Introduction to Programming

Part I

Lecture 4
The Basics of C: structs, unions, enumerations

Eugene Zouev
Fall Semester 2021
Innopolis University

What We Have Considered Before:

- The memory model: code, heap & stack.
- The typical C program structure.
- Variable scopes and program blocks.
- The notion of **type**. Static and dynamic typing. Type categories. The C type system.
- Storage class specifiers: `auto`, `static`, `extern`
- Pointers & arrays
- Statements & expressions
- Dynamic memory management

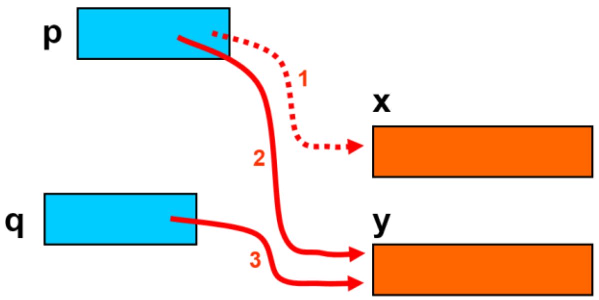
Some Key Points From the Previous Lecture

Pointers

1. **Pointer**:
   An object containing an address to some other object

```
int x;
int* p;
...
p = &x;   1
```

Unary "address-of" operator




```
int y;
...
p = &y; 2
```

```
int* q;
...
q = p;   3
```

20/27

From previous lectures

Some Key Points From the Previous Lecture

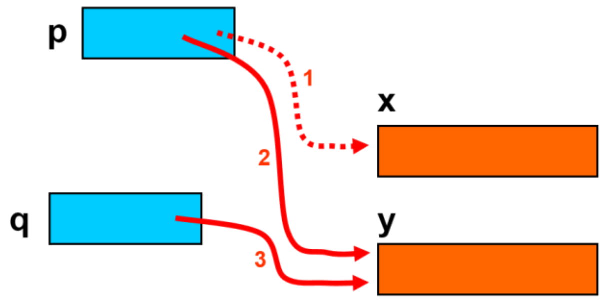
Pointers

1. Pointer:

<u>An object</u> containing an address to some other object

```
int x;
int* p;
...
p = &x;   1
```

Unary "address-of" operator




```
int y;
...
p = &y; 2
```

```
int* q;
...
q = p;  3
```

From previous lectures

Pointers

3. Operators on pointers

&object

Taking address of object

Unary prefix operator

```
int x;
int* p;
...
p = &x;
```

***pointer**

Dereferencing: Getting object pointed to by "pointer"

Unary prefix operator

```
int x;
int* p = &x;
...
*p = 777;     // x is 777
int z = *p+1; // z is 778
```

Some Key Points From the Previous Lecture

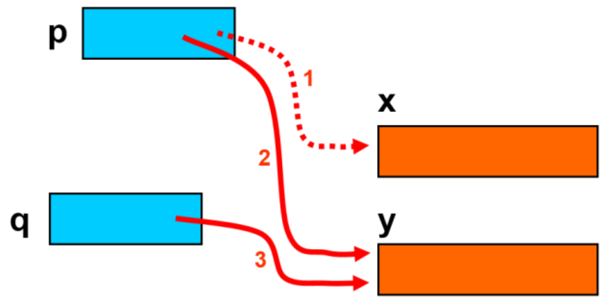
Pointers

1. Pointer:

<u>An object</u> containing an address to some other object

```
int x;
int* p;
...
p = &x;   1
```

Unary "address-of" operator




```
int y;
...
p = &y; 2
```

```
int* q;
...
q = p;  3
```

```
pointer+i
pointer-i
pointer++
pointer--
ptr1-ptr2
```

Operators defined on pointers

Pointers

3. Operators on pointers

&object Taking address of object

Unary prefix operator

```
int x;
int* p;
...
p = &x;
```

***pointer** Dereferencing: Getting object pointed to by "pointer"

Unary prefix operator

```
int x;
int* p = &x;
...
*p = 777;      // x is 777
int z = *p+1; // z is 778
```

22/27

From previous lectures

Dynamic Objects

From previous lectures

How dynamic objects are created (and destroyed)?

- Using special standard functions from the C library

```
void* malloc ( int size )
{
   ...
   Allocation algorithm
   ...
}
```

```
void free ( void* ptr )
{
   ...
   Deallocation algorithm
   ...
}
```

- Specification is a bit simplified.
- The function allocates space for an object whose size (in bytes) is passed via the parameter.
- The function returns a pointer to the memory allocated.
- The pointer is "untyped" (`void*`).
- There are more allocation functions in the library.

Library Organization

From previous lectures

Each translation unit is usually represented **by two source files**:
- with forward declarations ("interface");
- with full declarations ("implementation").

To remind...

```
void* malloc(int size);
void free(void* ptr);
...
And many other function
headers ("prototypes")
...
```

stdlib.h

```
void* malloc(int size)
{
    ...
    Implementation
    ...
}
...
And implementations
of many other standard
functions
...
```

stdlib.c

Precompiled

Dynamic Objects

From previous lectures

How dynamic objects are created?

- Using special standard functions from the C library

Example

```
#include <stdlib.h>

struct S { int a, b; };

void* ptr = malloc(sizeof(struct S));

struct S* s = (struct S*)ptr;

s->a = 5;
...
```

In order to use `malloc`, we should add its header

This is struct type declaration

Here, we dynamically allocate memory suitable to keep objects of type `struct S`…

…and **convert** the void pointer type to the type of pointer to `struct S`.

After that, we can use `s` to get access to elements of `struct S`.

Expressions in C

"Expression" is a formula for calculating values.

- Any expression (almost any ☺) issues a value.

In general, expressions are built of

- Operands
- Operators
- Parentheses

using ordinary rules (as in many other programming languages).

```
f()*(a+b)-*p++;
```

Expressions in C

Primary expression elements:

- Identifier (designates a variable/constant/function)

```
fun   abs   ptr_fun
```

Identifiers designate corresponding entities:
Either values of variables/constants or function addresses

- Literal: integer/floating/string

```
123   0xFE   0.01E-2   "string"
```

Literals designate themselves

- Subexpression enclosed in parentheses

```
(a-b)
```

Subexpressions designate values of enclosed expressions.

Expressions in C

Secondary expression elements ("postfix expressions") - are built on top of primary expressions:

- Array subscripting

```
arr[i+j*2]
```

Value of or reference to an array element.

- Function call

```
fun(*p,&x,777+y)
```

The result of the function call.

- Structure/union member access

```
ptr->m    s.m
```

Value of or pointer to a struct member.

- Postfix decrement/increment

```
ptr--    arr++
```

The result is the initial pointer (**YES**!)
The side effect: the pointer gets moved to the previous/next element depending of the type pointer to by the pointer

Expressions in C

Next (higher-level) building blocks: unary expressions
- are built on top of postfix expressions:

- Prefix increment/decrement

```
--p    ++x
```

Result: the value of the operand increased or decreased by one.

- Address & indirection

```
&x    *(p+1)
```

Result: the address of the operand OR the value pointed to by the pointer from the operand.

- Unary plus/minus

```
+x    -v
```

The value from the operand (not changed) OR the original value with the inverted sign.

- Bitwise complement & logical negation

```
~v    !v
```

The result: the initial value inverted or negated

- Sizeof operator

```
sizeof (T)   sizeof a+b
```

The result: an integer value

Expressions in C

The highest-level building blocks for expressions:
binary expressions:

- Additive & multiplicative operators

```
a+b     b-c     c*d     d/e     e%f
```

- Relational & equality operators

```
a<b     a<=b     a>b     a>=b     a==b     a!=b
```

- Bitwise shift operators

```
a << b      a >> b
```

- Bitwise logical operators

```
a & b      a | b      a ^ b
```

- Logical operators

```
a && b      a || b
```

Expressions in C

These are also binary operators:

- Assignment operators

```
a = b
```

```
a+=b   a-=b   a*=b   a/=b   a%=b
a <<= b   a >>= b
a &= b   a ^= b   a |= b
```

- **Comma operator (!!)**

```
expr1 , expr2
```

The left expression is evaluated; its value **is discarded**. Then the second expression is evaluated. Its value is the result of the whole comma expression.

- Conditional operator

```
expression ? expression : expression
```

The single **ternary** operator in the language

Expressions in C

Basic rules for expressions

- Unary operators are performed **from right to left**.

```
&*p    ~-v    *f()
```

- Binary operators are performed **in accordance with their preferences**.

```
a[i] + b * *p
```

- Binary operators of the same preference are performed **from left to right**.

```
x + y - z
```

```
a[i] = b = c + d*e
```

Is it aways true? – check by your own!

- The **side effect** of the expression (if any) happens after both operands are evaluated.

```
a[i++] = i
```

- Parentheses are used to change the default execution order.

```
(a[i] + b) * *p
```

Expressions in C

Some examples for the comma operator

```
if ( f(b),g(c) )
    ...
else
    ...
```

```
for ( int i=0, j=0; i<10 && j<10; i++, j++)
{
  Some calculations on a matrix…
}
```

Outline: Today

- Structures
- Bit-fields
- Alignment
- Unions
- Enumerations

Structures: How to Declare?

ISO C Standard, 6.7.2.1, §6

…A structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence

Type tag

```c
struct S
{
  int a, b;
  double c;
  int* p;
};
```

Member declarations

Structures: How to Declare?

ISO C Standard, 6.7.2.1, §6

…A structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence

Type tag

```
struct S
{
  int a, b;
  double c;
  int* p;
};
```

Member declarations

Two ways (as usual for C):

- **Static**
- **Dynamic**

```
struct S s1;
```

s1 is the object of type struct S created **in the stack** and accessible by its name from within a local scope or in the global scope.

Structures: How to Declare?

ISO C Standard, 6.7.2.1, §6

…A structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence

Type tag

```
struct S
{
   int a, b;
   double c;
   int* p;
};
```

Member declarations

Two ways (as usual for C):

- **Static**
- **Dynamic**

```
struct S s1;
```

s1 is the object of type **struct** S created **in the stack** and accessible by its name from within a local scope or in the global scope.

```
struct S* ps =
     (struct S*)malloc(sizeof(struct S));
```

ps points to the dynamic object of type **struct** S created **in the heap** and accessible via pointer.

Structures: How to Use

```
struct S
{
  int a, b;
  double c;
  int* p;
};
```

Two ways of accessing <u>structure elements</u>:

- Via name
- Via pointer

Structures: How to Use

```
struct S
{
  int a, b;
  double c;
  int* p;
};
```

Two ways of accessing structure elements:

- Via name
- Via pointer

Via name: **dot notation**

struct-name . member-name

```
struct S s1;
...
s1.a = 777;
s1.b = (int)s1.p;
...
```

Structures: How to Use

```
struct S
{
  int a, b;
  double c;
  int* p;
};
```

Two ways of accessing structure elements:

- Via name
- Via pointer

Via name: **dot notation**

struct-name . member-name

```
struct S s1;
...
s1.a = 777;
s1.b = (int)s1.p;
...
```

Via pointer: **arrow notation**

pointer-to-struct -> member-name

```
struct S* ps =
     (struct S*)malloc(sizeof(struct S));
...
ps->a = 777;
ps->b = (int)ps->p;
...
```

Structures: How to Initialize?

```
struct SheetOfPaper
{
  int height;
  int width;
};
```

```
struct S
{
  int a, b;
  double c;
  int* p;
};
```

Structures: How to Initialize?

```
struct SheetOfPaper
{
  int height;
  int width;
};
```

Usual initialization

```
struct SheetOfPaper letter;
letter.height = 279;
letter.width = 216;
```

```
struct S
{
  int a, b;
  double c;
  int* p;
};
```

Usual initialization

```
struct S my1;
my1.a = 1;
my1.b = 2;
my1.c = 0.3;
my1.p = NULL;
```

Structures: How to Initialize?

```c
struct SheetOfPaper
{
  int height;
  int width;
};
```

Usual initialization

```c
struct SheetOfPaper letter;
letter.height = 279;
letter.width = 216;
```

Advanced syntax

```c
struct SheetOfPaper A4 =
   { .height = 210, .width = 297 };
```

```c
struct S
{
  int a, b;
  double c;
  int* p;
};
```

Usual initialization

```c
struct S my1;
my1.a = 1;
my1.b = 2;
my1.c = 0.3;
my1.p = NULL;
```

Advanced syntax

```c
struct S my2 =
  { .a = 1, .b = 2, .c = 0.3, .p = NULL };
```

Structures: How to Initialize?

A bit more complicated example

- What's declared here?

```
struct { int a[3], b; } w[]
```

Structures: How to Initialize?

A bit more complicated example

- What's declared here? – **an array** `w`

```
struct { int a[3], b; } w[]
```

Structures: How to Initialize?

A bit more complicated example

- What's declared here? – **an array `w`**
- What's the size of the array `w`? – **not specified!**

```
struct { int a[3], b; } w[]
```

Structures: How to Initialize?

A bit more complicated example

- What's declared here? – **an array `w`**
- What's the size of the array `w`? – **not specified!**
- What's the type of array elements? – **a structure**
- What's the name of this structure? – **it's unnamed**

Unnamed structure

```
struct { int a[3], b; } w[]
```

Structures: How to Initialize?

A bit more complicated example

- What's declared here? – **an array** `w`
- What's the size of the array `w`? – **not specified!**
- What's the type of array elements? – **a structure**
- What's the name of this structure? – **it's unnamed**
- What are structure elements? – **array and a single integer**

Unnamed structure

```
struct { int a[3], b; } w[]
```

Structures: How to Initialize?

A bit more complicated example

- What's declared here? – **an array** `w`
- What's the size of the array `w`? – **not specified!**
- What's the type of array elements? – **a structure**
- What's the name of this structure? – **it's unnamed**
- What are structure elements? – **array and a single integer**

Unnamed structure

```
struct { int a[3], b; } w[] =
                { [0].a = {1}, [1].a[0] = 2 };
```

- Why the size of `w` is not specified? – **the real size is extracted from the initializer**.
- What's initialized? – **the array from the 0^{th} element of** `w` **and the 0^{th} element of the 1^{st} element of** `w`

Structures: How to Initialize?

Experiment on various forms of initializations at home!

A bit more complicated example

- What's declared here? – **an array** `w`
- What's the size of the array `w`? – **not specified!**
- What's the type of array elements? – **a structure**
- What's the name of this structure? – **it's unnamed**
- What are structure elements? – **array and a single integer**

Unnamed structure

```
struct { int a[3], b; } w[] =
                    { [0].a = {1}, [1].a[0] = 2 };
```

- Why the size of `w` is not specified? – **the real size is extracted from the initializer**.
- What's initialized? – **the array from the 0^{th} element of** `w` **and the 0^{th} element of the 1^{st} element of** `w`

Arrays: How to Initialize?

```
int x[] = { 1, 3, 5 };
```

The size of the array is not specified – it's extracted from the initializer.

Arrays: How to Initialize?

```
int x[] = { 1, 3, 5 };
```

The size of the array is not specified – it's extracted from the initializer.

BTW, what's an extra advantage of the shorter form comparatively with assignments?

```
int x[3];
x[0] = 1;
x[1] = 3;
x[2] = 5;
```

Arrays: How to Initialize?

```
int x[] = { 1, 3, 5 };
```

The size of the array is not specified – it's extracted from the initializer.

BTW, what's an extra advantage of the shorter form comparatively with assignments?

```
int x[3];
x[0] = 1;
x[1] = 3;
x[2] = 5;
```

Incomplete type

```
typedef int A[];
```

A is the <u>synonym</u> for "array of unspecified size containing integers"

See tutorial for typedef's semantics

Arrays: How to Initialize?

```
int x[] = { 1, 3, 5 };
```

The size of the array is not specified – it's extracted from the initializer.

BTW, what's an extra advantage of the shorter form comparatively with assignments?

```
int x[3];
x[0] = 1;
x[1] = 3;
x[2] = 5;
```

Incomplete type

```
typedef int A[];
```

A is the synonym for "array of unspecified size containing integers"

See tutorial for typedef's semantics

```
A a = { 1, 2 },
  b = { 3, 4, 5 };
```

Array declarations with initialization

Arrays: How to Initialize?

```
int x[] = { 1, 3, 5 };
```

The size of the array is not specified – it's extracted from the initializer.

BTW, what's an extra advantage of the shorter form comparatively with assignments?

```
int x[3];
x[0] = 1;
x[1] = 3;
x[2] = 5;
```

Incomplete type

```
typedef int A[];
```

A is the synonym for "array of unspecified size containing integers"

See tutorial for typedef's semantics

```
A a = { 1, 2 },
  b = { 3, 4, 5 };
```

Array declarations with initialization

```
int a[] = { 1, 2 },
    b[] = { 3, 4, 5 };
```

These forms are semantically the same

Nested Structures: Examples

```c
struct Person
{
  char* name;
  struct { int unique_num, salary; } personal_info;
  int* extra_info;
};
```

```c
struct Person john;
...
john.name = "John";
john.personal_info.unique_num = 12345678;
...
struct Person* p = &john;
...
p->personal_info.salary += 100;
```

Structures: Alignment

```
struct S
{
  char* m1;
  short m2[3];
  long m3;
};
```

```
...
struct S s;
...
```

Structures: Alignment

```
struct S
{
  char* m1;
  short m2[3];
  long m3;
};
```

```
...
struct S s;
...
```

What about the following equation:

```
sizeof(s) == ?
   sizeof(s.m1) + sizeof(s.m2) + sizeof(s.m3);
```

Structures: Alignment

```
struct S
{
  char* m1;
  short m2[3];
  long m3;
};
```

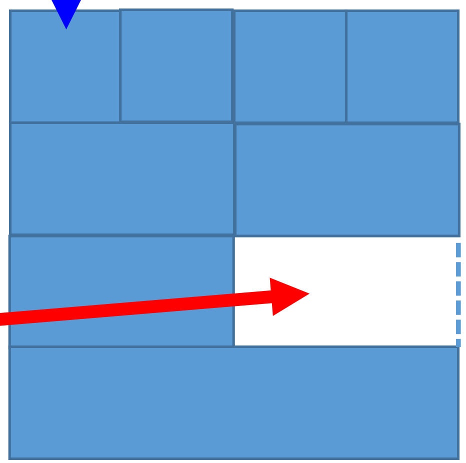
```
...
struct S s;
...
```

What about the following equation:

sizeof(s) == ?
  sizeof(s.m1) + sizeof(s.m2) + sizeof(s.m3);

Addressable bytes

S internal layout:



This memory is not used!

m1 is pointer to char: **4 bytes**

m[0], m[1] are shorts; **2 bytes** each

m[2] is short: **2 bytes**

m3 is long: **4 bytes**

Structures: Bit-fields

ISO C Standard, 6.7.2.1, §9-11

…A member may be declared to consist of a **specified number of bits** (including a sign bit, if any). Such a member is called a *bit-field.*

A bit-field is interpreted as having **a signed or unsigned integer type** consisting of the specified number of bits

An implementation may allocate **any addressable storage unit** large enough to hold a bit-field.

```c
struct S
{
  short m1[3];
  int m2:5;
  long m3;
};
```

Structures: Bit-fields

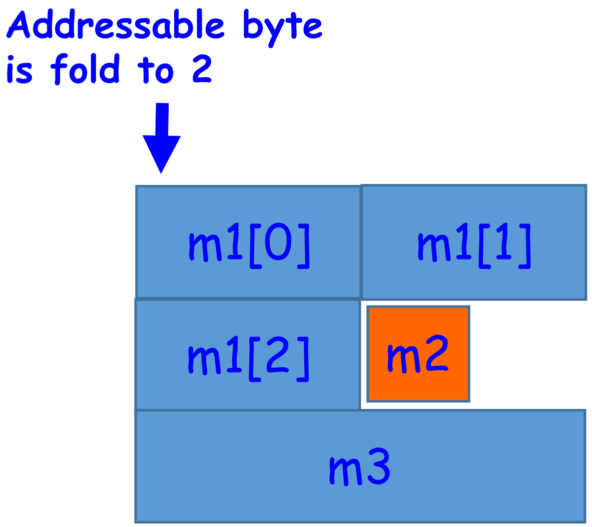
ISO C Standard, 6.7.2.1, §9-11

…A member may be declared to consist of a **specified number of bits** (including a sign bit, if any). Such a member is called a *bit-field.*

A bit-field is interpreted as having **a signed or unsigned integer type** consisting of the specified number of bits

An implementation may allocate **any addressable storage unit** large enough to hold a bit-field.

```
struct S
{
  short m1[3];
  int m2:5;
  long m3;
};
```



Structures: Bit-fields

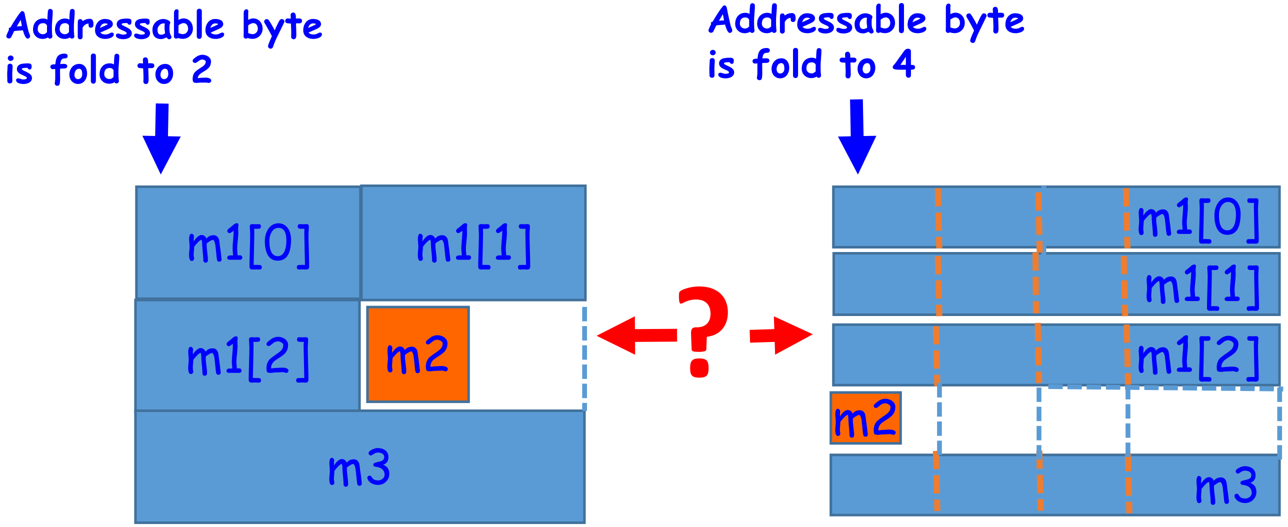
ISO C Standard, 6.7.2.1, §9-11

…A member may be declared to consist of a **specified number of bits** (including a sign bit, if any). Such a member is called a *bit-field.*

A bit-field is interpreted as having **a signed or unsigned integer type** consisting of the specified number of bits

An implementation may allocate **any addressable storage unit** large enough to hold a bit-field.

```
struct S
{
   short m1[3];
   int m2:5;
   long m3;
};
```



Which layout is used?
- Implementation-defined!

Structures: Bit-fields

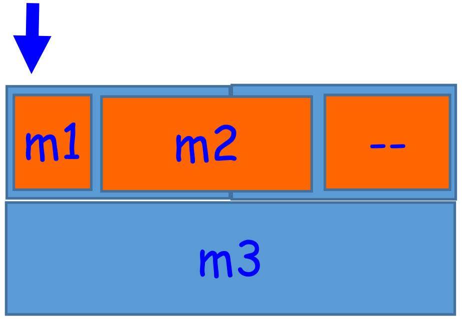
ISO C Standard, 6.7.2.1, §11

…If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed **into adjacent bits of the same unit**.

```
struct MyLayout
{
  unsigned int m1:2;
  unsigned int m2:10;
  unsigned int   :4;
  long m3;
};
```

Unnamed bit-field

Addressable byte is fold to 4



Unions

ISO C Standard, 6.7.2.1, §6

…**Union** is a type consisting of a sequence of members whose storage **overlap**.

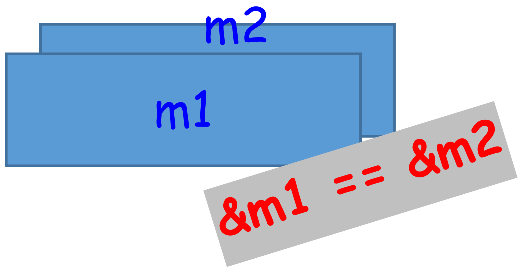
```
union U
{
  unsigned int m1;
  int*         m2;
};
```

Unions

ISO C Standard, 6.7.2.1, §6

…**Union** is a type consisting of a sequence of members whose storage **overlap**.

```
union U
{
  unsigned int m1;
  int*         m2;
};
```



Both m1 and m2 are in the same memory.

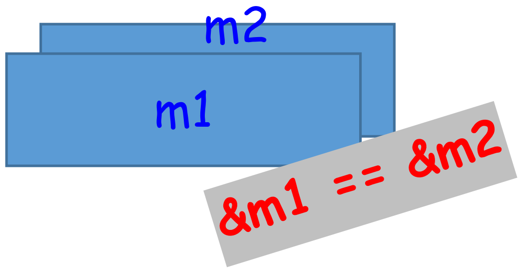
sizeof(U) == max(**sizeof**(m1), **sizeof**(m2))

Unions

ISO C Standard, 6.7.2.1, §6

…**Union** is a type consisting of a sequence of members whose storage **overlap**.

```c
union U
{
  unsigned int m1;
  int*         m2;
};
```



Both m1 and m2 are in the same memory.

sizeof(U) == max(**sizeof**(m1), **sizeof**(m2))

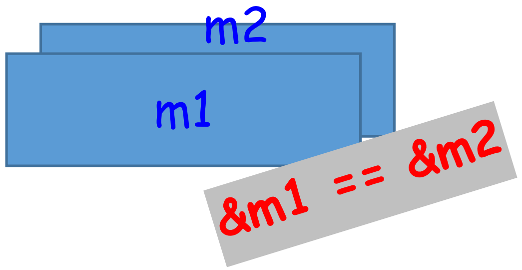
```c
int x;
union U u;
...
u.m2 = &x;
...
unsigned y = u.m1;
```

Unions

ISO C Standard, 6.7.2.1, §6

…**Union** is a type consisting of a sequence of members whose storage **overlap**.

```
union U
{
   unsigned int m1;
   int*         m2;
};
```



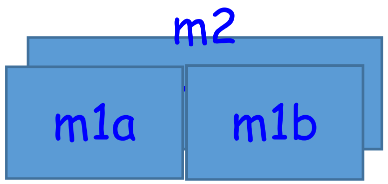
Both m1 and m2 are in the same memory.

sizeof(U) == max(**sizeof**(m1), **sizeof**(m2))

```
int x;
union U u;
...
u.m2 = &x;
...
unsigned y = u.m1;
```

Two members of the union; each declaration is considered as a member

```
union U1
{
   int m1a, m1b;
   int* m2;
};
```



Enumerations

An example:

светофор

Suppose we are going to control the traffic lights with three states: **red**, **yellow** and **green**.
How do we do that?

Enumerations

An example:

светофор

Suppose we are going to control the traffic lights with three states: **red**, **yellow** and **green**.
How do we do that?

Conventional solution

```
const int green  = 0;
const int yellow = 1;
const int red    = 2;

int tl;
...
```

This variable serves as a model of a traffic lights

Enumerations

An example:
Suppose we are going to control the traffic lights with three states: **red**, **yellow** and **green**.
How do we do that?

светофор

Conventional solution

```
const int green  = 0;
const int yellow = 1;
const int red    = 2;

int tl;
...
tl = 777;
```

Why these numbers?
Why not 4, 12, 78?

This variable serves as a model of a traffic lights

What happens if we write this ?

Enumerations

An example:

светофор

Suppose we are going to control the traffic lights with three states: **red**, **yellow** and **green**.
How do we do that?

Conventional solution

```
const int green  = 0;
const int yellow = 1;
const int red    = 2;

int tl;
...
tl = 777;
```

Why these numbers?
Why not 4, 12, 78?

This variable serves as a model of a traffic lights

What happens if we write this ?

Advanced solution

This is **enumeration type**!

```
enum Lights {
    green,
    yellow,
    red
};
...
Lights tl;
...
tl = 777; // ERROR
```

These are **enumerators**

Enumerations

An example:

светофор

Suppose we are going to control the traffic lights with three states: **red**, **yellow** and **green**.
How do we do that?

Conventional solution

```
const int green  = 0;
const int yellow = 1;
const int red    = 2;

int tl;
...
tl = 777;
```

Why these numbers?
Why not 4, 12, 78?

This variable serves as a model of a traffic lights

What happens if we write this ?

Advanced solution

This is **enumeration type**!

```
enum Lights {
    green,
    yellow,
    red
};
...
Lights tl;
...
tl = 777; // ERROR
```

These are **enumerators**

In general, we are not interested in actual values behind green, yellow & red!

However...
"Behind the scenes", the enumerator values are just integers, starting from 0.

Summary

- **Structs**
  are used for constructing complex data structures with heterogeneous elements
- **Bit-fields**
  in structs are used to define elements of arbitrary sizes
- **Unions**
  are low-level means for controlling the interpretation of values
- **Enumerations**
  are used for symbolic naming of objects when their values are not needed.