Why do we need C Unions?

Asked 16 years, 1 month ago Modified 1 month ago Viewed 185k times

318

When should unions be used? Why do we need them?

c unions

Share

Improve this question

Follow

edited Aug 14, 2012 at 17:49

timrau

23k ● 4 ● 52 ● 65

asked Oct 31, 2008 at 3:51

ramu

1 Related: [Portability of using union for conversion](https://stackoverflow.com/) – Gabriel Staples Jul 27, 2022 at 13:38

21 Answers

Sorted by: Highest score (default)

321

Unions are often used to convert between the binary representations of integers and floats:

```
union
{
  int i;
  float f;
} u;
```

```
// Convert floating-point bits to integer:
u.f = 3.14159f;
printf("As integer: %08x\n", u.i);
```

Although this is technically undefined behavior according to the C standard (you're only supposed to read the field which was most recently written), it will act in a well-defined manner in virtually any compiler.

Unions are also sometimes used to implement pseudo-polymorphism in C, by giving a structure some tag indicating what type of object it contains, and then unioning the possible types together:

```
enum Type { INTS, FLOATS, DOUBLE };
struct S
{
  Type s_type;
  union
  {
    int s_ints[2];
    float s_floats[2];
    double s_double;
  };
};

void do_something(struct S *s)
{
  switch(s->s_type)
  {
    case INTS:  // do something with s->s_ints
      break;

    case FLOATS:  // do something with s->s_floats
      break;

    case DOUBLE:  // do something with s->s_double
      break;
    }
}
```

This allows the size of `struct S` to be only 12 bytes, instead of 28.

Share Improve this answer

Follow

edited Aug 17, 2012 at 17:03

rubenvb
76.4k ● 37 ● 197 ● 346

answered Oct 31, 2008 at 4:02

Adam Rosenfield
399k ● 101 ● 522 ● 597

there should be u.y instead of u.f – Amit Singh Tomar Sep 8, 2011 at 8:17

1 Does the example which suppose to convert float to integer works? I don`t think so, as int and float are stored in different formats in memory. Can you explain your example? – spin_eight Oct 10, 2012 at 10:51

7 @spin_eight: It's not "converting" from float to int. More like "reinterpreting the binary representation of a float as if it were an int". The output is not 3: ideone.com/MKjwon I'm not sure why Adam is printing as hex, though. – endolith Feb 21, 2013 at 17:19

@Adam Rosenfield i didn't really understund the conversion i don't get a integer in the output :p – The Beast Jan 18, 2016 at 16:38

12 I feel the disclaimer about being undefined behavior should be removed. It is, in fact, defined behavior. See footnote 82 of the C99 standard: *If the member used to access the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part*

of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called "type punning"). This might be a trap representation. – Christian Gibbons Jun 18, 2018 at 20:29

188

Unions are particularly useful in Embedded programming or in situations where direct access to the hardware/memory is needed. Here is a trivial example:

```
typedef union
{
    struct {
        unsigned char byte1;
        unsigned char byte2;
        unsigned char byte3;
        unsigned char byte4;
    } bytes;
    unsigned int dword;
} HW_Register;
HW_Register reg;
```

Then you can access the reg as follows:

```
reg.dword = 0x12345678;
reg.bytes.byte3 = 4;
```

Endianness (byte order) and processor architecture are of course important.

Another useful feature is the bit modifier:

```
typedef union
{
    struct {
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char b4:1;
        unsigned char reserved:4;
    } bits;
    unsigned char byte;
} HW_RegisterB;
HW_RegisterB reg;
```

With this code you can access directly a single bit in the register/memory address:

```
x = reg.bits.b2;
```

Share Improve this answer Follow

edited Dec 9, 2013 at 22:59

torrential coding
1,765 ● 2 ● 24 ● 34

answered Oct 31, 2008 at 7:13

kgiannakakis
104k ● 28 ● 161 ● 197

---

10 Your answer here in conjunction with @Adam Rosenfield's answer above make the perfect complementary pair: you demonstrate using a **struct within a union**, and he demonstrates using a **union within a struct**. It turns out I need both at once: a **struct within a union within a struct** to implement some fancy message-passing polymorphism in C between threads on an embedded system, and I wouldn't have realized that had I not seen both your answers together. – Gabriel Staples Nov 21, 2018 at 0:58

---

1 I was wrong: it's a union within a struct within a union within a struct, nested left to write as I wrote that, from inner-most

nesting to outer-most level. I had to add another union at the inner-most level to allow values of different data types.
– Gabriel Staples Nov 21, 2018 at 1:19

2 Why do you start with `b1` and not `b0`? The problem is that there is no information about the order. In your example `b1` could be bit 0 or the highest bit (probably bit 7).
– 12431234123412341234123 Sep 7, 2020 at 17:03

1 @GabrielStaples Could you please post it as an answer with an example? I'm just curious. – Unknown123 Jul 27, 2022 at 12:58

73

Low level system programming is a reasonable example.

IIRC, I've used unions to breakdown hardware registers into the component bits. So, you can access an 8-bit register (as it was, in the day I did this ;-) into the component bits.

(I forget the exact syntax but...) This structure would allow a control register to be accessed as a control_byte or via the individual bits. It would be important to ensure the bits map on to the correct register bits for a given endianness.

```
typedef union {
    unsigned char control_byte;
    struct {
        unsigned int nibble  : 4;
        unsigned int nmi     : 1;
        unsigned int enabled : 1;
        unsigned int fired   : 1;
        unsigned int control : 1;
    };
} ControlRegister;
```

Share Improve this answer

Follow

edited Apr 3, 2014 at 23:11

ivan_pozdeev
35.8k ● 19 ● 113 ● 161

answered Aug 29, 2011 at 9:29

Snips
6,753 ● 8 ● 43 ● 66

4 This is an excellent example! Here is an example of how you could use this technique in embedded software: edn.com/design/integrated-circuit-design/4394915/…
– rzetterberg Dec 10, 2012 at 12:48

44

I've seen it in a couple of libraries as a replacement for object oriented inheritance.

E.g.

```
        Connection
     /       |       \
  Network   USB     VirtualConnection
```

If you want the Connection "class" to be either one of the above, you could write something like:

```
struct Connection
{
    int type;
    union
    {
        struct Network network;
        struct USB usb;
        struct Virtual virtual;
    }
};
```

Example use in libinfinity: http://git.0x539.de/?p=infinote.git;a=blob;f=libinfinity/common/inf-session.c;h=3e887f0d63bd754c6b5ec232948027cbbf4d61fc;hb=HEAD#l74

Share Improve this answer Follow

edited Nov 7 at 10:41
Arnau Sanz
100 ● 1 ● 7

answered Aug 29, 2011 at 9:24
bb-generation
1,527 ● 12 ● 10

Link above is rotten :/ Interesting example though ! – Orwellian Mentat May 26, 2022 at 9:12

@gordon_freeman well the answer was crafted 11 years prior so I suppose that is kind of expected. – mathreadler Jan 24 at 20:49

40

Unions allow data members which are mutually exclusive to share the same memory. This is quite important when memory is more scarce, such as in embedded systems.

In the following example:

```
union {
   int a;
   int b;
```

```
    int c;
} myUnion;
```

This union will take up the space of a single int, rather than 3 separate int values. If the user set the value of **a**, and then set the value of **b**, it would overwrite the value of **a** since they are both sharing the same memory location.

Share Improve this answer Follow

edited Oct 31, 2008 at 6:08

answered Oct 31, 2008 at 3:59

LeopardSkinPillBoxHat
29.4k ● 16 ● 79 ● 113

37

Lots of usages. Just do `grep union /usr/include/*` or in similar directories. Most of the cases the `union` is wrapped in a `struct` and one member of the struct tells which element in the union to access. For example checkout `man elf` for real life implementations.

This is the basic principle:

```
struct _mydata {
    int which_one;
    union _data {
            int a;
            float b;
            char c;
    } foo;
} bar;

switch (bar.which_one)
```

```
{
   case INTEGER  :  /* access bar.foo.a;*/ break;
   case FLOATING :  /* access bar.foo.b;*/ break;
   case CHARACTER:  /* access bar.foo.c;*/ break;
}
```

Share Improve this answer Follow

edited Jul 8, 2014 at 5:24

answered Aug 29, 2011 at 9:21

phoxis
61.8k ● 14 ● 83 ● 118

Exactly what I was looking for ! Very usefull to replace some ellipsis parameter :) – Nicolas Voron Jan 10, 2013 at 17:16

21

Here's an example of a union from my own codebase (from memory and paraphrased so it may not be exact). It was used to store language elements in an interpreter I built. For example, the following code:

```
set a to b times 7.
```

consists of the following language elements:

- symbol[set]
- variable[a]
- symbol[to]
- variable[b]

- symbol[times]
- constant[7]
- symbol[.]

Language elements were defines as '`#define`' values thus:

```
#define ELEM_SYM_SET        0
#define ELEM_SYM_TO         1
#define ELEM_SYM_TIMES      2
#define ELEM_SYM_FULLSTOP   3
#define ELEM_VARIABLE     100
#define ELEM_CONSTANT     101
```

and the following structure was used to store each element:

```
typedef struct {
    int typ;
    union {
        char *str;
        int   val;
    }
} tElem;
```

then the size of each element was the size of the maximum union (4 bytes for the typ and 4 bytes for the union, though those are typical values, the *actual* sizes depend on the implementation).

In order to create a "set" element, you would use:

```
tElem e;
e.typ = ELEM_SYM_SET;
```

In order to create a "variable[b]" element, you would use:

```
tElem e;
e.typ = ELEM_VARIABLE;
e.str = strdup ("b");   // make sure you free this lat
```

In order to create a "constant[7]" element, you would use:

```
tElem e;
e.typ = ELEM_CONSTANT;
e.val = 7;
```

and you could easily expand it to include floats (`float flt`) or rationals (`struct ratnl {int num; int denom;}`) and other types.

The basic premise is that the `str` and `val` are not contiguous in memory, they actually overlap, so it's a way of getting a different view on the same block of memory, illustrated here, where the structure is based at memory location `0x1010` and integers and pointers are both 4 bytes:

```
       +-----------+
0x1010 |           |
0x1011 |    typ    |
0x1012 |           |
0x1013 |           |
       +-----+-----+
0x1014 |     |     |
0x1015 | str | val |
0x1016 |     |     |
0x1017 |     |     |
       +-----+-----+
```

If it were just in a structure, it would look like this:

```
       +-------+
0x1010 |       |
0x1011 |  typ  |
0x1012 |       |
0x1013 |       |
       +-------+
0x1014 |       |
0x1015 |  str  |
0x1016 |       |
0x1017 |       |
       +-------+
0x1018 |       |
0x1019 |  val  |
0x101A |       |
0x101B |       |
       +-------+
```

Share Improve this answer Follow

edited Jun 26, 2020 at 14:50

jyz
6,179 ● 3 ● 30 ● 39

answered Oct 31, 2008 at 6:09

paxdiablo
880k ● 241 ● 1.6k ● 2k

Should the `make sure you free this later` comment be removed from the constant element? – Trevor Mar 6, 2013 at 12:21

2 Yes, @Trevor, though I can't believe you're the first person that saw it in the last 4+ years :-) Fixed, and thanks for that.

– paxdiablo Mar 6, 2013 at 12:38

9

I'd say it makes it easier to reuse memory that might be used in different ways, i.e. saving memory. E.g. you'd like to do some "variant" struct that's able to save a short string as well as a number:

```
struct variant {
    int type;
    double number;
    char *string;
};
```

In a 32 bit system this would result in at least 96 bits or 12 bytes being used for each instance of `variant`.

Using an union you can reduce the size down to 64 bits or 8 bytes:

```
struct variant {
    int type;
    union {
        double number;
        char *string;
    } value;
};
```

You're able to save even more if you'd like to add more different variable types etc. It might be true, that you can do similar things casting a void pointer - but the union makes it a lot more accessible as well as type safe. Such savings don't sound massive, but you're saving one third of the memory used for all instances of this struct.

Share Improve this answer Follow

answered Aug 29, 2011 at 9:21

Mario
36.4k ● 5 ● 67 ● 84

6

Many of these answers deal with casting from one type to another. I get the most use from unions with the same types just more of them (ie when parsing a serial data stream). They allow the parsing / construction of a *framed* packet to become trivial.

```
typedef union
{
    UINT8 buffer[PACKET_SIZE]; // Where the packet siz
                               // the entire set of fi
payload)

    struct
    {
        UINT8 size;
        UINT8 cmd;
        UINT8 payload[PAYLOAD_SIZE];
        UINT8 crc;
    } fields;

}PACKET_T;

// This should be called every time a new byte of data
// and point to the packet's buffer:
// packet_builder(packet.buffer, new_data);

void packet_builder(UINT8* buffer, UINT8 data)
{
    static UINT8 received_bytes = 0;
    // All range checking etc removed for brevity

    buffer[received_bytes] = data;
    received_bytes++;

    // Using the struc only way adds lots of logic tha
size
    // "byte 1" to cmd, etc...
}

void packet_handler(PACKET_T* packet)
{
    // Process the fields in a readable manner
    if(packet->fields.size > TOO_BIG)
    {
        // handle error...
    }

    if(packet->fields.cmd == CMD_X)
    {
        // do stuff..
    }
}
```

Edit The comment about endianness and struct padding are valid, and great, concerns. I have used this body of code almost entirely in embedded software, most of which I had control of both ends of the pipe.

Share Improve this answer Follow

edited Mar 17, 2014 at 12:45

answered Sep 27, 2013 at 16:34

Adam Lewis
7,227 ● 7 ● 47 ● 63

1 This code will not work(most of the times) if data is being exchanged across 2 dissimilar platforms because of following reasons: 1) Endianness may be different. 2) Padding in structures. – mahoriR Mar 16, 2014 at 16:36

@Ravi I agree with the concerns about endianness and padding. However it should be known that I have used this exclusively in embedded projects. Most of which I controlled both ends of the pipes. – Adam Lewis Mar 17, 2014 at 12:41

4

It's difficult to think of a specific occasion when you'd need this type of flexible structure, perhaps in a message protocol where you would be sending different sizes of messages, but even then there are probably better and more programmer friendly alternatives.

Unions are a bit like variant types in other languages - they can only hold one thing at a time, but that thing could be an int, a float etc. depending on how you declare it.

For example:

```
typedef union MyUnion MYUNION;
union MyUnion
{
   int MyInt;
   float MyFloat;
};
```

MyUnion will only contain an int OR a float, *depending on which you most recently set*. So doing this:

```
MYUNION u;
u.MyInt = 10;
```

u now holds an int equal to 10;

```
u.MyFloat = 1.0;
```

u now holds a float equal to 1.0. It no longer holds an int. Obviously now if you try and do printf("MyInt=%d", u.MyInt); then you're probably going to get an error, though I'm unsure of the specific behaviour.

The size of the union is dictated by the size of its largest field, in this case the float.

Share Improve this answer Follow

edited Oct 31, 2008 at 4:22

answered Oct 31, 2008 at 4:16
Xiaofu
15.8k ● 2 ● 34 ● 45

1 `sizeof(int) == sizeof(float)` (`== 32`) usually.
– Nick T Apr 16, 2013 at 19:54

2 For the record, assigning to the float then printing the int will *not* cause an error, as neither the compiler nor the run-time environment *know* which value is valid. The int that gets printed will, of course, be meaningless for most purposes. It will just be the memory representation of the float, interpreted as an int. – Jerry B Feb 13, 2014 at 8:38

4

union are used to save memory, especially used on devices with limited memory where memory is important. Exp:

```
union _Union{
  int a;
  double b;
  char c;
};
```

For example,let's say we need the above 3 data types(int,double,char) in a system where memory is limited.If we don't use "union",we need to define these 3 data types. In this case sizeof(a) + sizeof(b) + sizeof(c) memory space will be allocated.But if we use onion,only one memory space will be allocated according to the largest data t ype in these 3 data types.Because all variables in union structure will use the same memory space. Hence the memory space allocated accroding to the largest data type will be common space for all variables. For example:

```
union _Union{
int a;
double b;
char c;
};

int main() {
 union _Union uni;
 uni.a = 44;
 uni.b = 144.5;
 printf("a:%d\n",uni.a);
 printf("b:%lf\n",uni.b);
 return 0;
 }
```

Output is: a: 0 and b:144.500000

Why a is zero?. Because union structure has only one memory area and all data structures use it in common. So the last assigned value overwrites the old one. One more example:

```
 union _Union{
    char name[15];
    int id;
};


int main(){
   union _Union uni;
   char choice;
   printf("YOu can enter name or id value.");
   printf("Do you want to enter the name(y or n):");
   scanf("%c",&choice);
   if(choice == 'Y' || choice == 'y'){
     printf("Enter name:");
     scanf("%s",uni.name);
     printf("\nName:%s",uni.name);
   }else{
     printf("Enter Id:");
     scanf("%d",&uni.id);
     printf("\nId:%d",uni.id);
   }
return 0;
}
```

Note:Size of the union is the size of its largest field because sufficient number of bytes must be reserved to store the larges sized field.

Share Improve this answer

Follow

answered Nov 13, 2021 at 15:39

kfc
607 ● 1 ● 6 ● 11

3

Unions are used when you want to model structs defined by hardware, devices or network protocols, or when you're creating a large number of objects and want to save space. You really don't need them 95% of the time though, stick with easy-to-debug code.

Share Improve this answer

Follow

answered Oct 31, 2008 at 3:59

Ana Betts
74.6k ● 16 ● 143 ● 210

3

In school, I used unions like this:

```
typedef union
{
  unsigned char color[4];
  int       new_color;
}       u_color;
```

I used it to handle colors more easily, instead of using >> and << operators, I just had to go through the different index of my char array.

Share Improve this answer

Follow

answered Aug 29, 2011 at 9:23

Zoneur
1,099 ● 9 ● 20

2

In early versions of C, all structure declarations would share a common set of fields. Given:

```
struct x {int x_mode; int q; float x_f};
struct y {int y_mode; int q; int y_l};
struct z {int z_mode; char name[20];};
```

a compiler would essentially produce a table of structures' sizes (and possibly alignments), and a separate table of structures' members' names, types, and offsets. The compiler didn't keep track of which members belonged to which structures, and would allow two structures to have a member with the same name only if the type and offset matched (as with member `q` of `struct x` and `struct y`). If p was a pointer to any structure type, p->q would add the offset of "q" to pointer p and fetch an "int" from the resulting address.

Given the above semantics, it was possible to write a function that could perform some useful operations on multiple kinds of structure interchangeably, provided that all the fields used by the function lined up with useful fields within the structures in question. This was a useful feature, and changing C to validate members used for structure access against the types of the structures in question would have meant losing it in the absence of a means of having a structure that can contain multiple named fields at the same address. Adding "union" types to C helped fill that gap somewhat (though not, IMHO, as well as it should have been).

An essential part of unions' ability to fill that gap was the fact that a pointer to a union member could be converted into a pointer to any union containing that member, and a pointer to any union could be converted to a pointer to any member. While the C89 Standard didn't expressly say that casting a `T*` directly to a `U*` was equivalent to casting it to a pointer to any union type containing both `T` and `U`, and then casting that to `U*`, no defined behavior of the latter cast sequence would be affected by the union type used, and the Standard didn't specify any contrary semantics for a direct cast from `T` to `U`. Further, in cases where a function received a pointer of unknown origin, the behavior of writing an object via `T*`, converting the `T*` to a `U*`, and then reading the object via `U*` would be equivalent to writing a union via member of type `T` and reading as type `U`, which would be standard-defined in a few cases (e.g. when accessing Common Initial Sequence members) and Implementation-Defined (rather than Undefined) for the rest. While it was rare for programs to exploit the CIS guarantees with actual objects of union type, it was far more common to exploit the fact that pointers to objects of unknown origin had to behave like pointers to union members and have the behavioral guarantees associated therewith.

Share Improve this answer Follow

answered Apr 20, 2017 at 18:21

supercat
80.8k ● 9 ● 174 ● 220

---

can you give an example of this : ` it was possible to write a function that could perform some useful operations on multiple kinds of structure interchangeably`. How could be used multiple structures member with the same name? If two sctructures has same data alignment and thus a member with the same name and same offset as in you example, then from which structure would I yield the actual data? (value). Two structure has same alignment and same members, but different values on them. Can you please elaborate it – Herdsman May 3, 2020 at 17:54

@Herdsman: In early versions of C, a struct member name encapsulated a type and an offset. Two members of different structures could have the same name if and only if their types and offsets matched. If struct member `foo` is an `int` with offset 8, then `anyPointer->foo = 1234;` meant "take the address in anyPointer, displace it by 8 bytes, and perform an integer store of the value 1234 to the resulting address. The compiler wouldn't need to know or care whether `anyPointer` identified any structure type which had `foo` listed among its members. – supercat May 3, 2020 at 18:39

With pointer you can dereference any adress regardless the 'origin' of the pointer, that is true, but then what is the point of compiler to holds tables of structures members and their names (as you said in your post) if I can fetch data with any pointer just knowing the address of a member in a particular structure? And if compiler does not know whether the `anyPointer` indentifies with a struct member, then how will compiler checks these condition `to have a member with the same name only if the type and offset matched` of your post? – Herdsman May 3, 2020 at 19:16

@Herdsman: The compiler would keep the list of structure members' names because the precise behavior of `p->foo` would depend upon the type and offset of `foo`. Essentially, `p->foo` was shorthand for `*(typeOfFoo*)((unsigned char*)p + offsetOfFoo)`. As for your latter question, when a compiler encounters a struct member definition, it requires that either no member with that name exists, or that the member with that name have the same type and offset; I would guess that the would have squawked if a non-matching struct member definition existed, but I don't know how it handled errors. – supercat May 3, 2020 at 20:32

1

Unions are great. One clever use of unions I've seen is to use them when defining an event. For example, you might decide that an event is 32 bits.

Now, within that 32 bits, you might like to designate the first 8 bits as for an identifier of the sender of the event... Sometimes you deal with the event as a whole, sometimes you dissect it and compare it's components. unions give you the flexibility to do both.

```
union Event
{
  unsigned long eventCode;
  unsigned char eventParts[4];
};
```

Share Improve this answer Follow

answered Oct 31, 2008 at 4:36

dicroce
46.7k ● 31 ● 105 ● 149

1

What about `VARIANT` that is used in COM interfaces? It has two fields - "type" and a union holding an actual value that is treated depending on "type" field.

answered Aug 29, 2011 at 9:16

Share Improve this answer

Follow

sharptooth
170k ● 108 ● 535 ● 1k

0

I used union when I was coding for embedded devices. I have C int that is 16 bit long. And I need to retrieve the higher 8 bits and the lower 8 bits when I need to read from/store to EEPROM. So I used this way:

```
union data {
    int data;
    struct {
        unsigned char higher;
        unsigned char lower;
    } parts;
};
```

It doesn't require shifting so the code is easier to read.

On the other hand, I saw some old C++ stl code that used union for stl allocator. If you are interested, you can read the sgi stl source code. Here is a piece of it:

```
union _Obj {
    union _Obj* _M_free_list_link;
    char _M_client_data[1];    /* The client sees this
};
```

Share Improve this answer

Follow

edited Aug 29, 2011 at 9:26

answered Aug 29, 2011 at 9:20

Mu Qiao
7,077 ● 1 ● 32 ● 35

1 Wouldn't you need a grouping `struct` around your `higher` / `lower` ? Right now both should point to the first byte only. – Mario Aug 29, 2011 at 9:25

@Mario ah right, I just write it by hand and forget about it, thanks – Mu Qiao Aug 29, 2011 at 9:26

0

- A file containing different record types.
- A network interface containing different request types.

Take a look at this: [X.25 buffer command handling]

One of the many possible X.25 commands is received into a buffer and handled in place by using a UNION of all the possible structures.

Share Improve this answer Follow

edited Apr 25, 2013 at 20:58
Matthew Read
1,834 ● 2 ● 34 ● 53

answered Aug 29, 2011 at 9:20
James Anderson
27.4k ● 7 ● 54 ● 80

could you please explain both of these examples .I mean how these are related to union – Amit Singh Tomar Aug 29, 2011 at 9:22

0

A simple and very usefull example, is....

Imagine:

you have a `uint32_t array[2]` and want to access the 3rd and 4th Byte of the Byte chain. you could do `*((uint16_t*) &array[1])`. But this sadly breaks the strict aliasing rules!

But known compilers allow you to do the following :

```
union un
{
    uint16_t array16[4];
    uint32_t array32[2];
}
```

technically this is still a violation of the rules. but all known standards support this usage.

Share Improve this answer Follow

answered Sep 18, 2013 at 15:07

dhein
6,561 ● 5 ● 44 ● 79

0

I found `union` to come useful for determining the size of the largest constant.

```
#include <stdio.h>
#include <time.h>

#define MON_1 "January"
#define MON_2 "February"
#define MON_3 "March"
```

```
#define MON_4 "April"
#define MON_5 "May"
#define MON_6 "June"
#define MON_7 "July"
#define MON_8 "August"
#define MON_9 "September"
#define MON_10 "October"
#define MON_11 "November"
#define MON_12 "December"
#define WDAY_1 "Sunday"
#define WDAY_2 "Monday"
#define WDAY_3 "Tuesday"
#define WDAY_4 "Wednesday"
#define WDAY_5 "Thursday"
#define WDAY_6 "Friday"
#define WDAY_7 "Saturday"

int main(void)
{
  time_t t = time(NULL);
  struct tm *t_tm = gmtime(&t);
  char *t_wday[] = {
    WDAY_1,
    WDAY_2,
    WDAY_3,
    WDAY_4,
    WDAY_5,
    WDAY_6,
    WDAY_7
  },
  *t_mon[] = {
    MON_1,
    MON_2,
    MON_3,
    MON_4,
    MON_5,
    MON_6,
    MON_7,
    MON_8,
    MON_9,
    MON_10,
    MON_11,
    MON_12
  },
  t_arr[sizeof(union {
    char mon[sizeof MON_1],
    mon2[sizeof MON_2],
    mon3[sizeof MON_3],
    mon4[sizeof MON_4],
    mon5[sizeof MON_5],
    mon6[sizeof MON_6],
    mon7[sizeof MON_7],
    mon8[sizeof MON_8],
    mon9[sizeof MON_9],
    mon10[sizeof MON_10],
    mon11[sizeof MON_11],
    mon12[sizeof MON_12];
  }) + sizeof(union {
    char wday[sizeof WDAY_1],
    wday2[sizeof WDAY_2],
    wday3[sizeof WDAY_3],
    wday4[sizeof WDAY_4],
    wday5[sizeof WDAY_5],
    wday6[sizeof WDAY_6],
    wday7[sizeof WDAY_7];
  }) + sizeof "00 00:00:00 0000"];

  sprintf(t_arr, "%s %s %d %d:%d:%d %d", t_wday[t_tm->
>tm_mon], t_tm->tm_mday, t_tm->tm_hour, t_tm->tm_min,
>tm_year + 1900);
  printf("%s\n", t_arr);
}
```

This avoids the risk on defining improper array sizes by hand.

Share Improve this answer Follow

answered Sep 22, 2023 at 3:49

Matheus Garcia
41 ● 2

Use a union when you have some function where you return a value that can be different depending on what the

-1 function did.

Share Improve this answer

Follow

answered Feb 7, 2021 at 8:02

Tejaswini

29 ● 5