C PROGRAMMING
Lecture 4

1st semester 2021-2022

Structures

- Structure:
  - Collection of one or more variables
  - a tool for grouping heterogeneous elements together (different types)
- Array: a tool for grouping homogeneous elements together
- Help organize complicated data, permits group of related variables to be handled as a unit
- Example: storing calendar dates (day, month, year), students (name, address, telephone)

Structures

- A struct declaration defines a type:

```
struct [label]{
        type member1;
        …
        type membern;
};
```

- Keyword struct introduces a structure declaration (a list of declarations enclosed in {})
- Variables declared in a structure are called members

Structures

- A structure member and a regular variable can have the same name without conflict
- Struct declaration defines a type; } can be followed by variable names:

  `struct { … } var1, var2, … , varn;`
- If not followed by variable names, it reserves no storage
- If tagged (given a label), tag can be used for variables as structure type instances
- A member of a particular structure is used in an expression by

  `structure_name.member`

Structures

- A structure can be initialized by following it with a list of constant expressions for the members:

```
struct point{
                int p1;
                int p2;
            } pt;
pt = (struct point) { 100, 200 };
//struct point pt = {100, 200 };
```

or

```
pt = (struct point) { .p1 = 100, .p2 = 200 };
```

- By specifying values for each member:

```
pt.p1=100;
pt.p2=200;
```

Structures and typedef

- With typedef can define new datatypes
- Used to shorten declaration of structure variables
- Avoids using struct keyword for every variable of that structure type:

```
typedef struct pct{
    int x;
    int y;
  }  point;
  point p1;
  p1.x = 0;
  p1.y = 0;
  point p2 = {.x=100, .y=200 };
```

Operations on structures

- Possible operations on a structure :
  - initialize by a list of constant member values (expressions)
  - copy or assign to it as a unit
    - this includes passing arguments to functions and returning values from functions as well.
  - use its address (with &)
  - access its members.
  - structures can’t be compared as units !

Example

```
#include <stdio.h>
int main (void)
{
    struct date
    {
        int month;
        int day;
        int year;
    };
    struct date today, tomorrow;
    const int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
    31, 31, 30, 31, 30, 31 };

    printf ("Enter today's date (mm dd yyyy): ");
    scanf ("%i%i%i", &today.month, &today.day, &today.year);
```

```
if ( today.day != daysPerMonth[today.month - 1] ) {
    tomorrow.day = today.day + 1;
    tomorrow.month = today.month;
    tomorrow.year = today.year;
}
else if ( today.month == 12 ) {
   tomorrow.day = 1;
    tomorrow.month = 1;
    tomorrow.year = today.year + 1;
}
else {
    tomorrow.day = 1;
    tomorrow.month = today.month + 1;
    tomorrow.year = today.year;
}
printf ("Tomorrow's date is %i/%i/%i.\n", tomorrow.month,
                              tomorrow.day, tomorrow.year );
return 0;
}
```

Structures Containing Complex DataTypes

- Structures can contain any data type
  - Can contain arrays
  - Can contain other structures
- Also called complex structures

```
typedef struct {
    int a;
    double arr[3];
  }  some_struct;
 some_struct s1;
 s1.a = 10;
 int i;
 for (i=0; i<3; i++) {
   s1.arr[i] = i;
}
```

Structures Containing Complex DataTypes

```
struct address{
        unsigned int house_number;
        char *street_name;
        int zip_code;
        char *country;
      };
struct customer{
        char *name;
        struct address billing;
        struct address shipping;
      };
struct customer c1;
c1.name="John Spencer";
c1.billing.street_name="Second Avenue";
c1.shipping.street_name=c1.billing.street_name;
```

Arrays of Structures

- Declaring an array of structures is like declaring any other kind of array.
- Each element of the array is a structure of type struct. Thus, array[0] is one structure, array[1] is a second structure, and so on.
- To identify members of an array of structures, the rules used for individual structures apply: structure name followed by the dot operator and then with the member name:

  array[0].member_name

Array of Structures

- Can declare an array of structs:

  Point points[10];
- Each array element is a struct.
- To access member of a particular element:

  points[4].x = 100;

- Because the [] and . operators are at the same precedence and associate left-to-right, this is equivalent to:

  (points[4]).x = 100;

Pointers to Structures

- You can have pointers to structures.
- Just as pointers to arrays are easier to manipulate than the arrays themselves, pointers to structures are in general easier to manipulate than structures themselves.
- In some older implementations, a structure can't be passed as an argument to a function, but a pointer to a structure can.

Pointers to Structures

```
struct student *s1;
```

- The syntax is the same as for the other pointer declarations: First is the keyword struct, then the structure label and then an asterisk (*) followed by the pointer name.
- Declaration does not create a new structure, but the pointer is made to point to existing structure of that type.
- Unlike the case for arrays, the name of a structure is not the address of the structure; The & operator is needed.

Pointers to Structures

- In order to access the value of a structure member, one can use -> operator.
- A structure pointer followed by the -> operator works the same way as a structure name followed by the . (dot) operator.
- It is important to note that pointer_name is a pointer, but pointer_name->member is a member of the pointed-to structure.
- Another method to access a member value is to use (*) dereference operator:

struct_name.member_name == (*ptr_name).member_name

Pointers to Structures

- We can declare and create a pointer to a struct:

```
Point *pointPtr; pointPtr = &points[4];
```

- To access a member of the struct addressed by pointPtr:

```
(*pointPtr).x = 100;
```

- Because the . operator has higher precedence than *,this is NOT the same as:

```
*pointPtr.x = 100;
```

- C provides special syntax for accessing a struct memberthrough a pointer:

```
pointPtr->x = 100;
```

Structures and Functions

- For passing parameters, possible approaches:
  - Pass members
  - Pass structure
  - Pass a pointer to a structure
- For returning:
  - Return a member
  - Return a structure
  - Return a pointer

Structures and Functions

- Individual fields can be passed to functions in usual way; if the member is of basic type, the value is passed, if it is an array, the address is passed
- The entire structure is passed by value
- Alternative, pass a pointer

Structures and Functions

```
struct point{
    int p1;
    int p2;
};
struct point create(int x, int y){
    struct point p;
    p.p1=x;
    p.p2=y;
    return p;
}
```

Structures and Functions

```
struct point{
    int x;
    int y;
};
void display(struct point p){
  printf("The x coordinate for the point: %d\n",p.x);
  printf("The y coordinate for the point: %d\n",p.y);
}
int main(){
    struct point pct;
    printf("Enter x value: ");
    scanf("%d",&pct.x);
    printf("Enter y value: ");
    scanf("%d",&pct.y);
    display(pct);
    return 0;
}
```

Structures and Functions

- Unlike an array, a struct is always passed by value into a function.
- The struct members are copied to the function and changes inside the function are not reflected outside the function.
- To see the changes outside the function, solution is to pass a pointer to a struct.

```
int distance(Point *pctA, Point *pctB){
if (pctA->x == pctB->x && pctA->y == pctB->y)
{
   return 0;
}
else
   ...
}
```

Constant Pointers

- constant pointer: when the address it is pointing to can't be changed
- a constant pointer, if already pointing to an address, can't point to a new address
- Declaration:

```
<pointer_type> *const <pointer_name>
```

Constant Pointers

```
#include<stdio.h>
int main()
{
    int nr1 = 0;
    int nr2 = 1;
    int *const ptr = &nr1; //constant ptr
    ptr = &nr2; //Illegal assignement!!!!
    return 0;
}
```

Pointers to Constants

- type of pointer that can't change the value at the address pointed by it.
- Declaration:

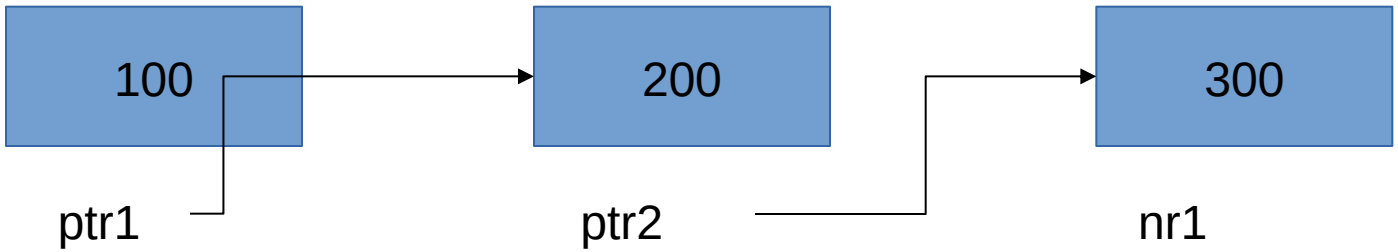
```
const <pointer_type> *<pointer_name>;
```

Pointers to Constants

```
#include<stdio.h>
int main()
{
    int nr1 = 0;
    const int *ptr = &nr1; //pointer
to
                           //constant
    *ptr = 2; // Illegal
assignement!!!
  //Cannot change the value at address
  // pointed by 'ptr'.
    return 0;
}
```

Pointers to Pointers

- A pointer is a variable that can store the address of another variable. The other variable can also be a pointer; so it means that a pointer can point to another pointer.
- Let's suppose a pointer ‘ptr1’ points to another pointer ‘ptr2’ that points to an int 'nr1'. In memory, the three variables can be imagined as:



Pointers to Pointers

```
#include<stdio.h>
int main()
{
    int **ptr1 = NULL;
    int *ptr2 = NULL;
    int nr1 = 0;
    ptr2 = &nr1;
    ptr1 = &ptr2;
    printf("\n nr1 = [%d]\n",nr1);
    printf("\n *ptr2 = [%d]\n",*ptr2);
    printf("\n **ptr1 = [%d]\n",**ptr1);
    return 0;
}
```

Pointer Arrays

- An array of pointers can be declared as :

```
<type> *<name>[<number_of_elements];
```

- For example :

```
int *ptr[5];
```

declares an array of five pointers to integer numbers.

Pointer Arrays

```
#include<stdio.h>
int main()
{
int nr1=0, nr2=2;
    int *arr[2];    arr[0] = &nr1;    arr[1] = &nr2;
   printf("\n nr1 = [%d] \n",nr1);
   printf("\n nr2 = [%d] \n",nr2);
   printf("\n arr[0] = [%p] \n",arr[0]);
   printf("\n arr[1] = [%p] \n",arr[1]);
   printf("\n val of *arr[0] = [%d] \n",*arr[0]);
   printf("\n val of *arr[1] = [%d] \n",*arr[1]);
   return 0;
}
```

Function Pointers

- Just like pointer to characters, integers etc, we can have pointers to functions
- Declaration:

```
<return_type> (*<pointer_name>) (type_of_
  arguments)
```

- The same as for arrays, the function name is the address of the function
- A pointer to a function can be manipulated in the same way as other pointers; most important, it can be passed to a function

Function Pointers

- When to use function pointers?
  - Passing function as parameters for functions
  - Callback functions
- Declaration example:

```
int (*func)(int)
```

Beware that is different from

```
int *funct(int)
```

Function Pointers

- Example:

```
void makesomething(int nr1, int nr2, int (*func)(int))
{
    int i;
    for(i=nr1;i<=nr2;i++)
       printf("%d %d\n", i, (*func)(i));
}
```

- How to interpret the printf from example:

func is a pointer to a function; *func is the function

i is the argument of the function, passed as arg between ()

the value returned is int, matched by %d

() around *f are used because of the operators precedence; *func(i) equivalent for *(func(i)) – has no meaning

Function Pointers

```
#include <stdio.h>
void performtask(int nr1, int nr2, float
(*func)(int))
{
   int i;
   for(i=nr1;i<=nr2;i++)
      printf("%d %f\n", i, (*func)(i));
}
float reciprocal(int nr)
{
   return(1.0/nr);
}
```

Function Pointers

```
float square(int nr)
{
   return(nr*nr);
}
int main()
{
   performtask(1,5,reciprocal);  // can be called
also square, depending on some condition…
  return 0;
}
```