User Defined Datatypes

Advanced C

User Defined Datatypes (Composite Data Types)

- Sometimes it becomes tough to build a whole software that works only with integers, floating values, and characters.
- In circumstances such as these, you can create your own data types which are based on the standard ones
- There are some mechanisms for doing this in C:
  - Structures (derived)
  - Unions (derived)
  - Typedef (storage class)
  - Enums (user defined)
- Hoo!!, let's not forget our old friend **_r_a_** which is a user defined data type too!!.

: Composite (or Compound) Data Type :

- Any data type which can be constructed from primitive data types and other composite types
- It is sometimes called a structure or aggregate data type
- Primitives types – int, char, float, double

Advanced C
UDTs

Syntax

```
struct StructureName
{
    /* Group of data types */
};
```

- If we consider the Student as an example, The admin should have at least some important data like name, ID and address.

- So if we create a structure of the above requirement, it would look like,

Example

```
struct Student
{
    int id;
    char name[20];
    char address[60];
};
```

Advanced C

UDTs – Structures – Declaration and definition

001_example.c

```
struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1;

    return 0;
}
```

- Name of the **datatype.** Note it's **struct Student** and not Student

- Are called as **fields** or **members** of of the structure

- Declaration **ends** here
- The memory is not yet allocated!!

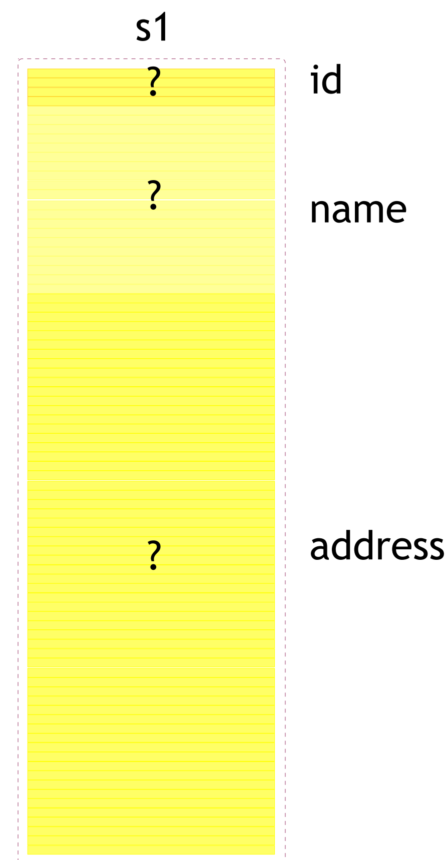
- **s1** is a **variable** of **type struct Student**
- The memory is allocated now

Advanced C

UDTs - Structures - Memory Layout

001_example.c

```c
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1;

    return 0;
}
```



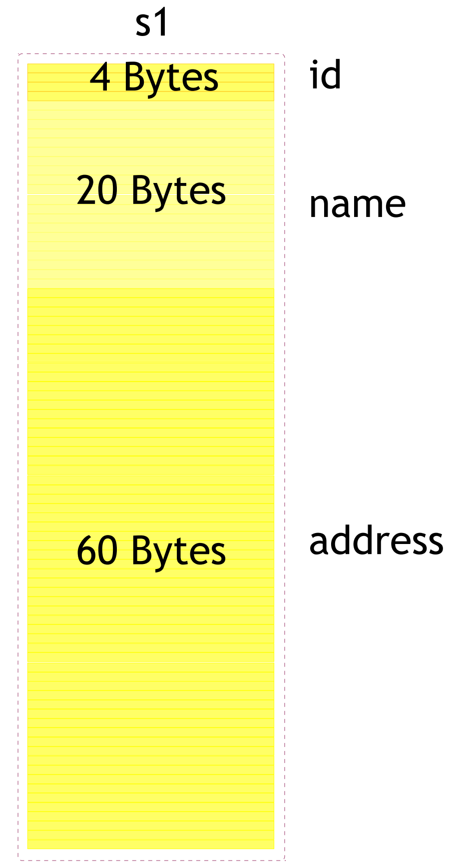
- What does s1 contain?
- How can we draw it's memory layout?

Advanced C

UDTs – Structures – Memory Layout

002_example.c

```
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1;

    printf("%zu\n", sizeof(struct Student));
    printf("%zu\n", sizeof(s1));

    return 0;
}
```



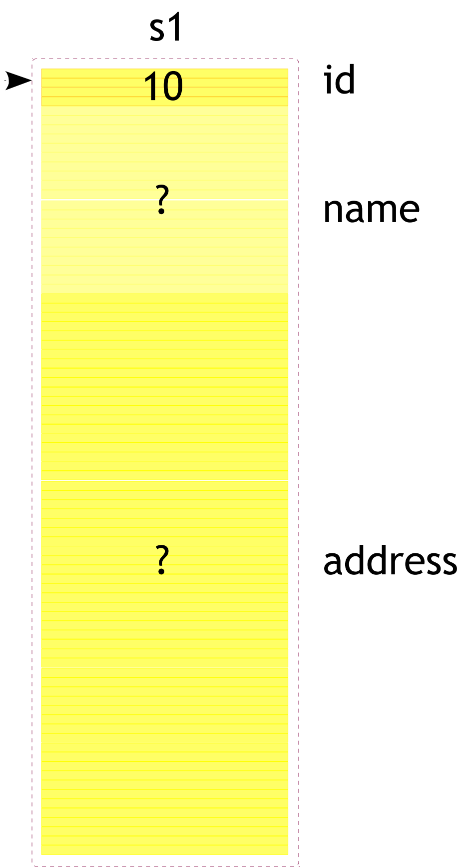
Structure size depends in the member arrangment!!. Will discuss that shortly

Advanced C

UDTs - Structures - Access

003_example.c

```c
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1;

    s1.id = 10;

    return 0;
}
```



- How to write into id now?
- It's by using "." (Dot) operator (member access operator)
- Now please assign the name member of s1

Advanced C

UDTs - Structures - Initialization

004_example.c

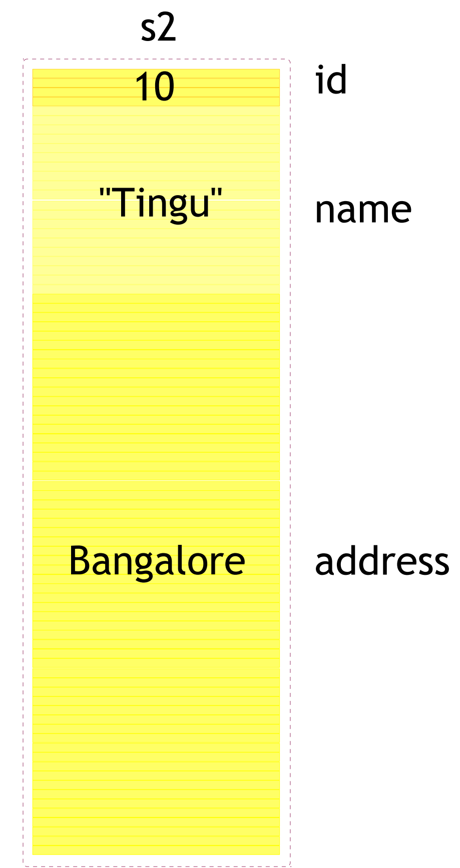
```c
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1 = {10, "Tingu", "Bangalore"};

    return 0;
}
```

Advanced C

UDTs – Structures - Copy

005_example.c

```c
#include <stdio.h>

struct Student
{
	int id;
	char name[20];
	char address[60];
};

int main()
{
	struct Student s1 = {10, “Tingu”, “Bangalore”};
	struct Student s2;

	s2 = s1;

	return 0;
}
```



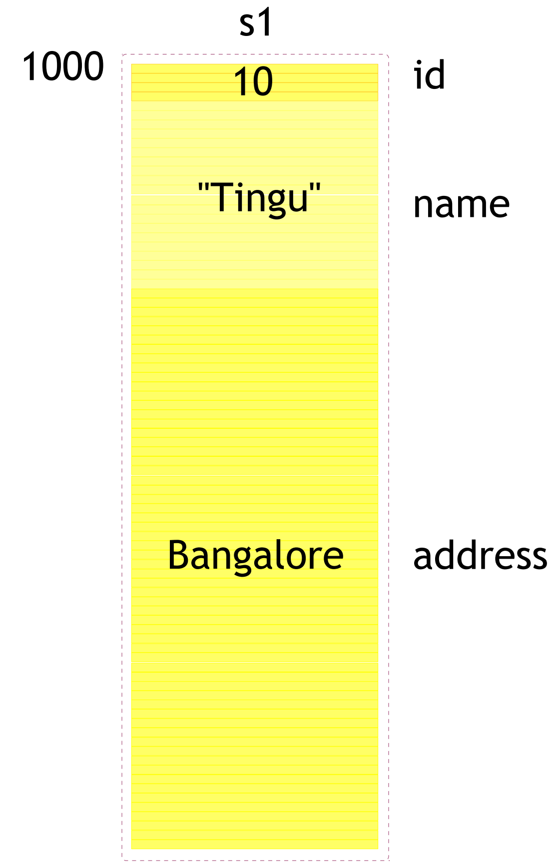
Structure name does not represent its address. (No correlation with arrays)

Advanced C

UDTs - Structures - Address

006_example.c

```c
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1 = {10, “Tingu”, “Bangalore”};

    printf(“Struture starts at %p\n”, &s1);
    printf(“Member id is at %p\n”, &s1.id);
    printf(“Member name is at %p\n”, s1.name);
    printf(“Member address is at %p\n”, s1.address);

    return 0;
}
```



ΣMERTXE

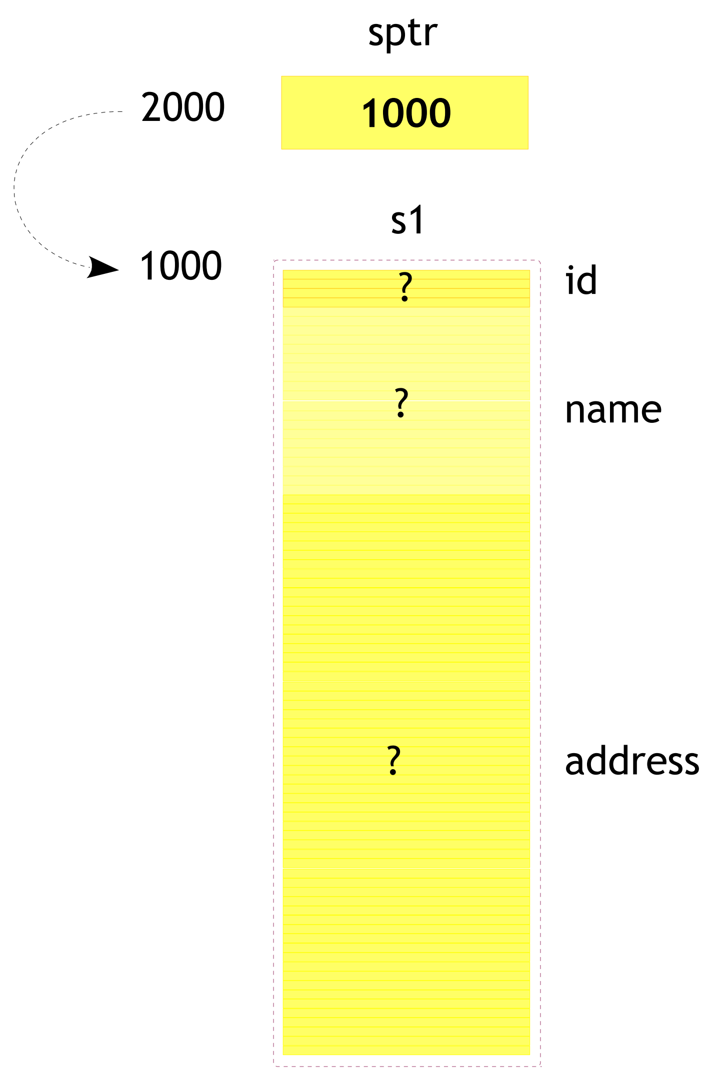
- Pointers!!!. Not again ;). Fine don't worry, not a big deal
- But do you any idea how to create it?
- Will it be different from defining them like in other data types?

Advanced C

UDTs - Structures - Pointer

007_example.c

```
#include <stdio.h>

struct Student
{
	int id;
	char name[20];
	char address[60];
};

static struct Student s1;

int main()
{
	struct Student *sptr = &s1;

	return 0;
}
```

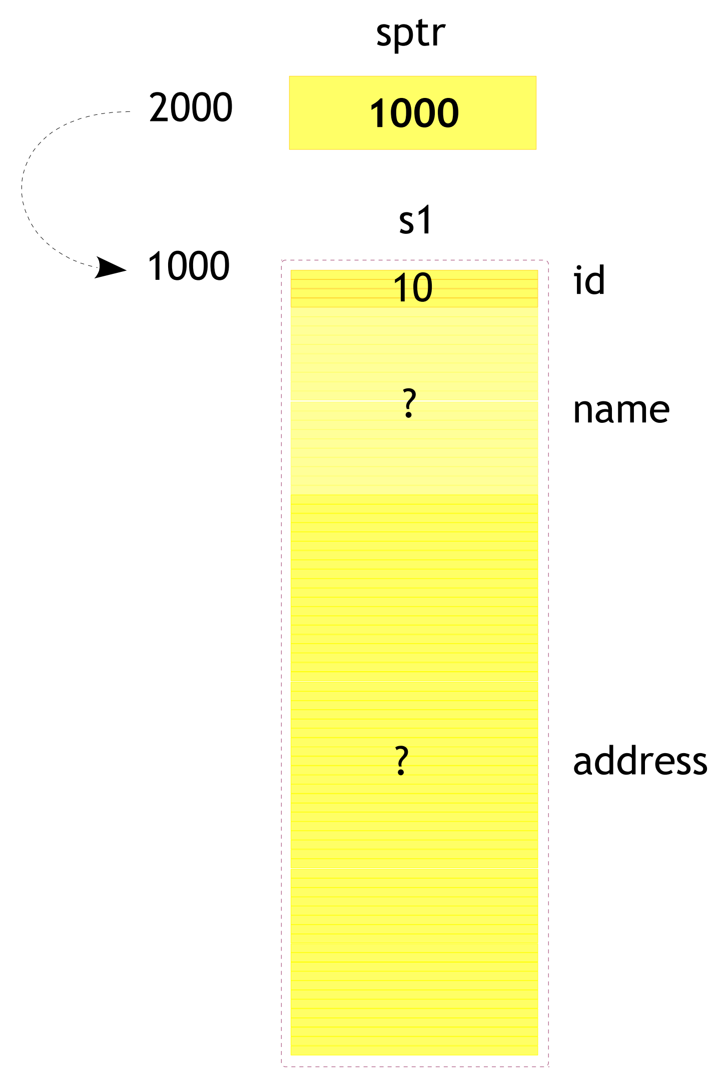


Advanced C

UDTs - Structures - Pointer - Access

008_example.c

```c
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

static struct Student s1;

int main()
{
    struct Student *sptr = &s1;

    (*sptr).id = 10;

    return 0;
}
```

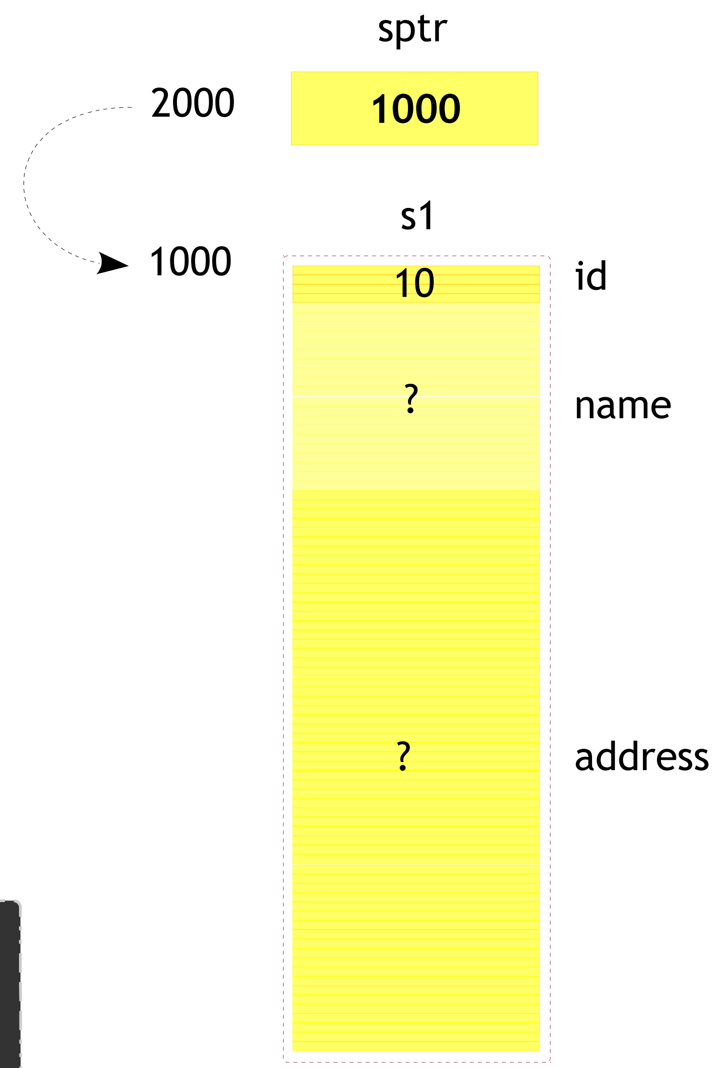


Advanced C

UDTs - Structures - Pointer - Access - Arrow

009_example.c

```
#include <stdio.h>

struct Student
{
     int id;
     char name[20];
     char address[60];
};

static struct Student s1;

int main()
{
     struct Student *sptr = &s1;

     sptr->id = 10;

     return 0;
}
```



Note: we can access the structure pointer as seen in the previous slide. The Arrow operator is just convenience and frequently used

- The structures can be passed as parameter and can be returned from a function
- This happens just like normal datatypes.
- The parameter passing can have two methods again as normal
  - Pass by value
  - Pass by reference

Advanced C

UDTs – Structures – Functions – Pass by Value

010_example.c

```
#include <stdio.h>

struct Student
{
    int id;
    char name[30];
    char address[150];
};

void data(struct Student s)
{
    s.id = 10;
}

int main()
{
    struct Student s1;

    data(s1);

    return 0;
}
```

Not recommended on larger structures

011_example.c

```
#include <stdio.h>

struct Student
{
    int id;
    char name[30];
    char address[150];
};

void data(struct Student *s)
{
    s->id = 10;
}

int main()
{
    struct Student s1;

    data(&s1);

    return 0;
}
```

Recommended on larger structures

Advanced C

UDTs - Structures - Functions - Return

012_example.c

```
struct Student
{
    int id;
    char *name;
    char *address;
};

struct Student data(void)
{
    struct Student s;

    s.name = (char *) malloc(30 * sizeof(char));
    s.address = (char *) malloc(150 * sizeof(char));

    return s;
}

int main()
{
    struct Student s1;

    s1 = data();

    return 0;
}
```

- Adding of few extra useless bytes (in fact skip address) in between the address of the members are called structure padding.
- What!!?, wasting extra bytes!!, Why?
- This is done for Data Alignment.
- Now!, what is data alignment and why did this issue suddenly arise?
- No its is not sudden, it is something the compiler would be doing internally while allocating memory.
- So let's understand data alignment in next few slides

Data Alignment

- A way the data is arranged and accessed in computer memory.
- When a modern computer reads from or writes to a memory address, it will do this in word sized chunks (4 bytes in 32 bit system) or larger.
- The main idea is to increase the efficiency of the CPU, while handling the data, by arranging at a memory address equal to some multiple of the word size
- So, Data alignment is an important issue for all programmers who directly use memory.

- If you don't understand data and its address alignment issues in your software, the following scenarios, in increasing order of severity, are all possible:
  - Your software will run slower.
  - Your application will lock up.
  - Your operating system will crash.
  - Your software will silently fail, yielding incorrect results.

Data Alignment

Example

```c
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

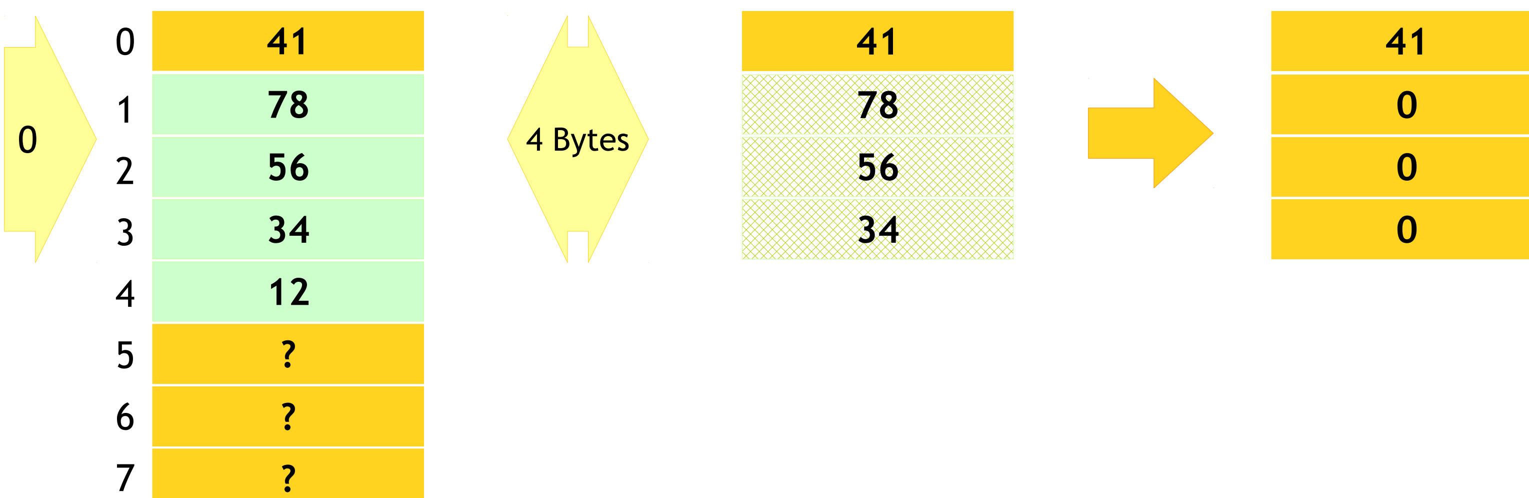
0	ch
1	78
2	56
3	34
4	12
5	?
6	?
7	?

- Lets consider the code as given
- The memory allocation we expect would be like shown in figure
- So lets see how the CPU tries to access these data in next slides

Example

```c
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching a character by the CPU will be like shown below

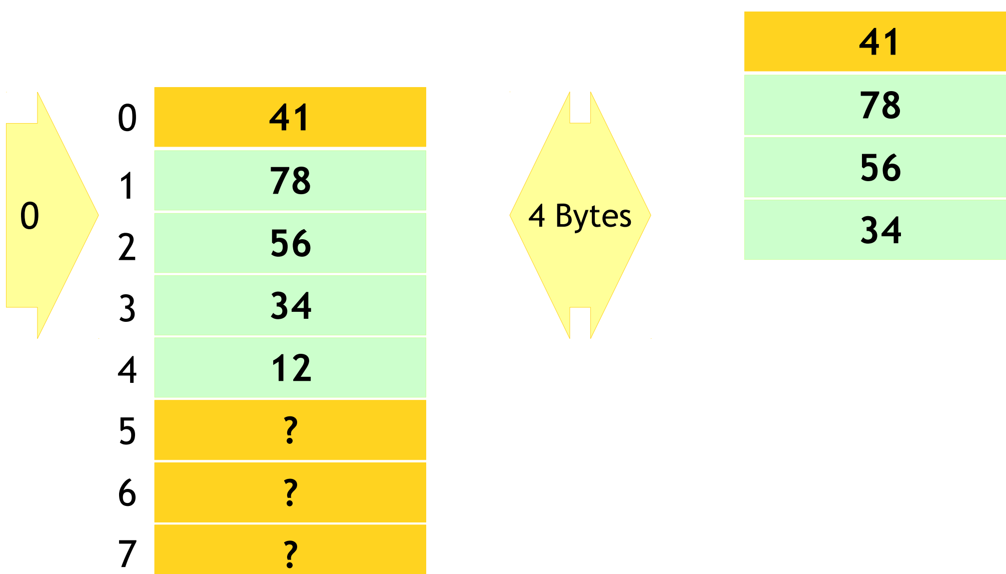


Advanced C
Data Alignment

Example

```c
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching integer by the CPU will be like shown below



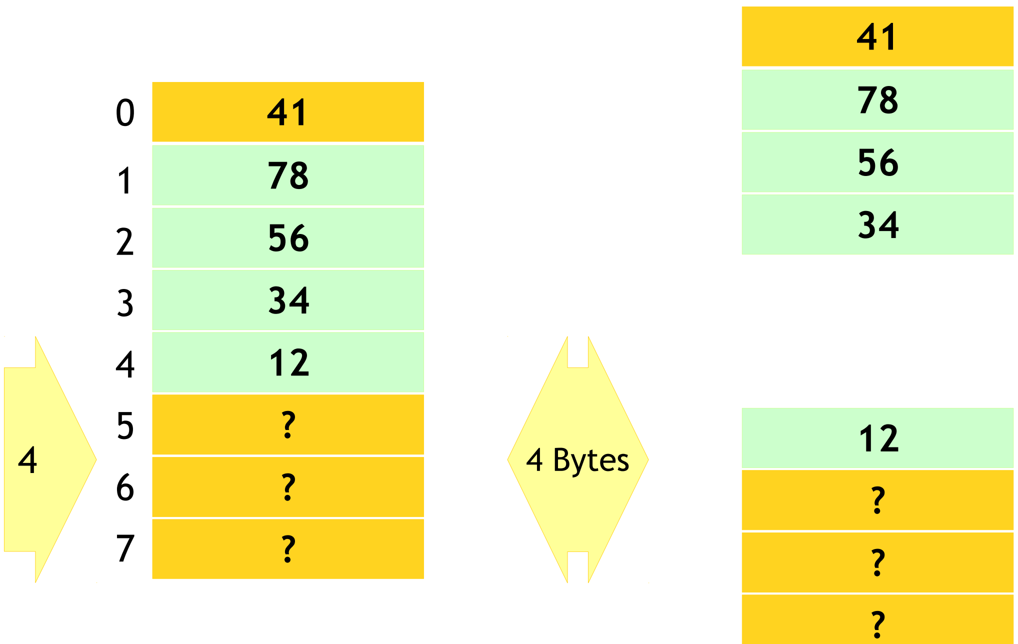
Advanced C

Data Alignment

Example

```c
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below

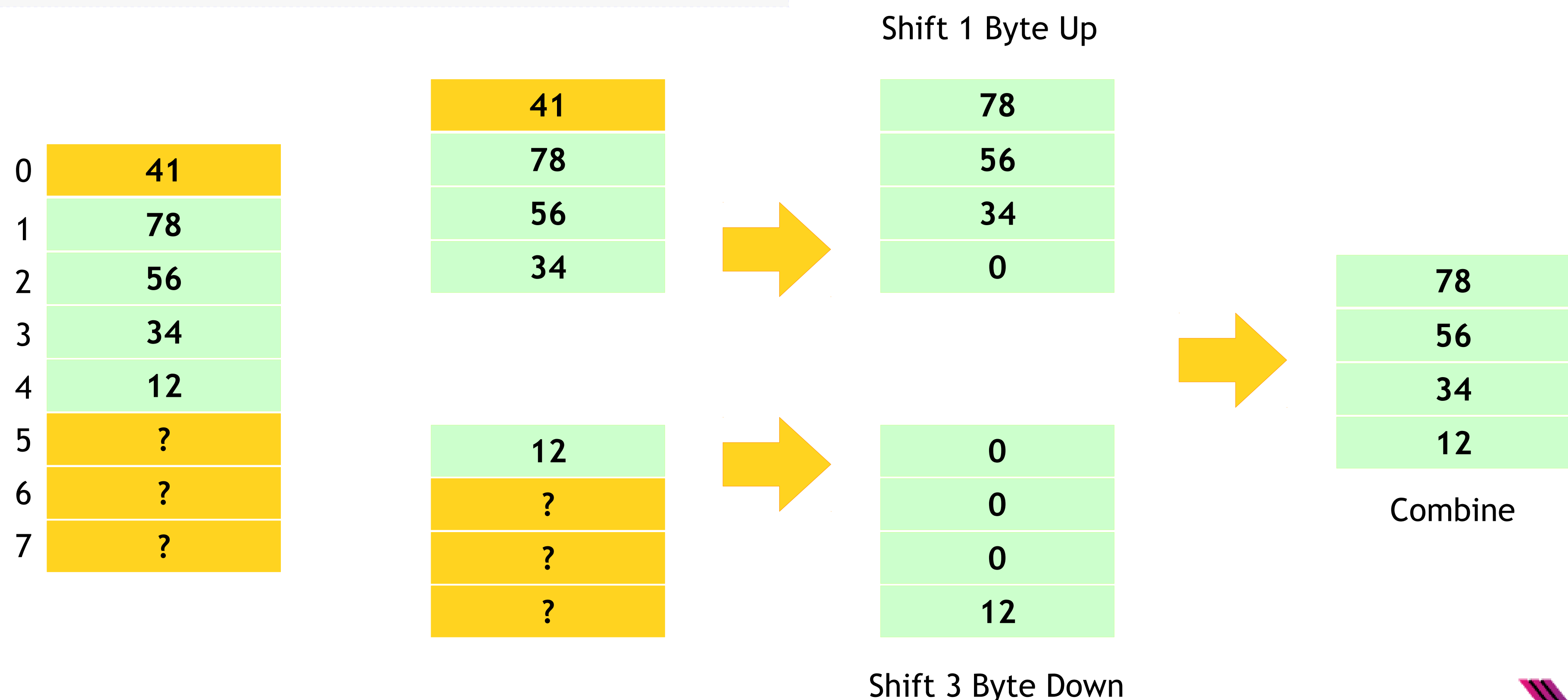


Advanced C
Data Alignment

Example

```c
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below



Advanced C

UDTs – Structures – Data Alignment - Padding

- Because of the data alignment issue, structures uses padding between its members if they don't fall under even address.
- So if we consider the following structure the memory allocation will be like shown in below figure

Example

```c
struct Test
{
    char ch1;
    int num;
    char ch2;
}
```

0	ch1
1	pad
2	pad
3	pad
4	num
5	num
6	num
7	num
8	ch2
9	pad
A	pad
B	pad

Advanced C

UDTs – Structures – Data Alignment - Padding

- You can instruct the compiler to modify the default padding behavior using #pragma pack directive

Example

```c
#pragma pack(1)

struct Test
{
    char ch1;
    int num;
    char ch2;
};
```

0	ch1
1	num
2	num
3	num
4	num
5	ch2

Advanced C

UDTs - Structures - Padding

013_example.c

```
#include <stdio.h>

struct Student
{
    char ch1;
    int num;
    char ch2;
};

int main()
{
    struct Student s1;

    printf("%zu\n", sizeof(struct Student));

    return 0;
}
```

Advanced C

UDTs – Structures – Padding

014_example.c

```
#include <stdio.h>

#pragma pack(1)

struct Student
{
    char ch1;
    int num;
    char ch2;
};

int main()
{
    struct Student s1;

    printf("%zu\n", sizeof(struct Student));

    return 0;
}
```

- The compiler generally gives the memory allocation in multiples of bytes, like 1, 2, 4 etc.,
- What if we want to have freedom of having getting allocations in bits?!.
- This can be achieved with bit fields.
- But note that
  - The minimum memory allocation for a bit field member would be a byte that can be broken in max of 8 bits
  - The maximum number of bits assigned to a member would depend on the length modifier
  - The default size is equal to word size

Example

```
struct Nibble
{
    unsigned char lower    : 4;
    unsigned char upper    : 4;
};
```

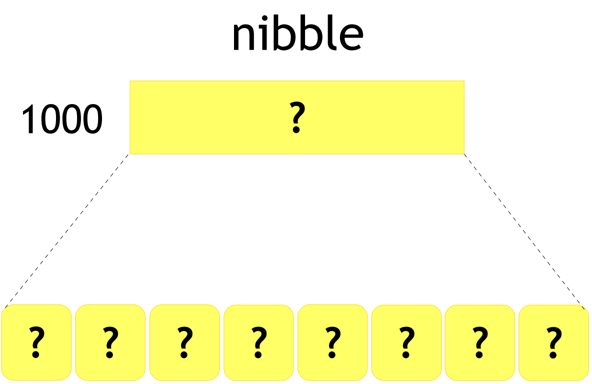
- The above structure divides a char into two nibbles
- We can access these nibbles independently

Advanced C

UDTs – Structures – Bit Fields

015_example.c

```
struct Nibble
{
    unsigned char lower    : 4;
    unsigned char upper    : 4;
};

int main()
{
    struct Nibble nibble;

    nibble.upper = 0x0A;
    nibble.lower = 0x02;

    return 0;
}
```

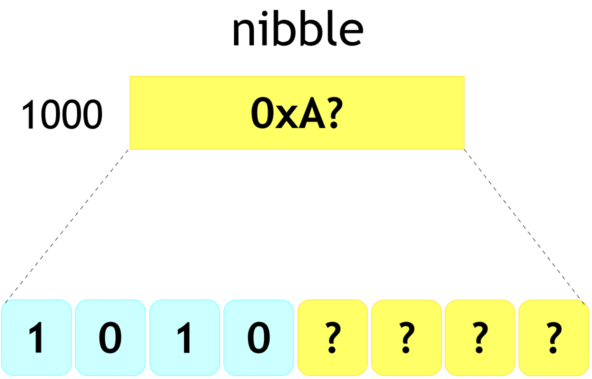


Advanced C

UDTs – Structures – Bit Fields

015_example.c

```c
struct Nibble
{
    unsigned char lower    : 4;
    unsigned char upper    : 4;
};

int main()
{
    struct Nibble nibble;

    nibble.upper = 0x0A;
    nibble.lower = 0x02;

    return 0;
}
```

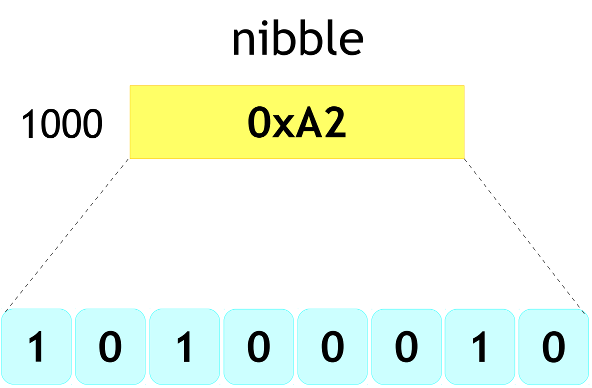


Advanced C

UDTs – Structures – Bit Fields

015_example.c

```c
struct Nibble
{
    unsigned char lower    : 4;
    unsigned char upper    : 4;
};

int main()
{
    struct Nibble nibble;

    nibble.upper = 0x0A;
    nibble.lower = 0x02;

    return 0;
}
```



016_example.c

```c
struct Nibble
{
    unsigned char lower    : 4;
    unsigned char upper    : 4;
};

int main()
{
    struct Nibble nibble;

    printf("%zu\n", sizeof(nibble));

    return 0;
}
```

Advanced C

UDTs – Structures – Bit Fields

017_example.c

```
struct Nibble
{
    unsigned lower : 4;
    unsigned upper : 4;
};

int main()
{
    struct Nibble nibble;

    printf("%zu\n", sizeof(nibble));

    return 0;
}
```

Advanced C

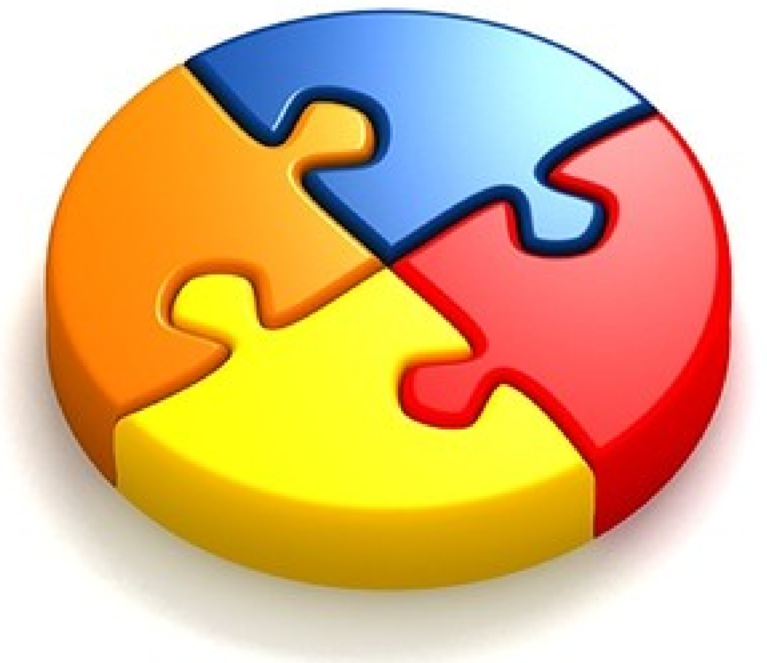
UDTs – Structures – Bit Fields

018_example.c

```
struct Nibble
{
    char lower : 4;
    char upper : 4;
};

int main()
{
    struct Nibble nibble;

    nibble.upper = 0x0A;
    nibble.lower = 0x02;

    printf(“%d\n”, nibble.upper);
    printf(“%d\n”, nibble.lower);

    return 0;
}
```

019_example.c

```
struct Nibble
{
    char lower : 4;
    char upper : 4;
};

int main()
{
    struct Nibble nibble = {0x02, 0x0A};

    printf(“%#o\n”, nibble.upper);
    printf(“%#x\n”, nibble.lower);

    return 0;
}
```



- Like structures, unions may have different members with different data types.
- The major difference is, the structure members get different memory allocation, and in case of unions there will be single memory allocation for the biggest data type

Advanced C

UDTs – Unions

Example

```
union Test
{
    char option;
    int id;
    double height;
};
```

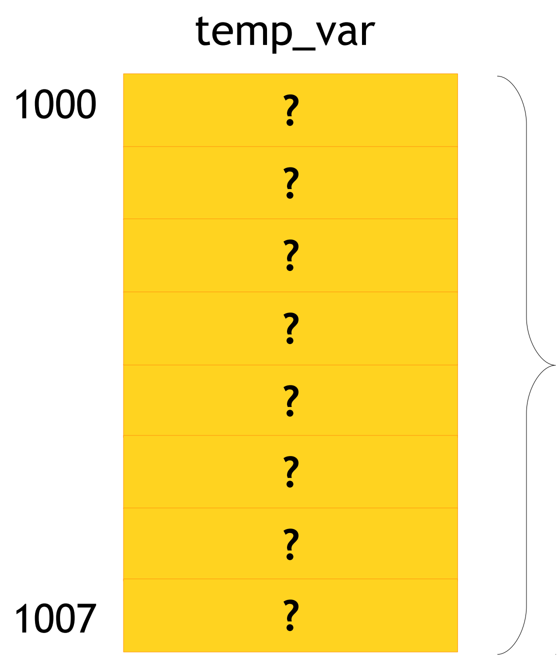
- The above union will get the size allocated for the type double
- The size of the union will be 8 bytes.
- All members will be using the same space when accessed
- The value the union contain would be the latest update
- So as summary a single variable can store different type of data as required

Advanced C

UDTs – Unions

020_example.c

```c
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

    temp_var.height = 7.2;
    temp_var.id = 0x1234;
    temp_var.option = '1';

    return 0;
}
```



Total 8 Bytes allocated since longest member is **double**

020_example.c

```c
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

    temp_var.height = 7.2;
    temp_var.id = 0x1234;
    temp_var.option = '1';

    return 0;
}
```

temp_var

	temp_var
1000	0xCD
	0xCC
	0xCC
	0xCC
	0xCC
	0xCC
	0X1C
1007	0X40

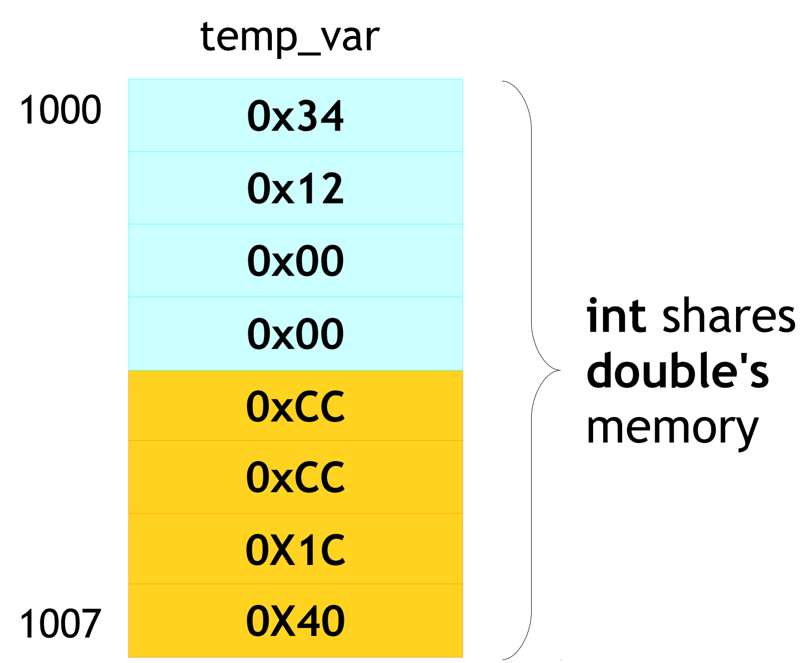
double uses its memory

Advanced C

UDTs – Unions

020_example.c

```
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

    temp_var.height = 7.2;
    temp_var.id = 0x1234;
    temp_var.option = '1';

    return 0;
}
```



Advanced C

UDTs – Unions

020_example.c

```c
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

    temp_var.height = 7.2;
    temp_var.id = 0x1234;
    temp_var.option = '1';

    return 0;
}
```

temp_var

1000	0x31
	0x12
	0x00
	0x00
	0xCC
	0xCC
	0X1C
1007	0X40

char shares **double's** memory

Advanced C

UDTs – Unions

021_example.c

```c
union FloatBits
{
    float degree;
    struct
    {
        unsigned m : 23;
        unsigned e : 8;
        unsigned s : 1;
    } elements;
};

int main()
{
    union FloatBits fb = {3.2};

    printf("Sign: %X\n", fb.elements.s);
    printf("Exponent: %X\n", fb.elements.e);
    printf("Mantissa: %X\n", fb.elements.m);

    return 0;
}
```

	fb
1000	0xCD
	0xCC
	0x4C
1004	0x40

022_example.c

```
union Endian
{
    unsigned int vlaue;
    unsigned char byte[4];
};

int main()
{
    union Endian e = {0x12345678};

    e.byte[0] == 0x78 ? printf("Little\n") : printf("Big\n");

    return 0;
}
```

- Typedef is used to create a new name to the existing types.
- K&R states that there are two reasons for using a typedef.
  - First, it provides a means to make a program more portable. Instead of having to change a type everywhere it appears throughout the program's source files, only a single typedef statement needs to be changed.
  - Second, a typedef can make a complex definition or declaration easier to understand.

Advanced C

UDTs - Typedefs

023_example.c

```c
typedef unsigned int uint;

int main()
{
    uint number;

    return 0;
}
```

024_example.c

```c
typedef int * int_ptr;
typedef float * float_ptr;

int main()
{
    int_ptr ptr1, ptr2, ptr3;
    float_ptr fptr;

    return 0;
}
```

025_example.c

```c
typedef int array_of_100[100];

int main()
{
    array_of_100 array;

    printf("%zu\n", sizeof(array));

    return 0;
}
```

Advanced C

UDTs - Typedefs

026_example.c

```
typedef struct _Student
{
    int id;
    char name[30];
    char address[150]
} Student;

void data(Student s)
{
    s.id = 10;
}

int main()
{
    Student s1;

    data(s1);

    return 0;
}
```

027_example.c

```
#include <stdio.h>

typedef int (*fptr)(int, int);

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    fptr function;

    function = add;
    printf("%d\n", function(2, 4));

    return 0;
}
```

Advanced C

UDTs - Typedefs

028_example.c

```
#include <stdio.h>

typedef signed int      sint, si;
typedef unsigned int    uint, ui;
typedef signed char     s8;
typedef signed short    s16;
typedef signed int      s32;
typedef unsigned char   u8;
typedef unsigned short  u16;
typedef unsigned int    u32;

int main()
{
    u8 count = 200;
    s16 axis = -70;

    printf("%u\n", count);
    printf("%d\n", axis);

    return 0;
}
```

Example

```
size_t – stdio.h
ssize_t – stdio.h
va_list – stdarg.h
```

Advanced C

UDTs – Typedefs - Usage

029_example.c

```
typedef struct Sensor {
    int id;
    char name[12];
    int version;
    /*
     * The members of an anonymous union
     * are considered to be members of the
     * containing structure.
     */
    union { // Anonymous union
        float temperature;
        float humidity;
        char motion[4];
    };
} Sensor;
```

Advanced C

UDTs – Enums

- Set of named integral values
- Generally referred as named integral constants

Syntax

```
enum name
{
    /* Members separated with , */
};
```

030_example.c

```
enum bool
{
    e_false,
    e_true
};

int main()
{
    printf("%d\n", e_false);
    printf("%d\n", e_true);

    return 0;
}
```

- The above example has two members with its values starting from 0.

  i.e, e_false = 0 and e_true = 1.

Advanced C

UDTs – Enums

031_example.c

```
typedef enum
{
    e_red = 1,
    e_blue = 4,
    e_green
} Color;

int main()
{
    Color e_white = 0, e_black;

    printf("%d\n", e_white);
    printf("%d\n", e_black);
    printf("%d\n", e_green);

    return 0;
}
```

- The member values can be explicitly initialized
- There is no constraint in values, it can be in any order and same values can be repeated
- The derived data type can be used to define new members which will be uninitialized

Advanced C

UDTs – Enums

032_example.c

```
int main()
{
    typedef enum
    {
        red,
        blue
    } Color;

    int blue;

    printf("%d\n", blue);
    printf("%d\n", blue);

    return 0;
}
```

- Enums does not have name space of its own, so we cannot have same name used again in the same scope.

Advanced C

UDTs – Enums

033_example.c

```
typedef enum
{
        red,
        blue,
        green
} Color;

int main()
{
    Color c;

    printf("%zu\n", sizeof(Color));
    printf("%zu\n", sizeof(c));

    return 0;
}
```

- Size of Enum does not depend on number of members

Advanced C
UDTs - DIY

- WAP to accept students record. Expect the below output

Screen Shot

```
user@user:~]./students_record.out
Enter the number of students : 2
Enter name of the student : Tingu
Enter P, C and M marks : 23 22 12
Enter name of the student : Pingu
Enter P, C and M marks : 98  87  87
------------------------------------------------------
Name            Maths         Physics        Chemistry
------------------------------------------------------
Tingu              12              23               22
Pingu              87              98               87
------------------------------------------------------
Average         49.50           60.50            54.50
------------------------------------------------------
user@user:~]
```

- WAP to program to swap a nibble using bit fields