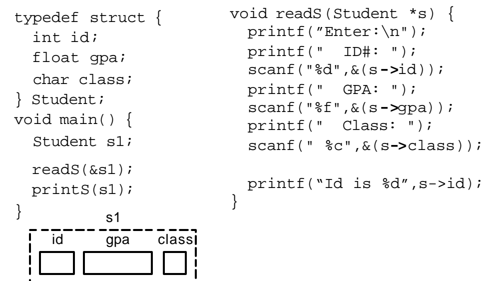
```
typedef struct {
  int id;
  float gpa;
  char class;
} Student;
void main() {
  Student s1;
  readS(&s1);
  printS(s1);
}
```

```
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



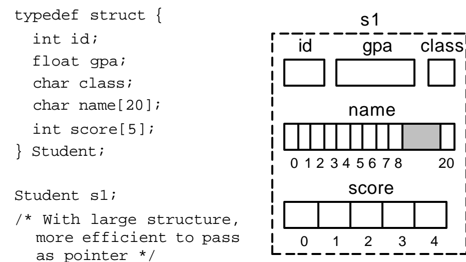
```
typedef struct {
  int id;
  float gpa;
  char class;
} Student;
void main() {
  Student s1;
  readS(&s1);
  printS(s1);
}
```

```
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]));
}
```

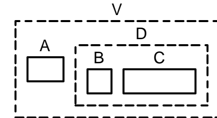
```
void printS(Student *s) {
  int i;

  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 */

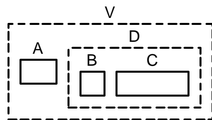


Nested Structure (cont)

Alternate mechanism (preferred):

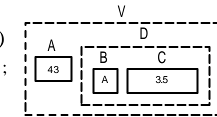
```
typedef struct {
  char B;
  float C;
} MyDType;
typedef struct {
  int A;
  MyDType D;
} MyType;

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

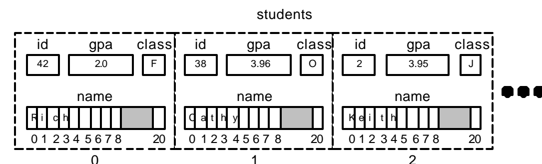
- For many programs it is appropriate to keep track of a *table* of information where we know several things about each entry in the table:

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");
}
```

Menu Function

```
int select() {
  int option, ch;

  printf("Option:\n");
  printf("  I)nsert student\n");
  printf("  R)emove student\n");
  printf("  P)rint students\n");
  printf("  S)ort students\n");
  printf("  Q)uit\n");
  printf("Choice: ");
  scanf(" %c",&option);
  while ((ch = getchar()) != '\n');
  printf("\n");
  return option;
}
```

Initializing Array

- Often the data in the table is entered over multiple uses of the program
  - At end of program save file of info, read file at start
- Write the data for each table item out in format:
  - ints, floats, chars separated by whitespace
  - strings on separate lines (followed by newline)
- Read the data, one entry at a time:
  - use fgets to read strings (terminated by newline)

Saving Table Info to File

```
void prntSFile(Student Ss[], int numS, char *fname) {
  FILE *outs;
  int J;

  if ((outs = fopen(fname,"w")) == NULL) {
    printf("Unable to open file %s\n",fname);
    exit(-1);
  }

  fprintf(outs,"%d\n",numS); /* Save # of students */
  for (J = 0; J < numS; J++) /* Print student J */
    fprintf(outs,"%c %d %f\n%s\n",Ss[J].class,
            Ss[J].id,Ss[J].gpa,Ss[J].name);

  fclose(outs);
}
```

Reading Table from File

```
void readSFile(Student Ss[], int *numS, char *fname) {
  FILE *ins;
  int J;

  if ((ins = fopen(fname,"r")) == NULL) {
    printf("Unable to open file %s\n",fname);
    exit(-1);
  }

  fscanf(ins,"%d",numS);
  for (J = 0; J < *numS; J++) {
    fscanf(ins," %c%d%f",&(Ss[J].class),
           &(Ss[J].id),&(Ss[J].gpa));
    while (getc(ins) != '\n');
    fgets(Ss[J].name,20,ins);
  }

  fclose(ins);
}
```

Insert Item Into Table

- Only if space available
- Unsorted table
  - read the new item in at the location (N) corresponding to the number of items in the table
  - increase N by 1
- Sorted table
  - search for location (J) where new item belongs
  - move items following that location (J..N-1) up one
    - work backward from N-1 to J
  - read item in at location J
  - increase N by 1

Inserting Student

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

  printf("Enter:\n");
  printf("  Name: ");
  fgets(s->name,19,stdin);
  I = strlen(s->name) - 1;
  if (s->name[I] == '\n')
    s->name[I] = '\0';
  printf("  ID#: ");
  scanf("%d",&(s->id));
  printf("  GPA: ");
  scanf("%f",&(s->gpa));
  printf("  Class: ");
  scanf(" %c",&(s->class));
}
```

```
void insS(Student Ss[],
  int *numS) {
  if (*numS < MAXSTUDENT) {
    readS(&(Ss[*numS]));
    *numS += 1;
  }
  else
    printf("Sorry, cannot
   add any more students -
   no room!\n");
}
```

Inserting Student (Sorted)

```
void sinsS(Student Ss[] int *numS) {
  int J,K;
  Student newS;
  if (*numS < MAXSTUDENT) {
    readS(&newS);
    J = 0;
    while ((J < *numS) && (newS.id > Ss[J].id))
      J++;
    for (K = *numS; K > J; K--)
      SS[K] = SS[K-1];
    SS[J] = newS;
    *numS += 1;
  }
  else
    printf("Sorry, no room!\n");
}
```

Removing Item From Table

- Find item to be deleted (at location J)
  - if not found, inform user
- Unsorted table
  - move last item in table (at location N-1) to location J
  - decrease N by 1
- Sorted table
  - move items following location J (J+1..N-1) down one
    - work forward from J to N-2
  - decrease N by 1

Removing Student

```
void remS(Student Ss[], int *numS) {
  int idNum, J;

  printf("ID# of student to remove: ");
  scanf("%d",&idNum);
  J = 0;
  while ((J < *numS) && (idNum != Ss[J].id)) J++;

  if (J < *numS) {
    Ss[J] = Ss[*numS - 1];
    *numS -= 1;
  }
  else
    printf("Can't remove (no student id %d)\n", idNum);
}
```

Removing Student (Sorted)

```
void sremS(Student Ss[], int *numS) {
  int idNum, J, K;

  printf("ID# of student to remove: ");
  scanf("%d",&idNum);
  J = 0;
  while ((J < *numS) && (idNum < Ss[J].id)) J++;

  if ((J < *numS) && (idNum == Ss[J].id)) {
    for (K = J; K < (*numS - 1); K++)
      Ss[K] = Ss[K+1];
    *numS -= 1;
  }
  else
    printf("Can't remove (no student id %d)\n", idNum);
}
```

Printing Students

```
void prntS(Student Ss[], int numS) {
  int J;
  printf("Student              ID# C  GPA\n");
  printf("-------------------------------\n");
  for (J = 0; J < numstudents; J++)
    printf("%-20s %3d %c %5.3f\n",
      Ss[J].name,Ss[J].id,Ss[J].class,Ss[J].gpa);
}
```

Sorting Table

- Pick an appropriate sorting method
  - selection sort is appropriate if the structures are large
  - insert sort if you expect the elements to be nearly sorted
- Determine the field or fields from the structures that will compared to determine which elements are small
  - determine an appropriate test for that field (if string use strcmp, for most other fields use <)

Selection Sorting Students

```
void sort(Student Ss[], int numS) {
  int J, K, smallAt;
  Student temp;

  for (J = 0; J < (numS - 1); J++) {
    smallAt = J;
    for (K = J + 1; K < numS; K++)
      if (Ss[K].id < Ss[smallAt].id)
        smallAt = K;
    temp = Ss[J];
    Ss[J] = Ss[smallAt];
    Ss[smallAt] = temp;
  }
}
```

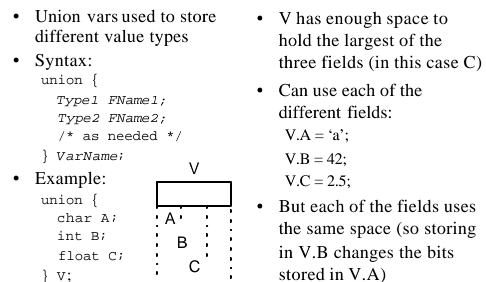
Union Variables

- Union vars used to store different value types
- Syntax:

```
union {
  Type1 FName1;
  Type2 FName2;
  /* as needed */
} VarName;
```

- Example:

```
union {
  char A;
  int B;
  float C;
} V;
```



- V has enough space to hold the largest of the three fields (in this case C)
- Can use each of the different fields:
  V.A = 'a';
  V.B = 42;
  V.C = 2.5;
- But each of the fields uses the same space (so storing in V.B changes the bits stored in V.A)

Union Types

Tag Syntax:

```
union Tag {
  Type1 FName1;
  Type2 FName2;
  /* as needed */
};
union Tag Vname;
```

Example:

```
union Utype1 {
  char A;
  int B;
  float C;
};
union Utype1 V;
```

Defined Type Syntax:

```
typedef union {
  Type1 FName1;
  Type2 FName2;
  /* as needed */
} TName;
TName Vname;
```

Example:

```
typedef union {
  char A;
  int B;
  float C;
} Utype2;
Utype2 V;
```

Unions in Structures

- Union types often used in structures where fields of structure depend item represented
- Example:
  - Seniors - number of credits to graduation
  - Juniors - upper-division GPA
  - Freshmen - character value indicating if student is in-state or out-of-state
- Often use another value in structure to determine which applies

Unions in Structures

```
typedef union {
  int cToGrad;
  float UDGPA;
  char instate;
} SUnionInfo;
typedef enum {Senior,
   Junior, Sophmore,
   Freshman} CEnum;
typedef struct {
  char name[20];
  CEnum whichclass;
  SUnionInfo sinfo;
} Student;
Student s1;
```

```
void prntS(Student s) {
  printf("%s\n",s.name);
  switch (s.whichclass) {
    case Senior:
      printf("  To grad: %d\n",
             s.sinfo.cToGrad);
      break;
    case Junior:
      printf("  UD GPA: %.3f\n",
             s.sinfo.UDGPA);
      break;
    case Freshman:
      if (s.sinfo.instate == 'N')
        printf("Welcome to MN!\n");
      break;
  }
}
```