Union

The data structure `union` allows different types of data to be stored in the same memory location

The data stored in the memory location must be read in the same format as it was exported

For example, a number can be an integer or a floating point number; but you want to minimize unnecessary memory usage

Let's make a `union` with `int` (4 bytes) and `double` (8 bytes)

Only one of these is used at a time, so 8 bytes of memory is required, not 12 bytes

File: Lecture5 > `Example1.c`

Accessing union members

To access any member of a union, we use the member access operator . (dot)

The member access operator works similarly as in the case of struct.

The difference is that we can now use only one “field” (member).

File: `Example2.c`

- The values of `i` and `f` members of union got corrupted, since the final value has occupied the memory location. The value of str member is getting printed very well.

File: `Example3.c`

- All the members are getting printed correcly because one member is being used at a time.

Enumeration

An enumeration type is a data type that consists of integral constants. To define enums, the `enum` keyword is used.

```
enum flag {const1, const2, ..., constN};
```

By default, `const1` is 0, `const2` is 1 and so on. You can change default values of enum elements during declaration (if necessary).

```
enum day {sunday = 1, monday, tuesday,
      wednesday, thursday, friday, saturday};
```

How to use enums for flags?

Suppose you are designing a button for Windows application. You can set flags ITALICS, BOLD and UNDERLINE to work with text.

```
enum designFlags {
     ITALICS = 1,
     BOLD = 2,
     UNDERLINE = 4
};
```

How to use enums for flags?

In binary:

- ITALICS = 00000001
- BOLD = 00000010
- UNDERLINE = 00000100

Since these constants are a power of 2, you can combine two or more flags at once without overlapping using bitwise OR | operator.

```
int myDesign = BOLD | UNDERLINE;
```

File: `Example4.c`

Dynamic data structures

As we know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions `malloc()`, `calloc()`, `realloc()` and `free()` are used.

These functions are defined in `stdlib.h` header file.

`malloc()`

The name "malloc" stands for memory allocation.

The `malloc()` function reserves a block of memory of the specified number of bytes.

Syntax: `ptr = (castType*) malloc(size);`

Example: `ptr = (float*) malloc(100 * sizeof(float));`

The above statement allocates 400 bytes of memory (size of float is 4 bytes).

The pointer `ptr` holds the address of the first byte in the allocated memory.

The expression results in a `NULL` pointer if the memory cannot be allocated.

`calloc()`

The name "calloc" stands for contiguous allocation.

`malloc()` is used to allocate a single block of memory space while the `calloc()` is used to allocate multiple blocks of memory space. Each block allocated by the `calloc()` function is of the same size.

The `malloc()` function allocates memory and leaves the memory uninitialized, whereas the `calloc()` function allocates memory and initializes all bits to zero.

Syntax: `ptr = (castType*)calloc(n, size);`

Example: `ptr = (float*) calloc(25, sizeof(float));`

- allocates contiguous space in memory for 25 elements of type float.

free()

Dynamically allocated memory created with either `calloc()` or `malloc()` doesn't get freed on their own. You must explicitly use `free()` to release the space.

If memory allocation fails, we should tell about it and exit the program:

```
if(ptr == NULL) {
  printf("Error! memory not allocated.");
  exit(0);
}
```

File: `Example5.c (malloc and free)`

File: `Example6.c (calloc and free)`

`realloc()`

Reallocates the given area of memory. It must be previously allocated by `malloc(), calloc()` or `realloc()`.

The reallocation is done by either:

a) expanding the existing area pointed to by ptr, if possible. The contents of the area remain unchanged up to the smaller of the new and old sizes. If the area is expanded, the contents of the new part of the array are undefined.

b) allocating a new memory block of size new_size bytes, copying memory area with size equal the lesser of the new and the old sizes, and freeing the old block.

If there is not enough memory, the old memory block is not freed and null pointer is returned.

File: `Example7.c`

The memory and memory addresses (recap)

- The memory is where variables and other information are stored while a program runs.
- The computer's memory is a collection of bytes, each with an integer address. For example, there is a byte with address 1, another with address 2, etc., up to a very large number.
- A program can fetch the current contents of the byte at a given memory address and it can store a given value into that byte.
- A byte is just 8 bits. Most of the data items that you use are larger than that. For example, a value of type int is usually 32 bits, so it occupies 4 bytes.
- A program refers to a block of memory using the address of the first byte in the block. For example, an integer stored in bytes 1000-1003 has address 1000.

Pointers and memory addresses as values

A memory address is called a pointer because you can think of it as pointing to a specific spot in memory.

Each pointer has a type that tells the type of thing in memory that it points to. To write the type of a pointer, write an asterisk after another type. For example,

- A value of type `int*` is a pointer to a location in memory that holds a value of type int.
- A value of type `double*` is a pointer to a location in memory that holds a value of type double.
- A value of type `char*` is a pointer to a location in memory that holds a value of type char.
- A value of type `char**` is a pointer to a location in memory that holds a value of type `char*`. That is, it points to another pointer. You can add a * to any type, including a pointer type.

If `x` is a variable then `&x` is the address where x is stored.

Null pointer

- Memory address 0 is called the **null pointer**. Your program is never allowed to look at or store anything into memory address 0, so the null pointer is a way of saying "a pointer to nothing". Note that a null pointer is not the same as a null character; do not confuse the two.

For example:

```
char* p = NULL;
```

creates a pointer variable p and makes it hold memory address 0.

Position of “*”

For the complier, it is the same whether you write:

```
type*  var;
type  *var;
type * var;
```

Arrow operator

An **arrow operator** is used with a pointer variable pointing to a structure or union. The arrow operator is formed by using a minus sign, followed by the greater than symbol as shown below.

Syntax: `(pointer_name)->(variable_name)`

- The dot operator (.) is used to normally access members of a structure or union.
- The arrow operator (->) is used to access the members of the structure or the unions using pointers.

File: `Example8.c`

Example

In our final example, there is another way to get information from a function to the calling program using dynamic memory allocation.

In the main program, the program allocates only a pointer to the desired data structure, and the subroutine `queryData()` dynamically allocates the required memory with malloc, fills in the data, and returns the address of the allocated memory area to the calling program.

In this way, the calling program can use the information collected in the function `queryData()` and free up the reserved memory space when it is no longer needed.

File: `Example9.c`