Programming Software Systems

Introduction to Programming for the Computer Engineering Track

Lecture 4 + Tutorial 4
The Basics of C

Eugene Zouev
Fall Semester 2020
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

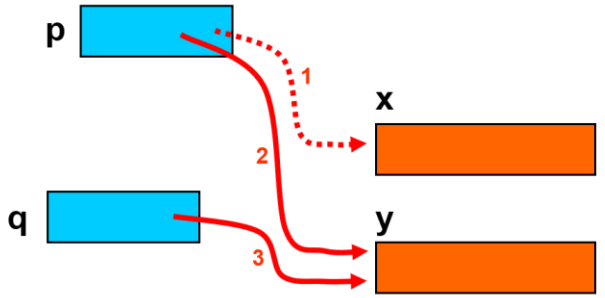
Some Key Points From the Previous Lecture

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

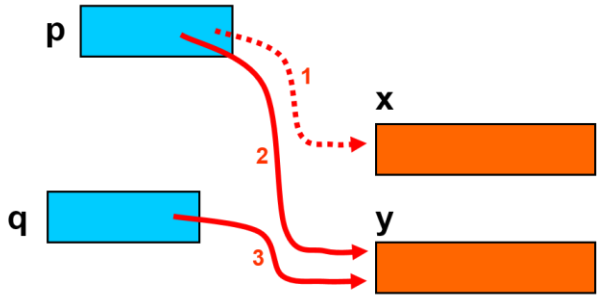
Some Key Points From the Previous Lecture

Pointers

1. **Pointer**:
   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
```

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
```

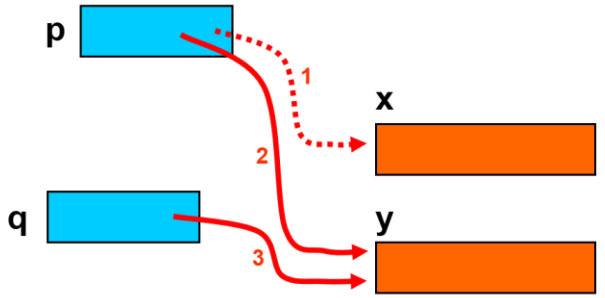
Some Key Points From the Previous Lecture

Pointers

1. **Pointer**:
   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
```

From previous lectures


```
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
```

From previous lectures

Dynamic Objects

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.

From previous lectures

Library Organization

To remind…

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

stdlib.h

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

stdlib.c

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

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.

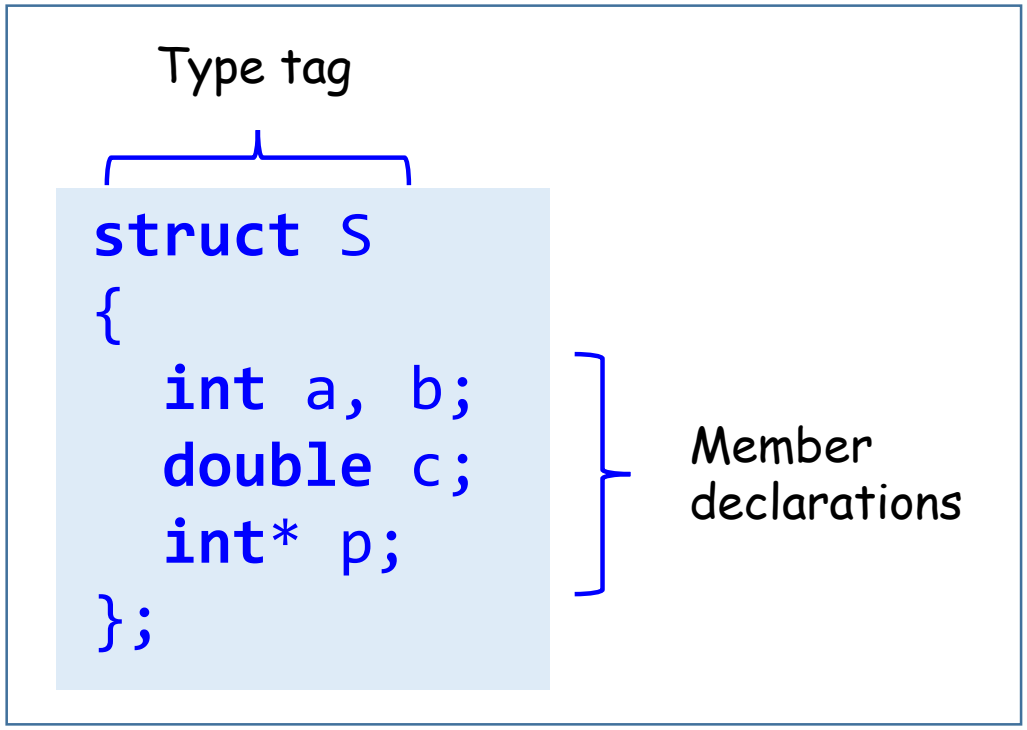
Outline: Today

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

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



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 structure elements:

- 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?

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

Usual initialization

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

Advanced syntax

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

```
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;
```

Advanced syntax

```
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 0th element of** `w` **and the 0th element of the 1st 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 0th element of `w` and the 0th element of the 1st 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 synonym for "array of unspecified size containing integers"

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"

```
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"

```
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

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

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

Structures: Alignment

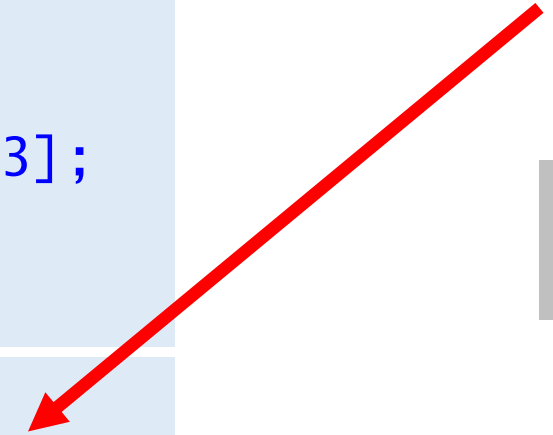
```
struct S
{
  char* m1;
  short m2[3];
  long m3;
};
```

```
...
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;
};
```

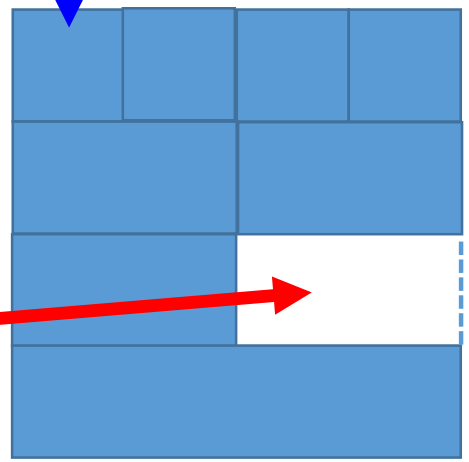
```
...
struct S s;
...
```

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.

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

Structures: Bit-fields

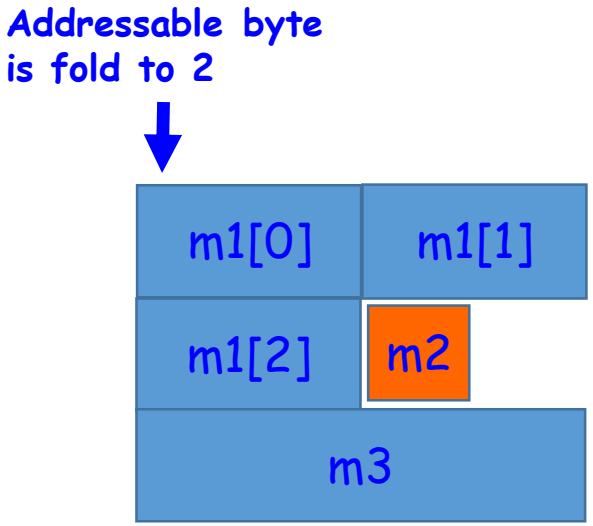
ISO C Standard, 6.7.2.1, §9-11

…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

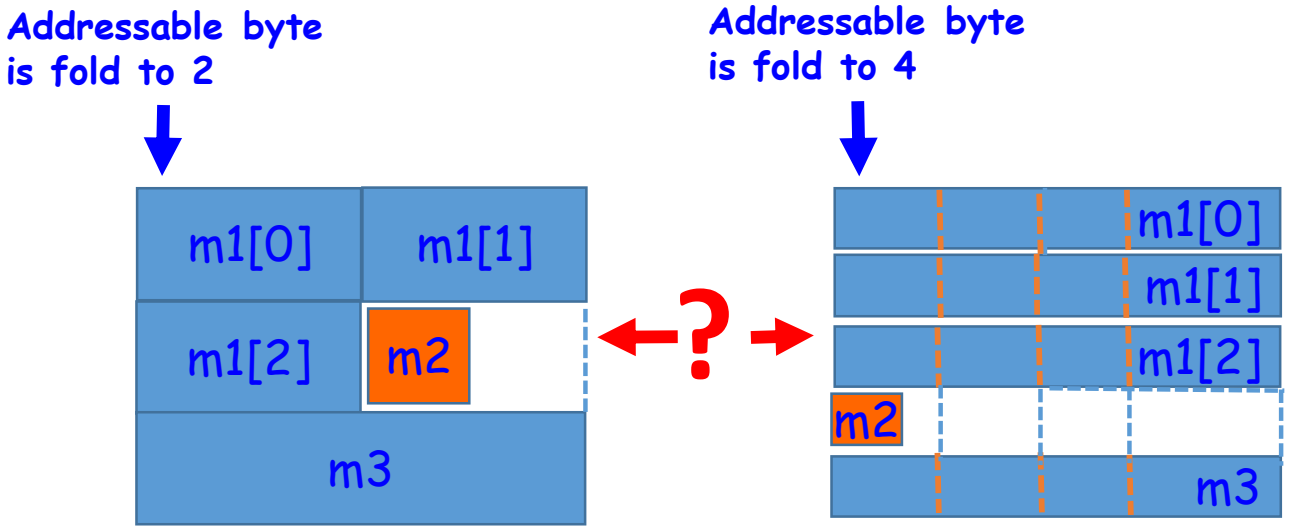
ISO C Standard, 6.7.2.1, §9-11

…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

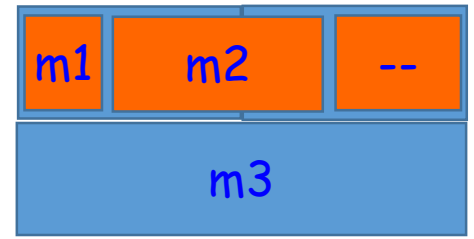
ISO C Standard, 6.7.2.1, §11

…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**.

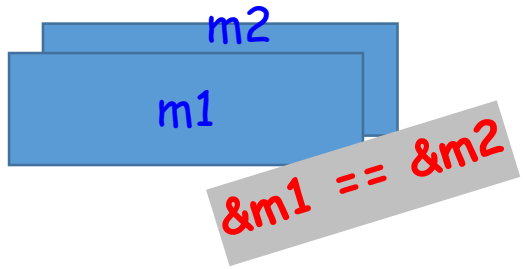
```
union U
{
  unsigned int m1;
  int*         m2;
};
```

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.

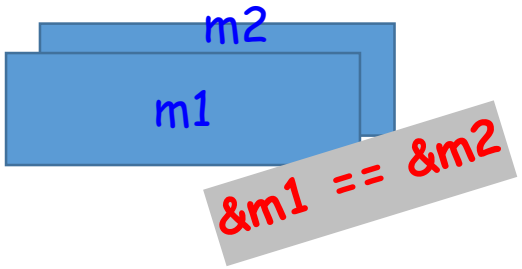
sizeof(U) == max(**sizeof**(m1), **sizeof**(m2))

Unions

ISO C Standard, 6.7.2.1, §6

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

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



Both m1 and m2 are in the same memory.

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

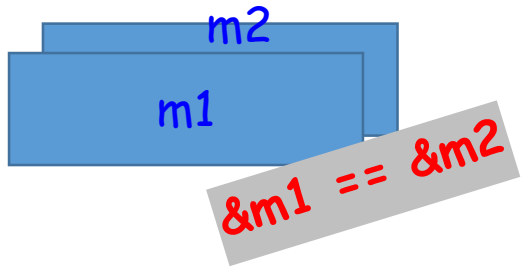
```
int x;
union U u;
...
u.m2 = &x;
...
unsigned y = u.m1;
```

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;
};
```



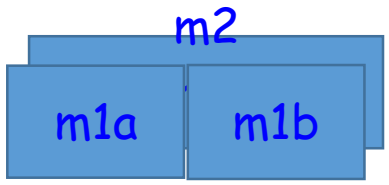
Both m1 and m2 are in the same memory.

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

Enumerations

An example:
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?

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:
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:
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.

Preprocessing

Preprocessing

"I would have killed preprocessor"
- B.Stroustrup

Major ideas and constructs

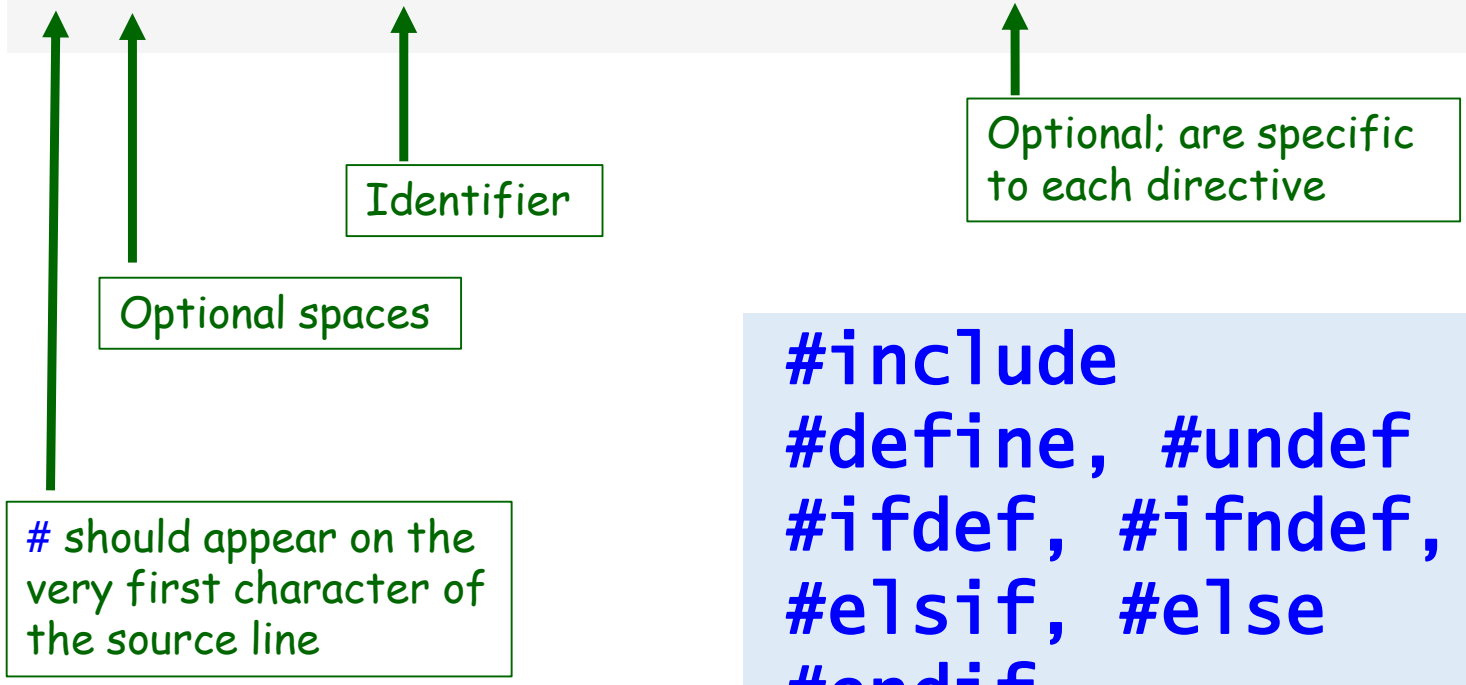
- This is purely **text-to-text** processing.
- **No C syntax/semantic rules** are involved.
- Preprocessing is usually performed by a separate tool – **preprocessor** – following its own rules.
- The main preprocessing mechanism is **text substitution**: some parts of the source text get replaced for other texts. Substitution process is usually **repetitive**.
- Main constructs:
  * **preprocessing directives**
  * **macro calls**
- The most popular preprocessing directive:
  `#include "filename"`

Preprocessing directives

They specify rules for substitution

Common syntax

```
# directive parameter(s)
```




```
#include
#define, #undef
#ifdef, #ifndef, #if
#elsif, #else
#endif
...
```

#include: Text inclusion

mainFile.ext

```
Some text
#include "toInclude.ext"
Some text
...
Some text
```


```
Some text
Line 1
Line 2
Line 3
...
Line N
Some text
...
Some text
```

toInclude.ext

```
Line 1
Line 2
Line 3
...
Line N
```

#include: Text inclusion

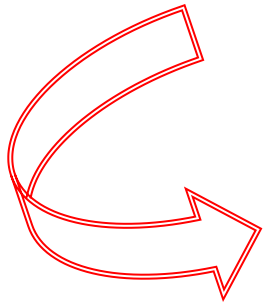
mainFile.ext

```
Some text
#include "toInclude.ext"
Some text
...
Some text
```


```
Some text
Line 1
Line 2
Line 3
...
Line N
Some text
...
Some text
```

toInclude.ext

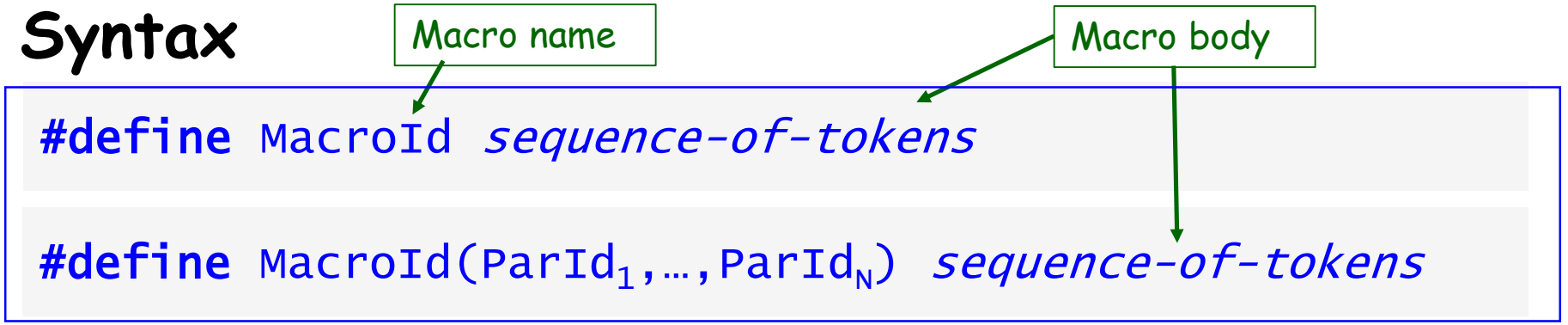
```
Line 1
Line 2
Line 3
...
Line N
```



Important:
If there are preprocessing directives within the text that was included (i.e., among `Line1`, …, `LineN`) then the resulting text **in turn gets preprocessed**.

#define: Macro definition

Syntax



Semantics: macro expansion

1. For each occurrence of `MacroId` in the source text, it gets replaced for the sequence of tokens specified after it.
2. For each occurrence of the construct like

   `MacroId(`*`SeqOfTokens`*$_1$`,…,`*`SeqOfTokens`*$_N$`)`

   it gets replaced for the sequence of tokens, **and** each occurrence of $\texttt{ParId}_i$ in the macro body gets replaced for the corresponding token sequence $\texttt{SeqOfTokens}_i$.

#define: Macro expansion

Examples

```
#define M x+y-
...
int a = M b;
```

Preprocessing

```
...
int a = x+y- b;
```

#define: Macro expansion

Examples

```
#define M x+y-
...
int a = M b;
```

Preprocessing

```
...
int a = x+y- b;
```

How to modify C syntax ☺

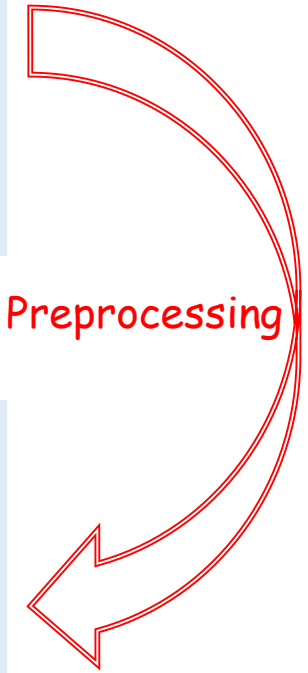
```
#define when if (
#define then ) {
#define end }
...
when a>0 then
   ...
end
```

```
...
if ( a>0 ) {
   ...
}
```

#define: Macro expansion

Examples

```
#define Max(a,b) a > b ? a : b
...
int test = Max(x,7);
```

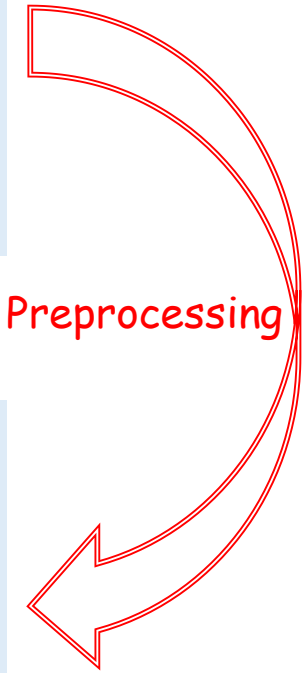



```
int test = x > 7 ? x : 7;
```

#define: Macro expansion

Examples

```
#define Max(a,b) a > b ? a : b
...
int test = Max(x,7);
int res1 = Max(x+y,x-y);
```

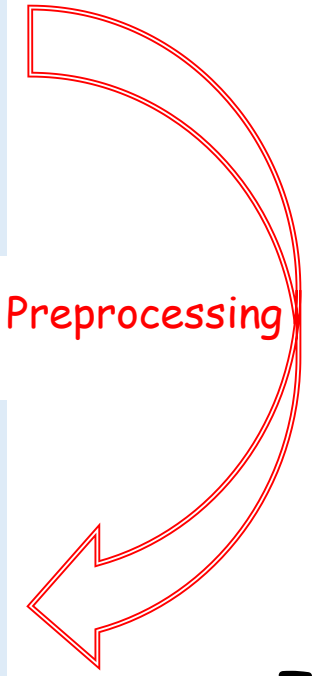



```
int test = x > 7 ? x : 7;

int res1 = x+y > x-y ? x+y : x-y;
```

#define: Macro expansion

Examples

```
#define Max(a,b) a > b ? a : b
...
int test = Max(x,7);
int res1 = Max(x+y,x-y);
int res2 = Max(x+=y,x*=y);
```



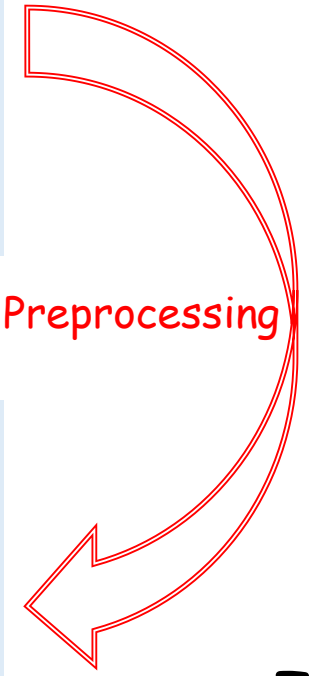

```
int test = x > 7 ? x : 7;

int res1 = x+y > x-y ? x+y : x-y;

int res2 = x+=y > x*=y ? x+=y : x*=y;
```

Is it the valid result??

#define: Macro expansion

Examples

```
#define Max(a,b) a > b ? a : b
...
int test = Max(x,7);
int res1 = Max(x+y,x-y);
int res2 = Max(x+=y,x*=y);
```




```
int test = x > 7 ? x : 7;

int res1 = x+y > x-y ? x+y : x-y;

int res2 = x+=y > x*=y ? x+=y : x*=y;
```

Is it the valid result??

Solution

```
#define Max(a,b) (a) > (b) ? (a) : (b)
```

#if(n)def: Conditional inclusion

```
int i =
#ifdef Max
     Max(x,7);
#else
     x>7 ? x : 7;
#endif
```

Text to be included to the resulting text in case Max macro was defined before (with any body)

Text to be included to the resulting text otherwise

Preprocessing

```
int i =
     Max(x,7);
```

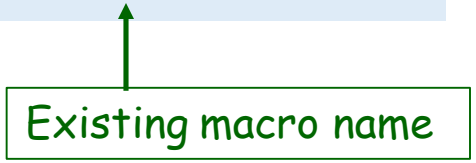
Directive with inverted condition

#ifndef

#ifdef & #undef

Syntax

```
#undef MacroId
```



Semantics

Just removes the macro `MacroId`

Example

```
#ifdef Max
#undef Max
int Max(int a, int b)
{
   return a>b ? a : b;
}
#endif
```

#include & #ifndef

mainFile.ext

```
Some text
#include "toInclude.ext"
#include "toInclude.ext"
Some text
...
Some text
```

How to prevent duplication??

#include & #ifndef

mainFile.ext

```
Some text
#include "toInclude.ext"
#include "toInclude.ext"
Some text
...
Some text
```

How to prevent duplication??

toInclude.ext

```
#ifndef __toInclude
#define __toInclude
Line 1
Line 2
Line 3
...
Line N
#endif
```

The solution

#if: Conditional inclusion

Semantics of #if is the same as for #ifdef
but the value of **an expression** is checked

```
int i =
#if Expression
    Max(x,7);
#else
    x>7 ? x : 7;
#endif
```

Text to be included to the resulting text in case *Expression* is evaluated to non-zero.

Text to be included to the resulting text otherwise

Preprocessing

```
int i =
    Max(x,7);
```

Expression should be actually **constant expression** – i.e., it should be evaluated while preprocessing.

Predefined macros

And an example of **#if**

Example

```
#define __STDC_ISO_10646__ 199901L
```

- Predefined in the ISO/IEC Standard 9899:1999

Predefined macros

And an example of **#if**

Example

```
#define __STDC_ISO_10646__ 199901L
```

- **Predefined in the ISO/IEC Standard** 9899:1999

```
#if __STDC_ISO_10646__ >= 199901L
   Some code that's specific to the
   version of C of 1999 or later
#elsif __STDC_ISO_10646__ >= 199409L
   Equivalent code legal in the
   previous version of the C standard
#else
   Equivalent code for an older
   version of C
#endif
```