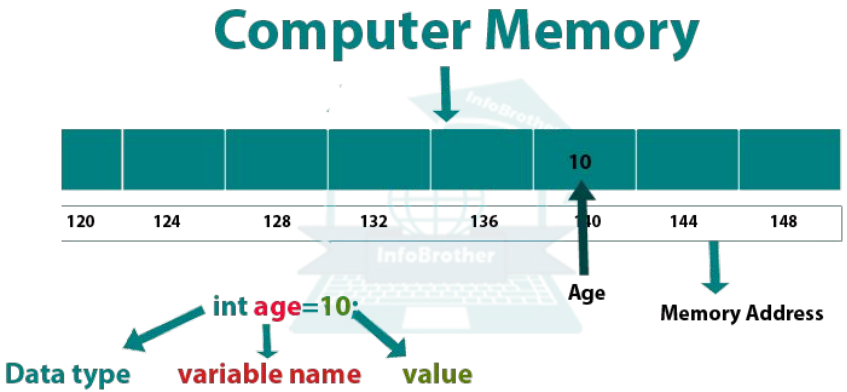
CENG 322
LAB 3

Pointers and Structures

Addresses and Pointers

Variable Addresses in C

- Every variable has a **memory location** and every memory location has its address.
- Every variable has an address.
- Every variable has a value.
- The **real variable name** is its address.
- When your program uses a variable the compiler inserts machine code that calculates the **address of the variable**.



Variable Addresses in C

- In C, we can use **ampersand (&)** operator to obtain an address in the memory.
- If you have a variable **var** in your program, **&var** will give you its address in the memory.
- We have used address numerous times while using the **scanf()** function.

```c
scanf("%d", &var);
```

- We can use **%p format specifier** to print memory address. This is used to print the pointer type data.

```c
int var = 10;
printf("Var: %d\n", var);
printf("Address of var: %p", &var);
```

Variable Addresses in C

- 0x is a message to the compiler: “**this number is in hexadecimal format!**”

```
int a = 42; // decimal number
int b = ??; // hexadecimal number

printf("%d \n", a);
printf("%d \n", b);
printf("%x \n", a);
printf("%x \n", b);
```

Variable Addresses in C

- 0x is a message to the compiler: "**this number is in hexadecimal format!**"

```
int a = 42;   // decimal number
int b = 0x2A; // hexadecimal number

printf("%d \n", a);
printf("%d \n", b);
printf("%x \n", a);
printf("%x \n", b);
```

- Is this true: **a==b** ?

Variable Addresses in C

- 0x is a message to the compiler: “**this number is in hexadecimal format!**”

```
int a = 42;    // decimal number
int b = 0x2A; // hexadecimal number

printf("%d \n", a);
printf("%d \n", b);
printf("%x \n", a);
printf("%x \n", b);
```

- Is this true: **a==b** ?

  The lesson is: don’t confuse **notations** with values or types.

  a and **b** have the exactly same value (Values are **in binary format** in memory).

Variable Addresses in C

- Most computers are "**byte addressable**". It means that **each byte of memory** has a distinct address.
- Integers are stored using **4 bytes** of memory in C (32 bit).

Variable name			**num**					
Address (hex)	...	7ffe129f0703	7ffe129f0704	7ffe129f0705	7ffe129f0706	7ffe129f0707	7ffe129f0708	...
Value (hex)	...	5B (random)	00	00	00	2A	CA (random)	...

`int num;`

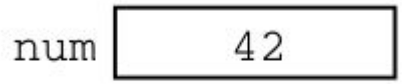
`sizeof(num)` → 4 (or 0x4)

`num` → 42; (or 0x2A)

`&num` → 0x7ffe129f0704 – address of first byte

Variable Addresses in C

- **num** is a simple variable:

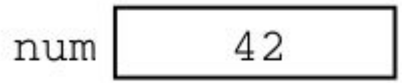



```c
int num;
num = 42;

printf("%d \n", num); // 42
printf("%p \n", &num); // Something like 0x7ffe129f0704
```

- Value can be arbitrarily changed: **num = 13**;
  What about **&num = 0x500;** ?

Variable Addresses in C

- **num** is a simple variable:




```c
int num;
num = 42;

printf("%d \n", num); // 42
printf("%p \n", &num); // Something like 0x7ffe129f0704
```

- Value can be arbitrarily changed: **num = 13**;

  Address **cannot be** arbitrarily changed: **&num = 0x500;** // ERROR

  The type of **&num** is **int* (Pointer to int)**, not **int**.

Pointers

- A pointer is a variable whose value is the **address of another variable**.
- We can access the variable **using the pointer**, instead of using the **variable's name**.
- An **indirection** in C is denoted by the operand * followed by the name of a **pointer variable**. Its meaning is "**access the content the pointer points to**" (indirect addressing)
- Like any variable or constant, you **must declare** a pointer **before** you **use it** to store any variable address.
- The general form of a pointer variable declaration is:

```
type* var_name;
int* p;
```

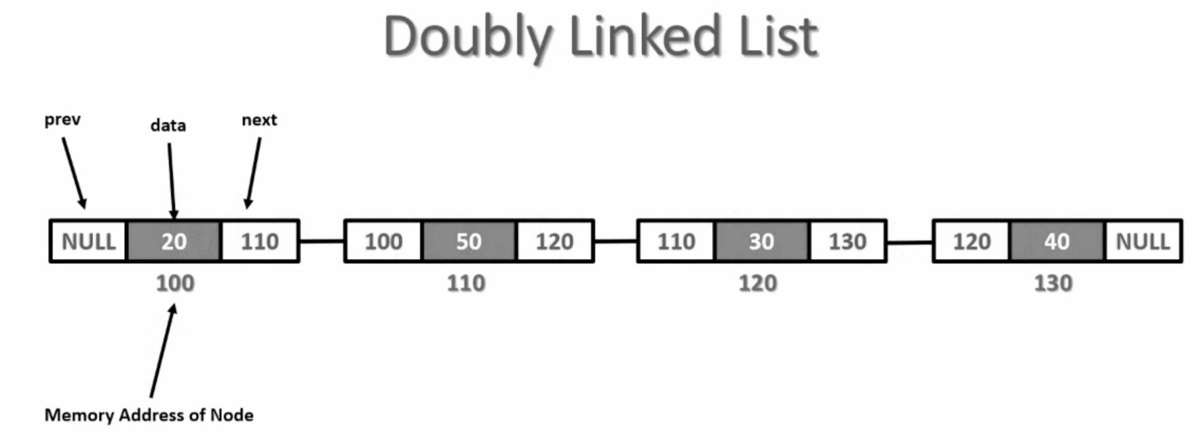
- You can also declare pointers in these ways:

```
int *p;
int * p;
```

Pointers

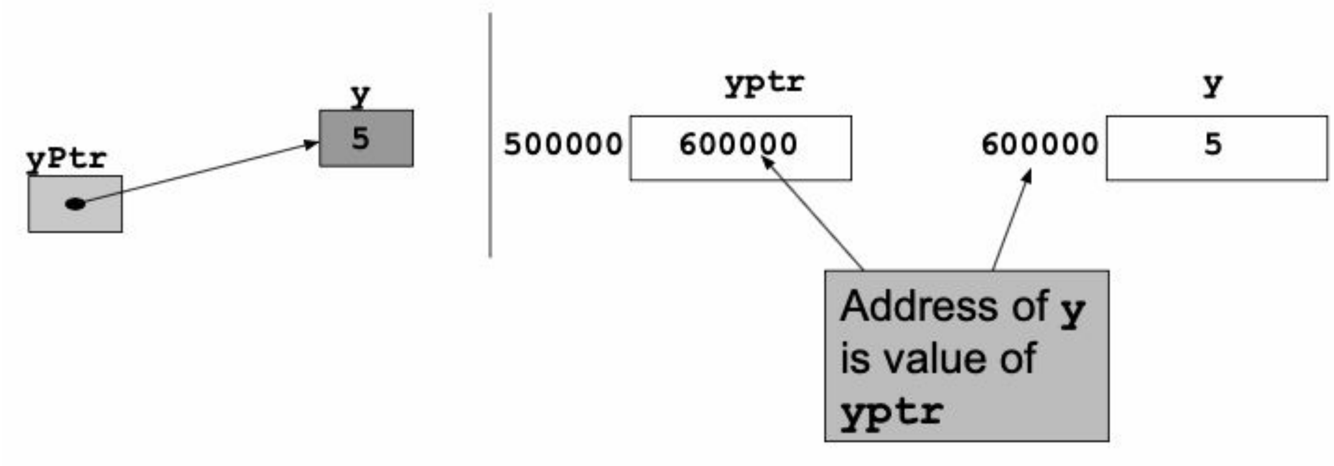
Why do we need pointers?

- They allow different sections of code to share information easily / **more efficiently**.
- Instead of passing **all elements of an array to a function**, we pass **address** of the array to the function.
- They enable **complex data structures** (linked lists, binary trees, etc.)



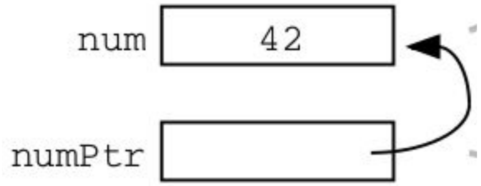
Pointers

- Pointee



Pointers

- Calculating the address of a variable and storing it in a pointer:

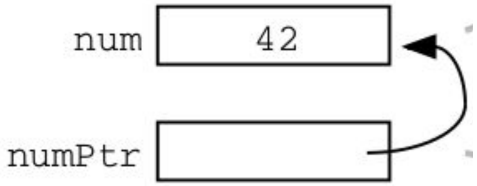



```
int num;
int* numPtr;
num = 42;
numPtr = &num;
printf "Address of the variable: %p\n", &num); // like 0x7ffc1647c59c
printf("Address stored in pointer: %p\n", numPtr);// like 0x7ffc1647c59c
printf("Value of pointer: %d\n", *numPtr);
```

- Value of numPtr is the address of num.
- How can we change value of num by pointer (numPtr) ?

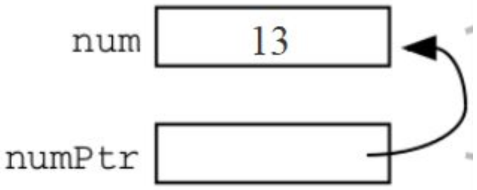
Pointers

- Calculating the address of a variable and storing it in a pointer:




```
int num;
int* numPtr;
num = 42;
numPtr = &num;
printf "Address of the variable: %p\n", &num); // like 0x7ffc1647c59c
printf("Address stored in pointer: %p\n", numPtr);// like 0x7ffc1647c59c
printf("Value of pointer: %d\n", *numPtr);
```

- Value of numPtr is the address of num.
- How can we change value of num by pointer (numPtr) ?



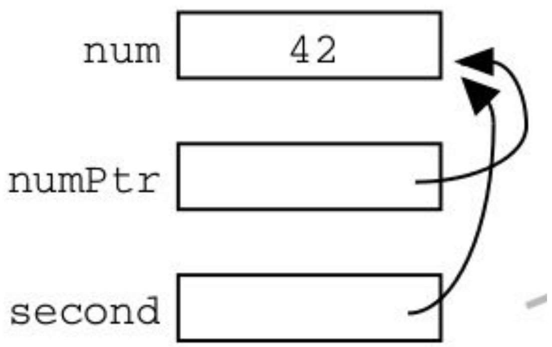

```
*numPtr = 13; // Assign new value in *pointer
           //or just write num = 13
```

Note: This is called dereferencing the pointer.

Pointers

- Assigning one pointer to another pointer:

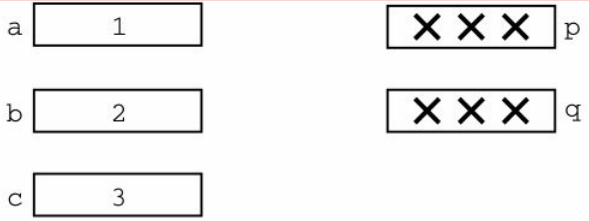
```
int* second;

second = numPtr; // or second = &num;
numPtr = &num;

printf("Value of pointer: %d\n"  *numPtr);
printf("Value of second pointer: %d\n", *second);
```



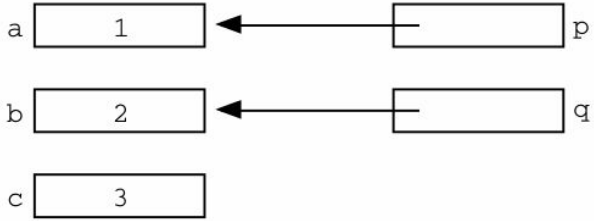
- **second** has the same value as **numPtr**.
- This value is an address. (They both refer to the **same point**!)

Pointers

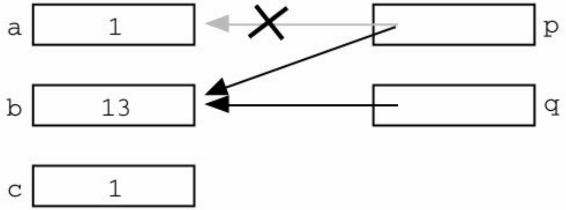
```
// allocate three integers and two pointers
int a = 1;
int b = 2;
int c = 3;
int* p;
int* q;
```




```
p = &a;      // set p to refer to a
q = &b;      // set q to refer to b
```




```
c = *p;  // retrieve p's pointee value (1) and put it in c
p = q;   // change p to share with q (p's pointee is now b)
*p = 13; // dereference p to set its pointee (b) to 13 (*q is now 13)
```



Pointers

- Pointers can also be **compared** using ==, !=, <, >, <=, and >=
- Two pointers are "**equal**" if they point to the same variable (i.e., the pointers have the same value!)
- A pointer p is "**less than**" some other pointer q if the **address currently stored** in p is smaller than the address currently stored in q.

```
int* ptr1;
int* ptr2;
int num1 = 5;
int num2 = 5;

ptr1 = &num1;
ptr2 = &num2;
```

- Are the pointers equal (ptr1 == ptr2) ?

Pointers

- x* is a pointer to x (where x is a data type). e.g. int*, float*, char*, ...
- From **this definition**, we can also **infer**: x** is a pointer to a pointer to x. (e.g. int** refers to an int* which refers to an int).
- Let’s visualize it! - https://goo.gl/4w5rru

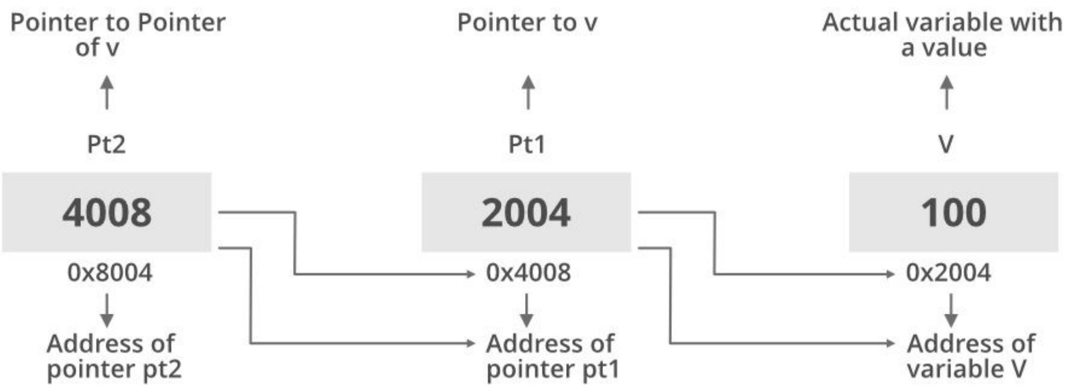
```
int v;
int* pt1;
int** pt2;
v = 100;
pt1 = &v;
pt2 = &pt1;
```

These are all equal to 100: v, *pt1, *(*pt2)

These are all equal to the address of v: &v, pt1, *pt2

These are all equal to the address of pt1: &pt1, pt2

This is the address of ptr2: &pt2



Pointers

```c
int* p, q;
```

is the same as

```c
int* p;
int q;
```

and different than

```c
int* p;
int* q;
```

You may want to write it as

`int *p, q;` (with a space after int).

Advice: do not declare multiple pointers in single line.

```c
int *p = &q;
```

is the same as

```c
int *p;
p = &q;
```

and different than

```c
int *p;
*p = &q;
```

You may want to write it as

`int* p = &q;` (with a space before p).

Advice: do not declare a pointer and assign its value in single line.

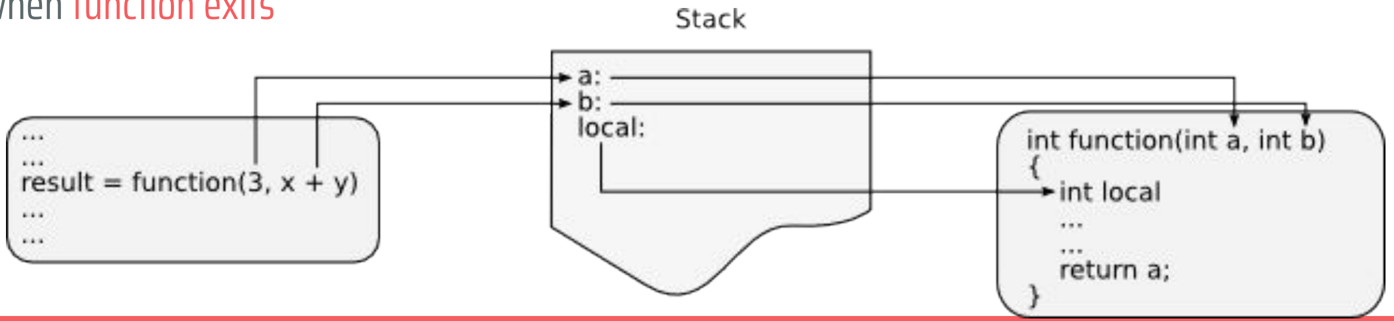
Local Memory

- It is a special area of computer's memory which stores **temporary variables** created by a function.
- In the local memory, variables are declared, stored and initialized **during runtime**.
- Local variables:
    - allocated (given an area of memory)
        - when function calls
    - deallocated (reclaim the memory from the variable)
        - when function exits

```
// Local storage example
int Square(int num) {
    int result;

    result = num * num;

    return result;
}
```



Local Memory

```
void X() {
    int a = 1;
    int b = 2;
    // T1

    Y(a);
    // T3
    Y(b);

    // T5

}
```

```
void Y(int p) {
    int q;
    q = p + 2;
    // T2 (first time through), T4 (second time through)
}
```

Local Memory

```
void X() {
    int a = 1;
    int b = 2;
    // T1

    Y(a);
    // T3
    Y(b);

    // T5

}
```

```
void Y(int p) {
    int q;
    q = p + 2;
    // T2 (first time through), T4 (second time through)
}
```

T1 - X()'s locals have been allocated and given values..	**T2** - Y() is called with p=1, and its locals are allocated. X()'s locals continue to be allocated.	**T3** - Y() exits and its locals are deallocated. We are left only with X()'s locals.	**T4** - Y() is called again with p=2, and its locals are allocated a second time.	**T5** - Y() exits and its locals are deallocated. X()'s locals will be deallocated when it exits.
	Y() p 1, q 3		Y() p 2, q 4	
X() a 1, b 2	X() a 1, b 2	X() a 1, b 2	X() a 1, b 2	X() a 1, b 2

Heap Memory

- The heap is a memory used by programming languages to store **global variables**.
- It supports **dynamic memory allocation**.
- The programmer explicitly requests the **allocation of a memory "block"** of a particular size, and the **block continues to be allocated** until the programmer explicitly requests that it be deallocated.

- Call the heap allocation function
  - `void* malloc(unsigned size);`
- Explicit deallocation request
  - `void free(void* pointer);`

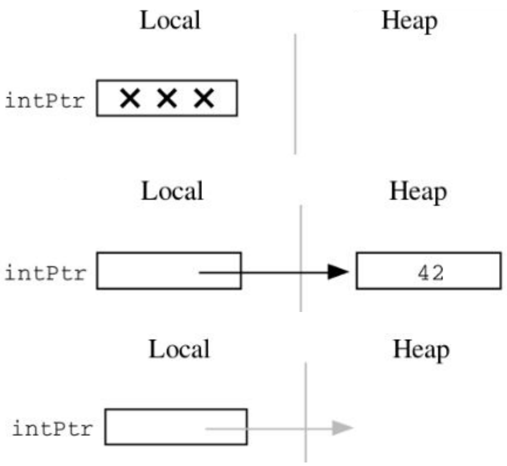
```
void heap_func() {

int* intPtr;

intPtr=
malloc(sizeof(int));

*intPtr=42;

free(intPtr)
}
```



Arrays and Pointer Arithmetic

- Name of an array is the address of the **first element** of the array:

  &arr[0] is equivalent to arr
  &arr[1] is equivalent to (arr+1)
  &arr[i] is equivalent to (arr+i)

- Therefore:
  arr[0] is equivalent to *arr
  arr[1] is equivalent to *(arr+1)
  arr[i] is equivalent to *(arr+i)

- What will be happen if we write:
  arr+1

int a [ ] = {10,20,30,40};

2886728	2886732	2886736	2886740
10	20	30	40
a[0]	a[1]	a[2]	a[3]

Arrays and Pointer Arithmetic

- Name of an array is the address of the **first element** of the array:

  &arr[0] is equivalent to arr
  &arr[1] is equivalent to (arr+1)
  &arr[i] is equivalent to (arr+i)

- Therefore:
  arr[0] is equivalent to *arr
  arr[1] is equivalent to *(arr+1)
  arr[i] is equivalent to *(arr+i)

```
int a [ ] = {10,20,30,40};
```

2886728	2886732	2886736	2886740
10	20	30	40
a[0]	a[1]	a[2]	a[3]

- What will be happen if we write:
  arr+1 does not mean the address is increased by 1!
  e.g. if an element of arr is of size **4 bytes,** it is **increased by 4** (so that it can refer to the **next element**).

Pass-by-value

```
#include<stdio.h>

void swap(int n1, int n2){
 int temp;
 temp = n1;
 n1 = n2;
 n2 = temp;
}
int main(){
 int n1 = 3, n2 = 5;
 swap(n1, n2);
 printf("n1 = %d and n2 = %d \n", n1, n2); // NOT SWAPPED!
 return 0;
}
```

- Let's visualize it! - https://goo.gl/7i2K4H

Pass-by-reference

```
#include<stdio.h>

void swap(int* p1, int* p2){
 int temp;
 temp = *p1;
 *p1 = *p2;
 *p2 = temp;
}
int main(){
 int n1 = 3, n2 = 5;
 int* ptr = &n1;
 swap(ptr, &n2); // or swap(&n1, &n2);
 printf("n1=%d and n2=%d\n", n1, n2); // SWAPPED

 return 0;}
```

- Let’s visualize it! - https://goo.gl/UnLrnP

Things to Remember

- Variables (especially pointers) should be initialized before using.
    - Dereferencing an uninitialized pointer can have arbitrary effects (including **program crash!**).
        - ```
          int* x; // This allocates space for the pointer, but not the pointee.
          *x = 5; // Error (x does not refer to any pointee!)
          ```

- Life of a pointer should be smaller or equal to the life of the object it points to.
    - Do not return local variables by reference!

- We simply write e.g. `printf("%p \n", numPtr);`
  But it is better to write `printf("%p \n", (void*)numPtr);`
  Because **%p** is defined for **void***
  (Otherwise "**undefined behavior**": different compilers may **behave differently**.)
          ```

Programming Advice

- If a pointer is not initialized at declaration, initialize it with NULL, the special value for uninitialized pointer. Before dereferencing a pointer check if value is NULL.

```
int* p = NULL;
// ...
if (p == NULL){
 printf("Cannot dereference pointer p.\n");
 exit(1);
}
// ... *p ...
```

Struct

Struct

- A structure is a **collection of variables** referenced under one name that allows us to define **custom data types**.
- The aim is to keep **related information** together.
- Here below we introduce **person** data type by using **struct** keyword

```
struct person {
    char gender;
    int age;
};
```

- At this point in the code, **no variable** has actually been **declared**. Only the form of the data has been **defined**.

Struct - Assigning values

- It is a way to assign values to the struct:

```
#include <stdio.h>

struct person  {
    char gender;
    int age;
};

int main() {
    struct person my_person = {'F', 36};
    return 0;
}
```

Struct - Accessing elements

- We can access and manipulate the elements of a struct by using **.(dot)** operator.

```c
#include <stdio.h>

struct person  {
    char gender;
    int age;
};

int main() {
    struct person my_person = {'F', 36};
    my_person.age += 5;
    my_person.gender = 'M';

    printf("Gender: %c\n", my_person.gender);
    printf("Age: %d\n", my_person.age);

    return 0; }
```

Struct - Definition

```c
struct addr {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
};
```

Creates a variable type

```c
struct addr addr_info;
```

```c
struct{
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
} addr_info;
```

Creates a variable

```c
addr_info.zip = 2649921;
printf("%d" ,addr_info.zip);
```

Struct - Sizeof

- We can use **sizeof(...)** function to check the size of our custom data type.

```c
#include <stdio.h>

struct person  {
    char gender;
    int age;
};

int main() {

    printf("sizeof(struct person): %ld\n", sizeof(struct person));

    return 0;
}
```

Struct - Padding bytes

- People generally think that the size of a custom data type would be the accumulation of each individual elements' size; however, compilers are free to add some padding bytes to these custom types in order to access the data fast.
- To prevent the compiler adding padding bytes we can use GCC compiler directive.

```
struct person  {
    char gender;
    int age;
}__attribute__((__packed__));
```

Struct - Pointers

- We can define pointers for the variables whose types are our custom data types.

```c
#include <stdio.h>

struct person  {
    char gender;
    int age;
};

int main() {
    struct person my_person = {'F',
    36};
    struct person* my_person_pointer;
    my_person_pointer = &my_person;

    return 0;
}
```

Struct - Individual elements

- We can employ 2 different methods to access individual elements through a pointer (**dereferencing** * or **arrow** operator ->)

```
#include <stdio.h>

struct person  {
    char gender;
    int age;
};

int main() {
    struct person my_person = {'F', 36};
    struct person* my_person_pointer;
    my_person_pointer = &my_person;
    my_person_pointer->age = 20; // or (*my_person_pointer).age = 20;
    (*my_person_pointer).gender = 'M'; // or my_person_pointer->gender = 'M'

    return 0;}
```

Struct - Assigning a structure

- We can assign the values of a structure to another.

```
#include <stdio.h>

struct {
    int a;
    int b;
} x, y;

int main(void)
{
    x.a=10;
    y=x;
    printf("%d", y.a);
    return 0;}
```

Struct - Array

- Structure is a powerful data structure when used with arrays.

```
#include <stdio.h>

struct addr {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
};

int main(void)
{
    struct addr addr_info[100];

    for(i=0;i<100;i++)
    {
        printf("%s", addr_info[i].name);
    }
    return 0;}
```

Struct - Array

Employee number	Employee name	Employee pay rate
32479	Abrams, B.	6.72
33623	Bohm, P.	7.54
34145	Donaldson, S.	5.56
35987	Ernst, T.	5.43

- Program 12.3

```c
#include <stdio.h>
#define NUMRECS 5
struct PayRecord /* construct a global structure type */
{
  int id;
  char name[20];
  double rate;
};

int main()
{
  int i;
  struct PayRecord employee[NUMRECS] = {{32479, "Abrams, B.", 6.72},
                                        {33623, "Bohm, P.", 7.54},
                                        {34145, "Donaldson, S.", 5.56},
                                        {35987, "Ernst, T.", 5.43},
                                        {36203, "Gwodz, K.", 8.72}
                                       };

  for (i = 0; i < NUMRECS; i++)
    printf("%d %-20s %4.2f\n",
           employee[i].id,employee[i].name,employee[i].rate);

  return 0;
}
```

Struct - Passing an element to a func

- When you pass an element of a structure variable to a function, you are actually passing the value of that element to the function.

```
struct person {
     char x;
     int y;
     float z;
     char s[10];
};

int main(void)
{
     struct person mike, fred;
     func(mike.x);
     func2(mike.y);
     func3(mike.s[2]);
     func4(mike.s);

     return 0;}
```

Struct - Passing entire struct to a func

```
#include <stdio.h>

struct arg{
    int a, b;
    char ch;
};

void main(void)
{
    struct arg arguments;
    arguments.a=1000;
    f1(arguments);
}

f1( struct arg param){ printf(“%d”, param.a);}
```

Struct - Nested structure

- When a structure is an element of another structure, it is called a nested structure.

```
struct addr {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
};

struct emp {
    struct addr address;
    float wage;
} worker;

worker.address.zip = 379311;
```

Struct - Typedef

- Typedef provides a simple method for creating a new and typically shorter name for an existing structure type. For example:

```
struct date{
     int month;
     int day;
     int year;
};
```

- Then typedef statement:

```
typedef struct date myDate;
```

- Instead:

```
struct date a, b, c;
```

- We can type:

```
myDate a, b, c;
```