CSC215 Lecture

User-Defined Data Types

Outline

- Enumerated
  - definition , declaration of variables
- Structures
  - definition , declaration of variables , members access , initialization
  - nested structures , size of structure
  - pointer to structure , array of structure
- Union
  - definition , declaration of variables , size of union
- Bitfield
- typedef keyword

Enumerated Constants

- An enumeration is a user-defined data type that consists of integral constants.
- Synax:

```
enum <type_name> {<id1>[=<val1], <id2>[=<val2>], ..., <idn>[=<valn>]};
```

- Example:

```
enum suit {
    club = 0,
    diamonds = 10,
    hearts = 20,
    spades = 3,
};
```

- Values can be omitted
  - Assigned automatically starting from 0, or from last assigned value, and increasing

```
enum week {sunday,monday,tuesday,wednesday,thursday,friday,saturday };
```

Structure

- A Structure is a collection of related variables:
  - possibly of different types, unlike arrays
  - grouped together under a single name
- A structure type in C is called `struct`
- A Structure holds data that belongs together
- **Examples**:
  - Student record: student id, name, major, gender, ..
  - Bank account: account number, name, balance, ..
  - Date: year, month, day
  - Point: x, y
- `struct` defines a new datatype.

`struct` Definition

- Syntax:

```
struct <struct_tag>{  struct [<struct_tag>]{
        <type> <identifier_list>;
        <type> <identifier_list>;
        <type> <identifier_list>;
        <type> <identifier_list>;
    …
        …
        };
        }<variable_list>;
```

- **Examples**

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

```
struct Student{
   int st_id;
   char fname[100];
   char lname[100];
   int age;
}
```

Declaration of `struct` Variable

- Declaration of a variable of struct type:

```
struct <struct_type> <identifier_list>;
```

- Example1

```
struct studentRec {
  int student_idno;
  char student_name[20];
  int age;
} s1, s2;
```

```
struct studentRec {
  int student_idno;
  char student_name[20];
  int age;
};
struct studentRec s1, s2;
```

- Example2

```
struct s1 { char c; int i; } u ;
struct s2 { char c; int i; } v ;
struct s3 { char c; int i; } x ;
struct s3 y ;
```

  - The types of u , v and x are all different, but the types of x and y are the same.

struct Members

- Individual components of a struct type are called members (or fields)
  - can be of different types (simple, array or struct).
- Complex data structures can be formed by defining arrays of structs.
- Members of a struct type variable are accessed with direct access operator (.)
- Syntax: `<struct-variable>.<member_name>;`
- Example:

```
strcpy(s1.student_name, "Mohamed Ali");
s1.studentid = 43321313;
s1.age = 20;
printf("The student name is %s", s1.student_name);
struct point ptA;
```

`struct` Variable Initialization

- Initialization is done by specifying values of every member.

```
struct point ptA={10,20};
```

- Assignment operator copies every member of the structure
  - be careful with pointers
  - Cannot use == to compare two structure variables
- A variable of a structure type can be also initialized by any the following methods:
- Example:

```
struct date {
  int day, month , year ;
} birth_date = {31 , 12 , 1988};
struct date newyear={1, 1};
```

Nested Structures

- Let's consider the structures:
- We can define the Client inside the BankAccount

```
struct BankAccount{
  char name[21];
  int accNum[20];
  double balance;
  struct{
    char name[21];
    char gender;
    int age;
    char address[21];
  } aHolder;
} b1;
ba.aHolder.age = 35;
```

```
struct Client{
  char name[21];
  char gender;
  int age;
  char address[21];
};
struct BankAccount{
  char name[21];
  int accNum[20];
  double balance;
  struct Client aHolder;
} ba;
ba.aHolder.age = 35;
```

- Client is not visible outside the BankAccount which makes its name optional.

Pointer to Structure

- Created the same way we create a pointer to any simple data type.

```
struct date *cDatePtr,  cDate;
```

- We can make cDatePtr point to cDate by:

```
cDatePtr = &cDate
```

- The pointer variable `cDatePtr` can now be used to access the member variables of `date` using the dot operator as:

```
(*cDatePtr).year
(*cDatePtr).month
(*cDatePtr).day
```

- The parentheses are necessary!
  - the precedence of the dot operator `.` is higher than that of the dereferencing operator `*`

Pointer to Structure Example

```
#include <stdio.h>
#include <math.h>
struct Point{
  int x;
  int y;
};

float distance(struct Point p1, struct Point p2){
  return sqrt((p1.x-p2.x)*(p1.x-p2.x)+
              (p1.y-p2.y)*(p1.y-p2.y));
}

int main(){
  struct Point pp = {3,7};
  struct Point ppp = {-5,2};
  printf("%.2f\n", distance(pp, ppp));
  return 0;
}
```

Pointer to Structure

- Pointers are so commonly used with structures.
- C provides a special operator `->` called the structure pointer or arrow operator or the indirect access operator, for accessing members of a structure variable pointed by a pointer.
- Syntax:
  `<pointer-name>` **`->`** `<member-name>`
- Examples:

```
cDatePtr-> year     ⇔     (*cDatePtr).year
cDatePtr-> month    ⇔     (*cDatePtr).month
cDatePtr-> day      ⇔     (*cDatePtr).day
```

- You cannot edclare a member `x` of type `struct T` inside `struct T`
- But you can declare a member `x` of type `structT*` inside `struct T`

Pointer to Structure Example

```
#include <stdio.h>
#include <math.h>
struct Point{
  int x;
  int y;
};

float distance(struct Point *p1, struct Point *p2){
  return sqrt((p1->x-p2->x)*(p1->x-p2->x)+
              (p1->y-p2->y)*(p1->y-p2->y));
}

int main(){
  struct Point pp = {3,7};
  struct Point ppp = {-5,2};
  printf("%.2f\n", distance(&pp, &ppp));
  return 0;
}
```

Size of structure

- size of a structure is greater than or equal to the sum of the sizes of its members.
- when computer reads/writes from/to memory address
  - it reads/writes a whole word
  - a word size is determined by platform: Ex. 4 bytes in 32-bit systems
  - Self-alignment speeds up memory access to fetch/write typed data
- Alignment

  On modern processors, basic C data types has some storage constraints:
  - Variables of 8-bit length can start at any address
  - Variables of 16-bit length must start on even address
  - Variables of 32-bit length must start on an address that is divisible by 4
  - Variables of 64-bit length must start on an address that is divisible by 8
- Padding:
  - Meaningless bytes were inserted between the end of a structure member and the next

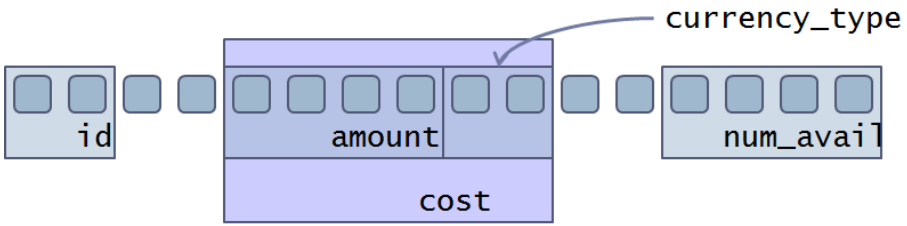
Size of structure

- `struct` alignment is based on its widest scalar member
- Address of `struct` is the same as its first member
- Padding bytes will be added between `struct` members as needed
- Trailing bytes will be added after `struct` variables as needed
- `struct` reordering is not guaranteed to shrink the size of the `struct`
- Compiler directive #pragma can be used to override the alignment:
  - Not a good idea since it slows down the execution
  - Needed when a format has to be followed
- There are too many other details and some are implementation dependent

Size of structure

- Alignment

```
struct COST {
  int amount;
  char currency_type[2]; };
struct PART {
        char id[2];
  struct COST cost;
        int num_avail; };
```

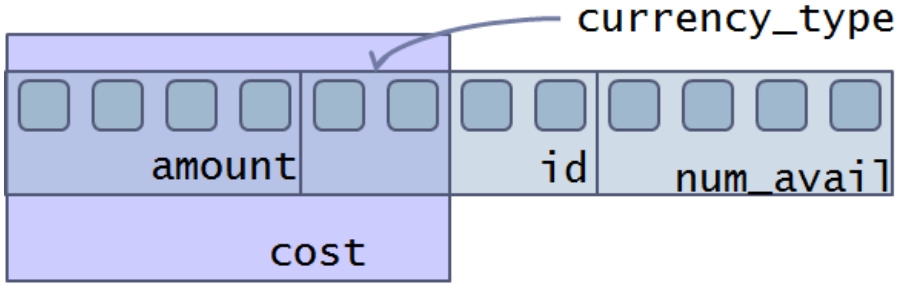


  - Size of struct Part: 16 bytes

- You can try to reduce the structure size by structure reordering or using `#pragma`

- Better:

```
struct COST {
  int amount;
  char currency_type[2]; }
struct PART {
  struct COST cost;
  char id[2];
  int num_avail; }
```



  - Size of struct Part: 12 bytes

Array of Structures

- Can create an array of structures
- Example:

```
struct studentRec {
  int student_idno;
  char *student_name;
  int age;
};
struct studentRec studentRecords[500];
```

  - `studentRecords` is an array containing 500 elements of the type `struct studentRec`.
  - Member variable inside `studentRecords` can be accessed using array subscript and dot operator: `studentRecords[0].student_name = "Mohammad";`

Example

```
#include <stdio.h>

struct Employee {/* declare a global structure type */
  int idNum; double payRate; double hours;
};
double calcNet(struct Employee *); /* function prototype */
int main() {
  struct Employee emp = {6787, 8.93, 40.5};
  double netPay;
  netPay = calcNet(&emp); /* pass an address*/
  printf("The net pay for employee %d is $%6.2f\n", emp.idNum, netPay);
  return 0;
}
/* pt is a pointer to a structure of Employee type */
double calcNet(struct Employee *pt) {
  return(pt->payRate * pt->hours);
}
```

Union

- A variable that may hold objects of different types/sizes in same memory location

```
union data{
  int idata;        d1.idata = 10;
  float fdata;      d2.fdata = 3.14F ;
  char* sdata;      d3.sdata = "hello world" ;
} d1, d2, d3;
```

- Size of union variable is equal to size of its largest element.
- Compiler does not test if the data is being read in the correct format.

```
union data d; d.idata=10; float f=d.fdata; /* will give junk */
```

- A common solution is to maintain a separate variable.

```
enum dtype {INT,FLOAT,CHAR};
struct variant {
  union data d;
  enum dtype t;
};
```

BitField

- A set of adjacent bits within a single ’word’.

  Example:

```
struct flag{
  unsigned int is_color:1;
  unsigned int has_sound:1;
  unsigned int is_ntsc:1;
} ;
```

- Number after the colons specifies the width in bits.
- Each variables should be declared as unsigned int Bit fields
- Portability is an issue

typedef Keyword

- Gives a type a new name

```
typedef unsigned char BYTE;
BYTE  b1, b2;
```

- Can be used to give a name to user defined data types as well

```
struct Books {
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
};
typedef struct Books Book;
Book b1, b2;
```

```
typedef struct {
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
} Book;

Book b1, b2;
```

Example1: Polygon (polygon.h)

```
#ifndef POLYGON
#define POLYGON
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
typedef struct {
  int x;
  int y;
} Point;
typedef struct{
  Point* points;
  int count;
} Polygon;

float distance(Point*, Point*);
Polygon* getPG();
int isParallelogram(Polygon*);

#endif
```

Example1: Polygon (polygon.c)

```
#include "polygon.h"

Polygon* getPG(){
  Polygon* pg;
  Point* p;
  int i=0;
  pg = (Polygon*)calloc(1, sizeof(Polygon));
  printf("Enter number of points:");
  scanf("%d", &(pg->count));
  p=pg->points=(Point*)calloc(pg->count, sizeof(Point));
  if (!p) return NULL;
  while (p < (pg->points)+(pg->count)){
    printf("Enter x for point p%d:", i+1);
    scanf("%d", &(p->x));
    printf("Enter y for point p%d:", i+++1);
    scanf("%d", &(p->y));
    p++;
  }
  return pg;
}
```

Example1: Polygon (polygon.c)

```
float distance(Point* p1, Point* p2){
  return sqrt((p1->x-p2->x)*(p1->x-p2->x) + (p1->y-p2->y)*(p1->y-p2->y));
}

int isParallelogram(Polygon* pg){
  if (pg->count == 4)
    if (distance(pg->points, pg->points+1) == distance(pg->points+2, pg->points+3) &&
        distance(pg->points+1, pg->points+2) == distance(pg->points+3, pg->points))
      return 1;                          /* not a good idea, why? */
  return 0;
}
```

Example1: Polygon (pgtest.c)

```
#include "polygon.h"

int main(){
  Polygon *pg1, *pg2;
  pg1 = getPG();
  printf("This polygon is %sa parallelogram.", isParallelogram(pg1)?"":"not ");
  pg2 = getPG();
  printf("This polygon is %sa parallelogram.", isParallelogram(pg2)?"":"not ");

  return 0;
}
```

data.txt

```
4        5 -3          6 5           1 4           -4 0
4        -3 5          5 6           4 1           -4 0
```

Example2: Matrix - revisited

```
#if !defined MAT
#define MAT

typedef struct{
  int** data;
  int rows;
  int cols;
} Matrix;

Matrix get_matrix(int, int);
void free_matrix(Matrix);     /* OR */  void free_matrix(Matrix*);
void fill_matrix(Matrix);
void print_matrix(Matrix);
Matrix transpose(Matrix);

#endif
```