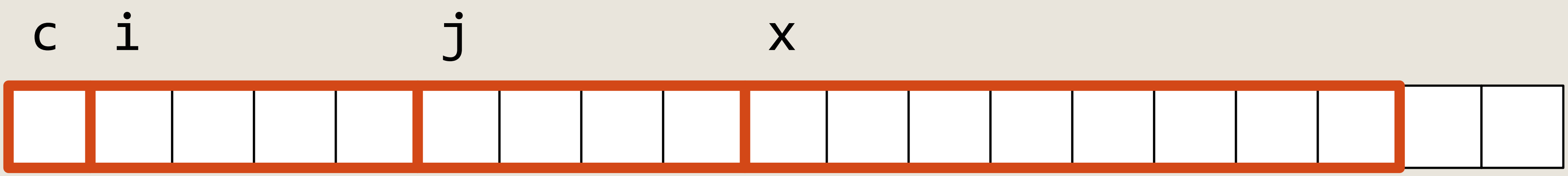
2nd week agenda

- Functions definitions and declarations
- Arrays
- Pointers
- Structs

Memory and arrays

Memory

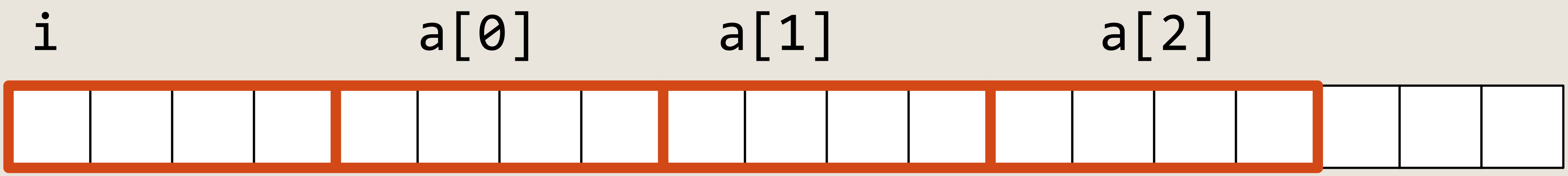
```c
int main()
{
    char c;
    int i,j;
    double x;
```



Arrays

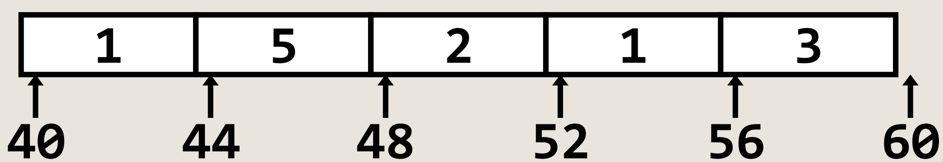
Defines a block of consecutive cells

```c
int main()
{
    int i;
    int a[3];
```



Arrays - the [ ] operator

```
int arr[5] = { 1, 5, 2, 1 ,3  };
/*arr begins at address 40*/
```



Address Computation Examples:

```
1.   arr[0]    40+0*sizeof(int) = 40
2.   arr[3]    40+3*sizeof(int) = 52
3.   arr[i]    40+i*sizeof(int) = 40 + 4*i
4.   arr[-1]   40+(-1)*sizeof (int) = 36  // can be the code
                                          // segment or other variables
```

Arrays

C does not provide any run time checks:

```
int a[4];
a[-1] = 0;
a[4] = 0;
```

This will compile and run…

But can lead to unpredictable results/crash.

It is the programmer's responsibility to check whether the index is out of bound.

Arrays

C does not provide array operations:

```c
int a[4];
int b[4];

a = b; // illegal

// and how about:

if( a == b ) // legal, address comparison (==0)
```

Array Initialization

```
int arr[3] = {3, 4, 5}; // Good

int arr[] = {3, 4, 5};  // Good - The same

int arr[3] = {0};       // Init all items to 0, takes O(n)

int arr[4] = {3, 4, 5}; // Bad style - The last is 0

int arr[2] = {3, 4, 5}; // Bad
int arr[2][3] = {{2,5,7},{4,6,7}};// Good

int arr[2][3] = {2,5,7,4,6,7};   //Good – The same

int arr[3][2] = {{2,5,7},{4,6,7}}; // Bad

int arr[3];

arr = {2,5,7}; // Bad - array assignment only in initialization
```

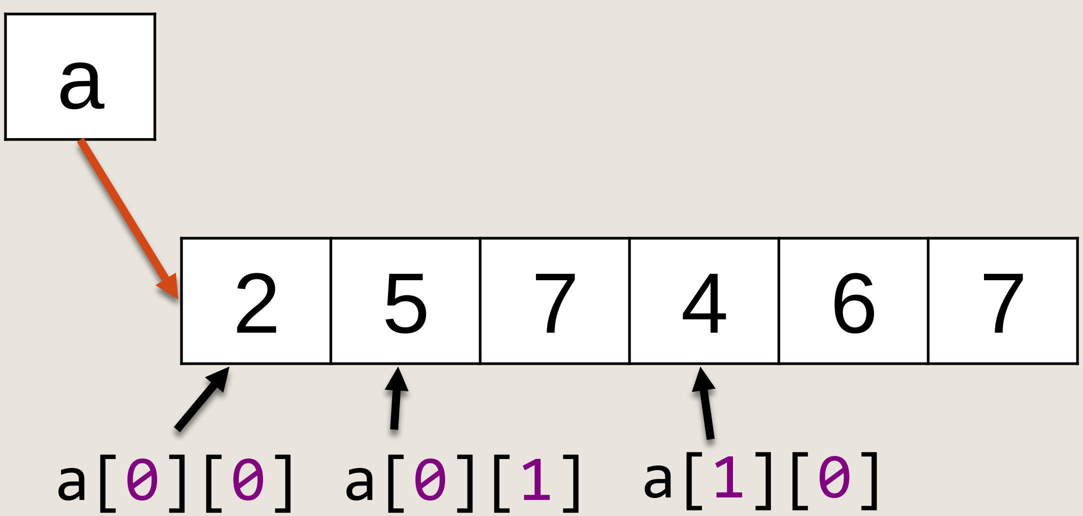
2D Array Memory Map

```
int a[2][3] = {{2,5,7},{4,6,7}};
```

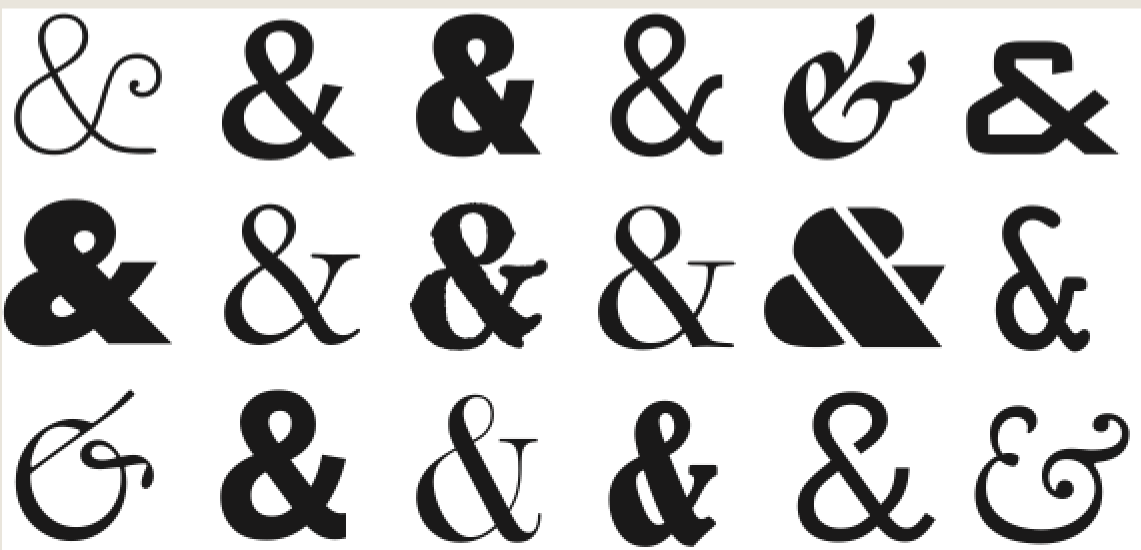
Generally we would look at arrays as

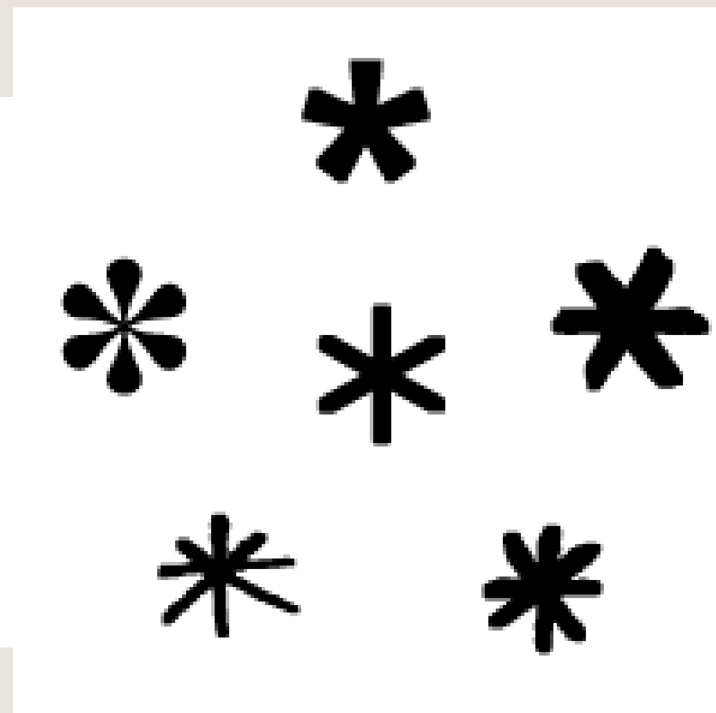
```
int a[ROWS][COLS];
```

2	5	7
4	6	7



Pointers





Pointers

Pointers are variables that store the address of other variables.

- **Declaration**

  `<type> *p;  (e.g. int *p;)`

  p points to object of type `<type>`

- **Pointer → value (de-reference)**

  `*p = x; (e.g. *p=5;)`

  `y = *p; (e.g. int y = *p;)`

  *p refers to the object p points to

- **Value → pointer**

  &x - the pointer to x `(e.g.  p = &y;)`

Pointers – spaces in declaration

```
int *p; // p is a pointer to an int
int* p; // p is a pointer to an int
int*p;  // p is a pointer to an int
int * p;// p is a pointer to an int

int *p, q; // p is a pointer to an int
           // q is an int

int* p, q; // same, but much less readable
           // so don’t do that
```

Pointers - 64 bit!

```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
```

i				j				x										
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

```
int main()
{
   int i,j;
   int *x; // x points to an integer
   i = 1;
   x = &i;
   j = *x;
   x = &j;
   (*x) = 3;
```

i						j				x								
1	0	0	0	?	?	?	?	?	?	?	?	?	?	?	?			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

```
int main()
{
   int i,j;
   int *x; // x points to an integer
   i = 1;
   x = &i;
   j = *x;
   x = &j;
   (*x) = 3;
```

i				j				x										
1	0	0	0	?	?	?	?	0	0	0	0	0	0	0	0			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
```

i j x

1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
```

i				j				x										
1	0	0	0	1	0	0	0	4	0	0	0	0	0	0	0			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

```
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
```

i j x

i				j				x										
1	0	0	0	3	0	0	0	4	0	0	0	0	0	0	0			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Pointers - 64 bit!

```c
int main()
{
    int i,j;
    int *x; // x points to an integer
    i = 1;
    x = &i;
    j = *x;
    x = &j;
    (*x) = 3;
```

X86 works in
Little Endian

i j x

1	0	0	0	3	0	0	0	4	0	0	0	0	0	0	0			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Example – the swap function

Does nothing

```
void swap(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}
int main()
{
   int x, y;
   x = 3; y = 7;
   swap(x, y);
   // now x==3, y==7
```

Works

```
void swap(int *pa, int *pb)
{
   int temp = *pa;
   *pa = *pb;
   *pb = temp;
}
int main()
{
   int x, y;
   x = 3; y = 7;
   swap(&x, &y);
   // x == 7, y == 3
```

Pointers & Arrays

```
int *p;
int a[3];
p = &a[0];  // same as p = a
*(p+1) = 1; // assignment to a[1]!
```

P								a[0]				a[1]				a[2]			
8	0	0	0	0	0	0	0					1	0	0	0				

Pointers & Arrays

Array name can **sometimes** be treated as the address of the first member.

```
int *p;
int a[4];
p = a;              // same as p = &a[0];
p[1] = 102;         // same as *(p+1)=102;
*(a+1) = 102;       // same as prev. line
p++;                // p == a+1 == &a[1]
a = p;              // illegal
a++;                // illegal
```

Pointers & Arrays - size

But:

```
int *p;
int a[4];
sizeof (p) == sizeof (void*)
sizeof (a) == 4 * sizeof (int)
```

→ Size of the array is known in compile time

Pointers & Arrays

```
int main()
{
    int arr[4] = {1,3,5,4};
    int i, sum = 0;
    for (i=0; i<sizeof(arr)/sizeof(arr[0]); ++i)
    {
        sum += arr[i];
    }
}
```

Pointers & Arrays

```
int foo( int *p );
```

and

```
int foo( int a[] );
and
int foo( int a[NUM] );
```

Are declaring the same interface.

In both cases, a **pointer to int** is being passed to the function foo

Pointers & Arrays

How about this code?

```
int sum (int arr[])
{
    int i, sum = 0;
    for (i=0; i<sizeof(arr)/sizeof(arr[0]); ++i)
    {
        sum += arr[i];
    }
    return sum;
}
```

error: sizeof (arr) = sizeof (void*)

Pointers & Arrays

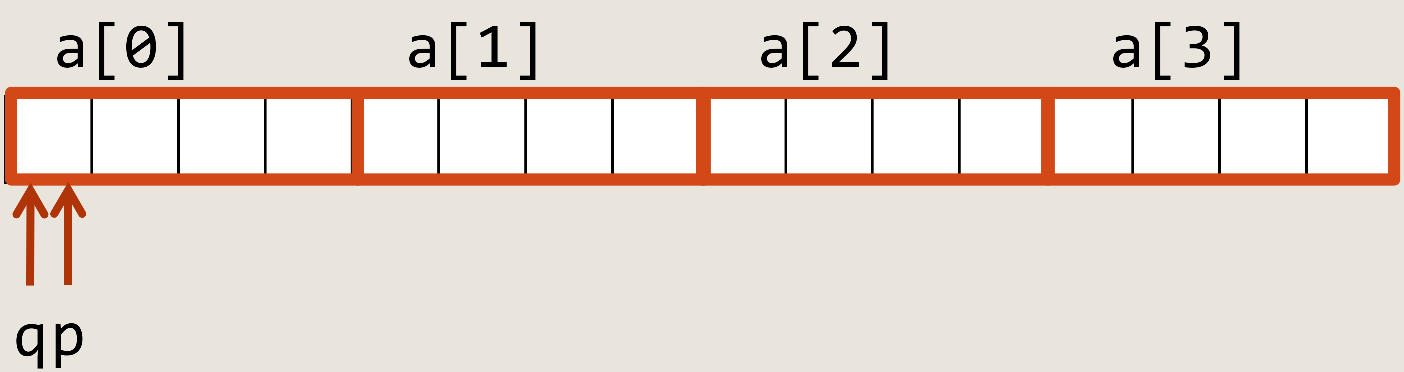
```
int sum (int arr[], int n)
{
    int i, sum = 0;
    for (i=0; i<n; ++i)
    {
        sum += arr[i]; // arr[i] = arr+ i*sizeof(int)
    }
    return sum;
}
```

Array size must be passed as a parameter

Pointer Arithmetic

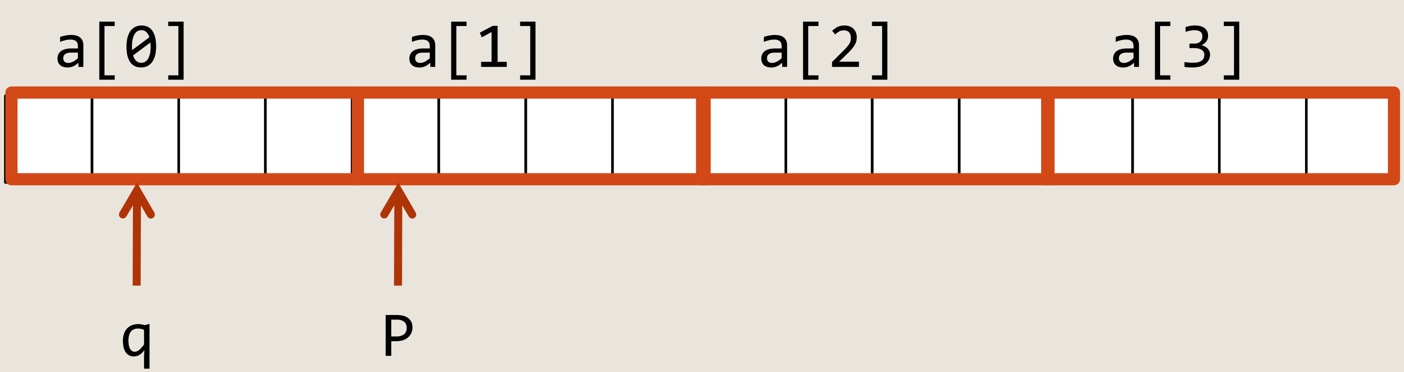
```
int a[3];
int *p = a;
char *q = (char *)a; // Explicit cast
// p and q point to the same location
p++;
q++;
```

What is the difference?



Pointer Arithmetic

```c
int a[3];
int *p = a;
char *q = (char *)a; // Explicit cast
// p and q point to the same location
p++; // increment p by 1 int (4 bytes)
q++; // increment q by 1 char (1 byte)
```



Pointer Arithmetic

```
int FindFirstNonZero( int a[], int n )
{
    int *p=a;
    for( ; a < p+n && (*a) == 0; a++ );
    return a-p;
}
```

Same as

```
int FindFirstNonZero( int a[], int n )
{
    int i;
    for( i = 0; i < n && a[i] == 0; i++ );
    return i;
}
```

Preferable

Pointer Arithmetic

```
int a[4];
int *p = a;
long i = (long)a;
long j = (long)(a+1); // adds 1*sizeof(int) to a
long dif = (long)(j-i);// dif = sizeof(int), not 1
```

Be careful: Pointer arithmetic works just with pointers

void *

`void *p` defines a pointer to undetermined type

```
int j;
int *p = &j;
void* q = p; // no cast needed
p = (int*)q ; // cast is needed
```

All pointers can be casted one to the other, it may be useful sometimes, but beware…

void *

- No pointer arithmetic is defined for void*
  (gcc has an extension, treating the size of a void as 1)
- We cannot access to the content of the pointer – dereferencing is not allowed

```
int j;
void *p = &j;
int k = *p;         // illegal
int k = (int)*p ; // still illegal
int k = *(int*)p; // legal
```

NULL pointer

Special value: uninitialized pointer (defined in stdlib.h):

```
int *p = NULL;


if( p != NULL )
{


}
```

NULL pointer

```
int *p = NULL;
*p = 1;
```

and also

```
int *p;
*p = 1;
```

Will compile… but will (probably) lead to runtime error

Pointers to pointers

Pointer is a variable type, so we can create a pointer to pointer.

```
int main()
{
    int n = 17;
    int *p = &n;
    int **p2 = &p;
    printf("the address of p2 is %p \n",&p2);
    printf("the address of p is %p \n",p2);
    printf("the address of n is %p \n",*p2);
    printf("the value of n is %d \n",**p2);
    return 0;
}
```

Arrays and Pointers – reminder and multiple dimension arrays

- Array

```
int a[4];
int a[4][10];
int a[4][10][5];      ...
```

- Pointers
  - ```
    int j;
    int *p = &j
    ```
  - NULL
  - Pointer Arithmetic
  - void*
  - sizeof
    - in functions
    ```

Structs

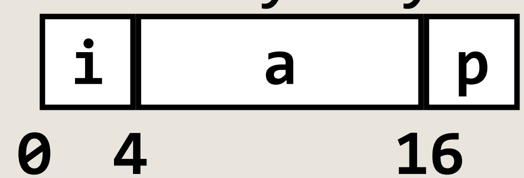
The origin of classes

Structs

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types
- Example:

```
struct rec
{
    int i;
    int a[3];
    int *p;
};
```

Memory Layout



Struct initialization

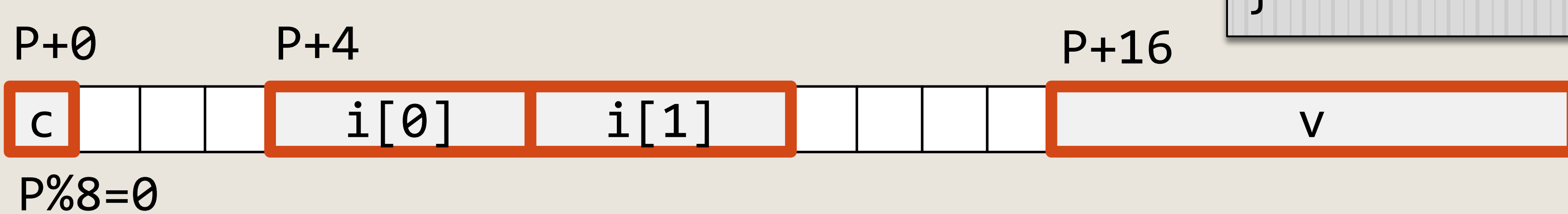
`struct`s can be initialized in a way similar to arrays:

```
struct rec
{
    int i;
    int a[3];
    int *p;
};
int k;
struct rec r = { 5, 0,1,2, &k };
r.i=1;
r.a[0]=5;
r.p=&k;
```

sizeof struct / structs padding / alignment

- sizeof (struct) $\geq$ sizeof (its members)
- Each member will be aligned to the lowest address after the previous member that satisfies: mod(address/sizeof(member))=0
  - e.g. int (4 bytes) will be aligned to 8/12/…
- Hardware fetches memory in chunks
  - Some platforms (CPUs) don't support "mis-aligned" memory access
- The compiler takes care of that for us
- We can plan a compact struct
- We can tell the compiler to “pack” the structs

```c
struct S1
{
    char c;
    int i[2];
    double v;
}
```



`Structs` are like any other type

- variables of type struct
- a pointer to a struct
- arrays of structs
- pass a struct to a function
- return a struct from a function
- …

Access to struct members via pointers

```c
struct MyStr
{
    int _a[10];
};
```

```c
main()
{
    struct MyStr x;
    struct MyStr *p_x = &x;

    x._a[2] = 3;

    (*p_x)._a[3] = 5;
    p_x->_a[4] = 6;
}
```

The -> operator

typedef

- Synonyms for variable types – make your program more readable
- Can be used also for built-in types

```
typedef <existing_type_name> <new_type_name>
```

```
typedef struct MyStr MyStrStruct;
typedef struct MyStr MyStr;
typedef unsigned long size_t;

size_t l = strlen(“abc”);
```

typedef

- Defining struct typedef while defining the struct

complex.h

```
typedef struct Complex
{
    double _real, _imag;
} Complex;

Complex addComplex(Complex, Complex);
Complex subComplex(Complex, Complex);
```

typedef

- Defining struct typedef while defining the struct

complex.h

```
typedef struct Complex
{
   double _real, _imag;
} Complex;

Complex addComplex(Complex, Complex);
Complex subComplex(Complex, Complex);
```

optional

typedef

- Defining struct typedef while defining the struct

complex.h

```
typedef struct
{
   double _real, _imag;
} Complex;

Complex addComplex(Complex, Complex);
Complex subComplex(Complex, Complex);
```

Structs – poor oop

complex.h

```c
struct Complex
{
    double _real, _imag;
};
struct Complex addComplex(struct Complex, struct Complex);
```

complex.c

```c
#include "complex.h"
// implementation
struct Complex addComplex(struct Complex a, struct Complex b)
{
```

MyProg.c

```c
#include "complex.h"
int main()
{
    struct Complex c;
    ...
```

#if – header safety

Complex.h:

```
struct Complex
{ ... };
```

MyStuff.h:

```
#include "Complex.h"
```

Main.c:

```
#include "MyStuff.h"
#include "Complex.h"
```

Error:
Complex.h:1: redefinition of `struct Complex'

#if – header safety

Complex.h (revised):

```
#ifndef COMPLEX_H
#define COMPLEX_H
struct Complex
{
...
#endif
```

Main.c:

```
#include "MyStuff.h"
#include "Complex.h" // no error this time
```

structs copying

Copy structs using ‘=‘:
copies struct values

just

```
Complex a,b;
a._real = 5;
a._imag = 3;
b = a;
```

a:

```
_real = 5
_imag = 3
```

b:

```
_real = 5
_imag = 3
```

Arrays in structs copying

struct definition:

vec.h

```
typedef struct Vec
{
   double _arr [MAX_SIZE];
} Vec;

Vec addVec(Vec, Vec);
...
```

Arrays in structs copying

copy struct using ‘=‘:

```
Vec a,b;
a._arr[0] = 5;
a._arr[1] = 3;
b = a;
```

a:

```
_arr =
{5,3,?,…}
```

b:

```
_arr =
{5,3,?,…}
```

But !!!

Pointers in structs copying

struct definition:

vec.h

```
typedef struct Vec
{
   double _arr[MAX_SIZE];
   double * _p_arr;
} Vec;

Vec addVec(Vec, Vec);
...
```

Pointers in structs copying

Copy structs using ‘=‘:

copies **just** struct values!

```
Vec a,b;
a._arr[0] = 5;
a._arr[1] = 3;
a._p_arr = a._arr;
b = a;
```

a:

```
_arr =
{5,3,?,…}

_p_arr =
0x55
```

b:

```
_arr =
{?,?,?,…}

_p_arr =
?
```

Pointers in structs copying

Copy structs using '=':

copy just struct values!!!

```
Vec a,b;
a._arr[0] = 5;
a._arr[1] = 3;
a._p_arr =
a._arr;
b = a;
```

a:

```
_arr =
{5,3,?,…}

_p_arr =
0x55
```

b:

```
_arr =
{5,3,?,…}

_p_arr =
0x55
```

Pointers copied by value!!!

Pointers in structs copying

The result:

```
Vec a,b;
a._arr[0] = 5;
a._arr[1] = 3;
a._p_arr = a._arr;
b = a;
*(b._p_arr) = 8;
// same as b->_p_arr = 8
printf ("%f", a._arr[0]);
```

```
// output
8
```

How to copy structs correctly?

Implement a clone function:

```
void cloneVec (Vec *a, Vec *b)
{
    int i=0;
    for (i=0;I<MAX_SIZE;i++)
    {
        b->_arr[i] = a->_arr[i];
    }
    b->_p_arr = b->_arr;
}
```

Arrays & structs as arguments

When an **array** is passed as an argument to a function, the **address of the 1st element** is passed.

Structs are passed **by value**, exactly as the basic types.

Arrays & structs as arguments

```
typedef struct MyStr
{
    int _a[10];
} MyStr;

void f(int a[])
{
    a[7] = 89;
}

void g(MyStr s)
{
    s._a[7] = 84;
}
```

Output:

```
main()
{
    MyStr x;
    x._a[7] = 0;
    f(x._a);
    printf("%d\n", x._a[7]);
    g(x);
    printf("%d\n", x._a[7]);
}
```

Arrays & structs as arguments

```
typedef struct MyStr
{
    int _a[10];
} MyStr;

void f(int a[])
{
    a[7] = 89;
}

void g(MyStr s)
{
    s._a[7] = 84;
}
```

```
main()
{
    MyStr x;
    x._a[7] = 0;
    f(x._a);
    printf("%d\n", x._a[7]);
    g(x);
    printf("%d\n", x._a[7]);
}
```

Output:
89
89

enum

User Defined Type - enum

- Enumerated data - a set of named constants.
- Usage: enum [identifier]{enumerator list}
- More readable code
- Code less error prone
- Use enum to eliminate magic numbers

Why magic numbers are bad?

```
enum { SUNDAY=1, MONDAY,TUESDAY,…};
enum Color{BLACK,RED,GREEN,YELLOW,BLUE,WHITE=7,GRAY};
enum Seasons
{
       E_WINTER,      // = 0 by default
       E_SPRING,      // = E_WINTER + 1
       E_SUMMER,      // = E_WINTER + 2
       E_AUTUMN       // = E_WINTER + 3
};
```

enum

- An aggregation of constant int values, that defines a type
- The defined constants can be used as int everywhere in the same scope:

```
int n=RED;
```

```
enum Seasons curr_season;
curr_season = E_AUTUMN;
curr_season = 19;                // legal, but ugly

int E_SUMMER;                    // error, redefinition
int prev_season = E_SUMMER;      // legal, but ugly
```

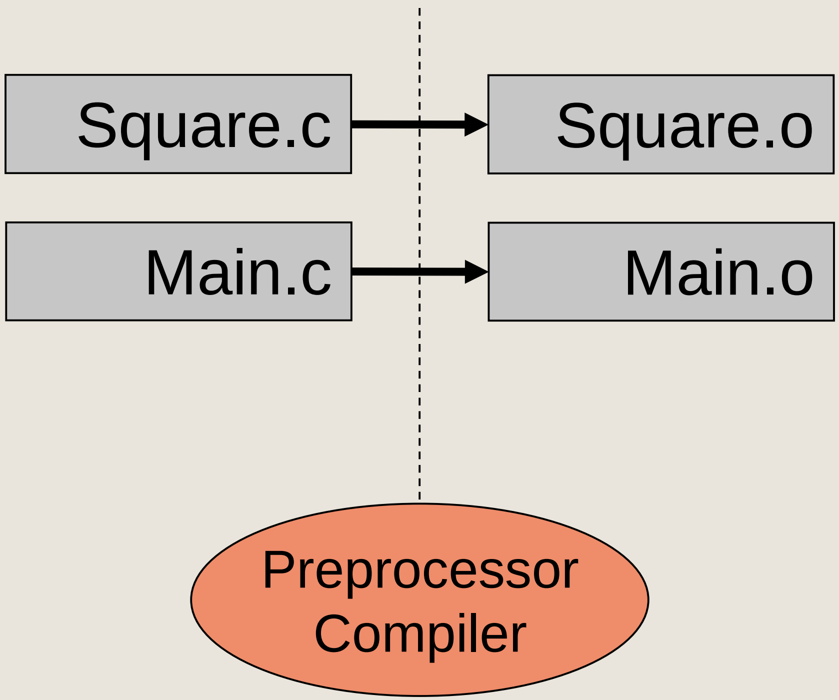
Use enum to eliminate magic numbers – alternative to #define

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

enum{INPUT_FILE_NAME = 1,OUTPUT_FILE_NAME,ARGS_NUM};

int main(int argc, char* argv[])
{
    if(argc != ARGS_NUM)
    {
        printf("usage: wc <input file name> <output file name>\n");
        exit(1);
    }

    FILE* inFile, outFile;

    inFile = fopen(argv[INPUT_FILE_NAME],"r");
    if( inFile == NULL)
    {
        printf("error reading file: %s\n", argv[INPUT_FILE_NAME]);
        exit(1);
    }
```

Revisiting compilation process

Compiling

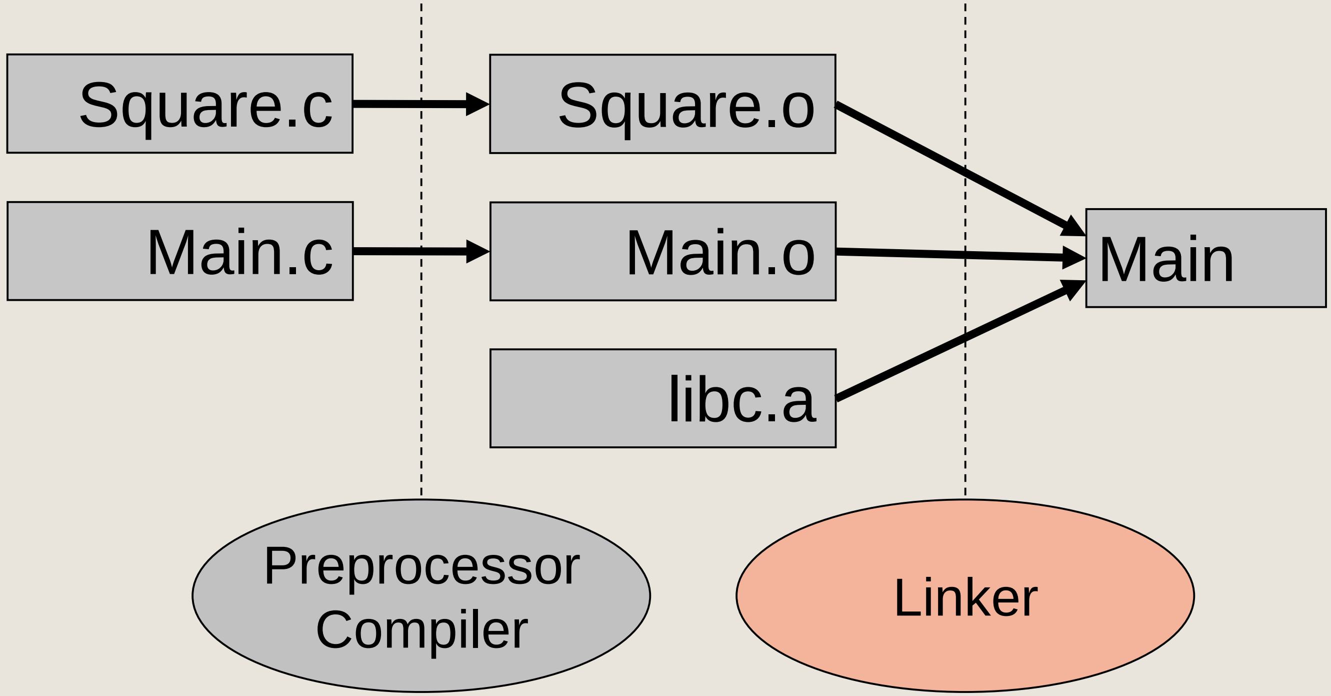
- Creates an object file for each code file (.c -> .o)
- Each .o file contains code (of functions, structs, variables etc..)
- Unresolved references still remain



Linking

Combines several object files into an executable file

- No unresolved references

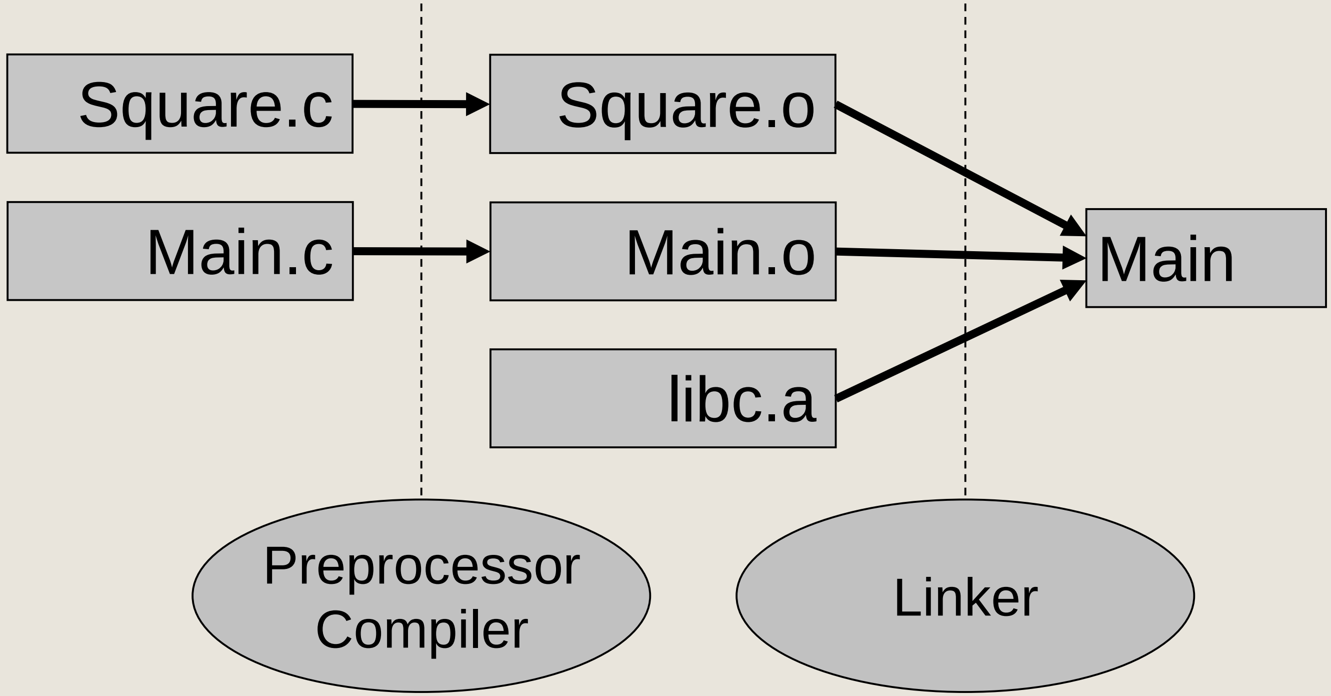


The whole process

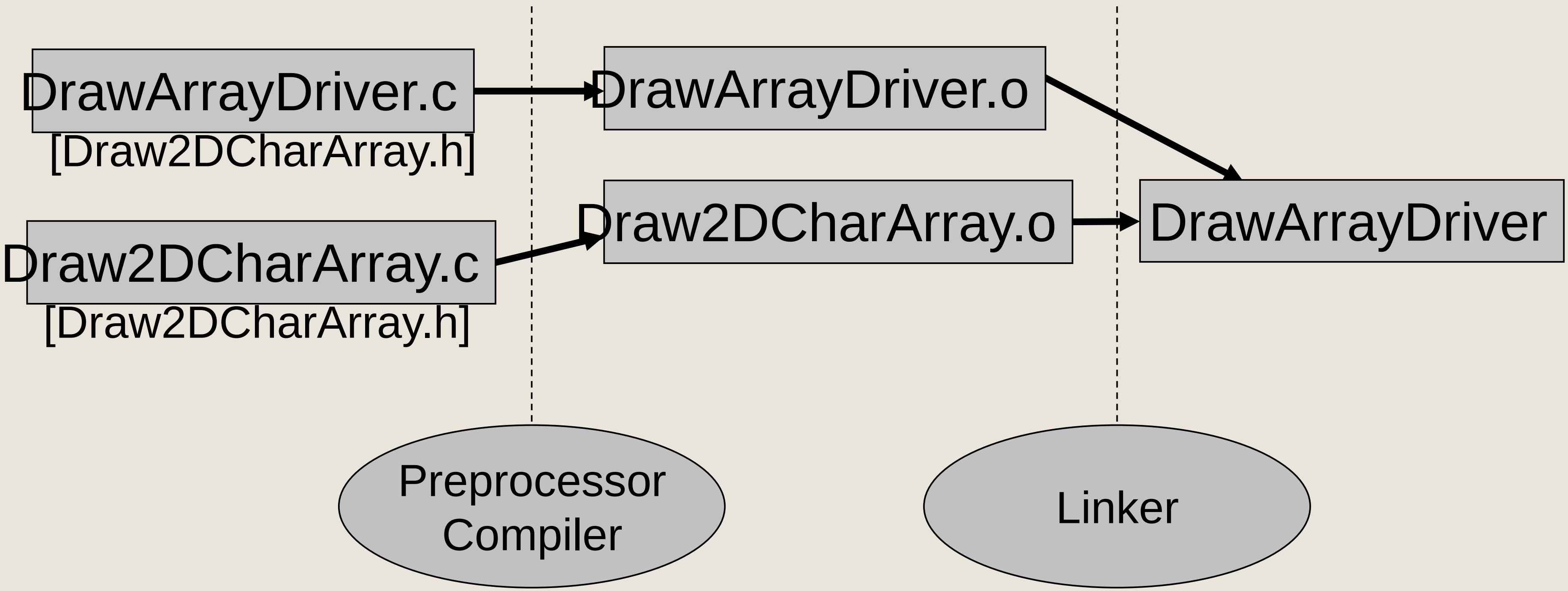
$ gcc –c –Wall Square.c –o Square.o

$ gcc –c –Wall Main.c –o Main.o

$ gcc Square.o Main.o –o Main



Compiling and Linking example



Link errors

The following errors appear only at link time

1. Missing implementation

```
> gcc -o –Wall Main Main.c
Main.o(.text+0x2c):Main.c: undefined
reference to `foo'
```

2. Duplicate implementation

```
> gcc -o Main Main.o foo.o
foo.o(.text+0x0):foo.c: multiple definition of
`foo'
Main.o(.text+0x38):Main.c: first defined here
```

Inter module scope rules

Visibility, duration and linkage

- **Translation unit/module**

a ".c" file (+ its included headers)

- **Visibility** – the lines in the code where an object (variable, function, typedef) is accessible through a declaration.
    - Global declaration – visible throughout the translation unit, starting from the declaration location.
    - Local declarations (inside {}) – visible in their block (starting from the declaration).
    - Can be hidden by inner scope declarations.
    - Scope is related to the compiler, not the linker.

Visibility, duration and linkage

- **Duration (a.k.a Lifetime):** the amount of time, where it is guaranteed that the memory for an object is allocated.
  - Functions - the entire running time of the program.
  - Globals - the entire running time of the program.
  - Locals – Until their scope ends.
  - Dynamic – will talk later.
  - Static – next slides

- We will talk about variables duration again when we learn about memory segments.

Static variables

- Static variables duration is the entire program running time.
- Static variables in a function keep their value for the next call to the function
- Memory is allocated on global space (called: static heap)

```
int getUniqueID()
{
   static int id=0;
   id++;
   return id;
}
int main()
{
   int i = getUniqueID(); //i=1
   int j = getUniqueID(); //j=2
}
```

Duration - example

```c
int func1( void ); // all running time.
int a; // all running time.
static int c; // all running time.
static void func2() // all running time.
{
  int b = 2; // until func2 ends
  static int e; // all running time.
}
```

Visibility, duration and linkage

- **Linkage:**

the association of a name (identifier of variables/ functions) to a particular entity (i.e. particular memory address).

  - **External linkage:**

    names are associate with the same object throughout the program.

    - All functions and global variables have external linkage (unless declared as static)

  - **Internal linkage:**

  names are associate with the same object in the particular translation unit.

    - objects declared as “static” have internal linkage.

  - **No linkage:**

  the object is unique to its scope

    - All locals have no linkage

Static and extern variables, cont.

"static" variable on the global scope

- Available only in the current module

"extern" variable

- May be defined outside the module

file1.c

```c
int y;
static int x;
int z;
int myFunc1()
{
    x = 3;
}
```

file2.c

```c
extern int y; // y should be imported (from file1.c )
extern int x; // x should be imported (from file1.c)
int myFunc2()
{
    extern int z; // z from file1.c
    y = 5;
    x = 3; //linker error
}
```

Static functions.

"static" function - available only in the current module.

funcs.h :

```c
static void Func1();
void Func2();
```

main.c:

```c
#include "funcs.h"
int main()
{
    Func1(); //link error
    Func2();
}
```

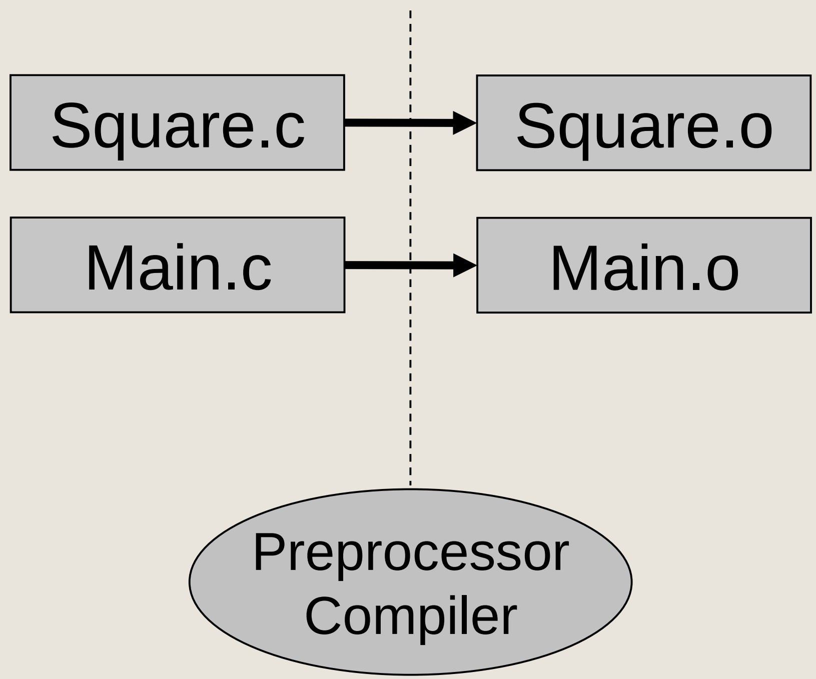
Linkage – example (to read at home)

```
int func1( void ); // func1 has external linkage.
int a;                // a has external linkage.
extern int b = 1;  // b has external linkage.
static int c;       // c has internal linkage.
static void func2( int d ) // func2 has internal
                             linkage; d has no  linkage.
{
   extern int a; // This a is the same as that
                    above, with  external linkage.
   int b = 2;     // This b has no linkage, and hides
                    the external b declared above.
   extern int c; // This c is the same as that   above,
                    and  retains internal linkage.
   static int e; // e has no linkage.
}
```

Multiple file project management

Compiling

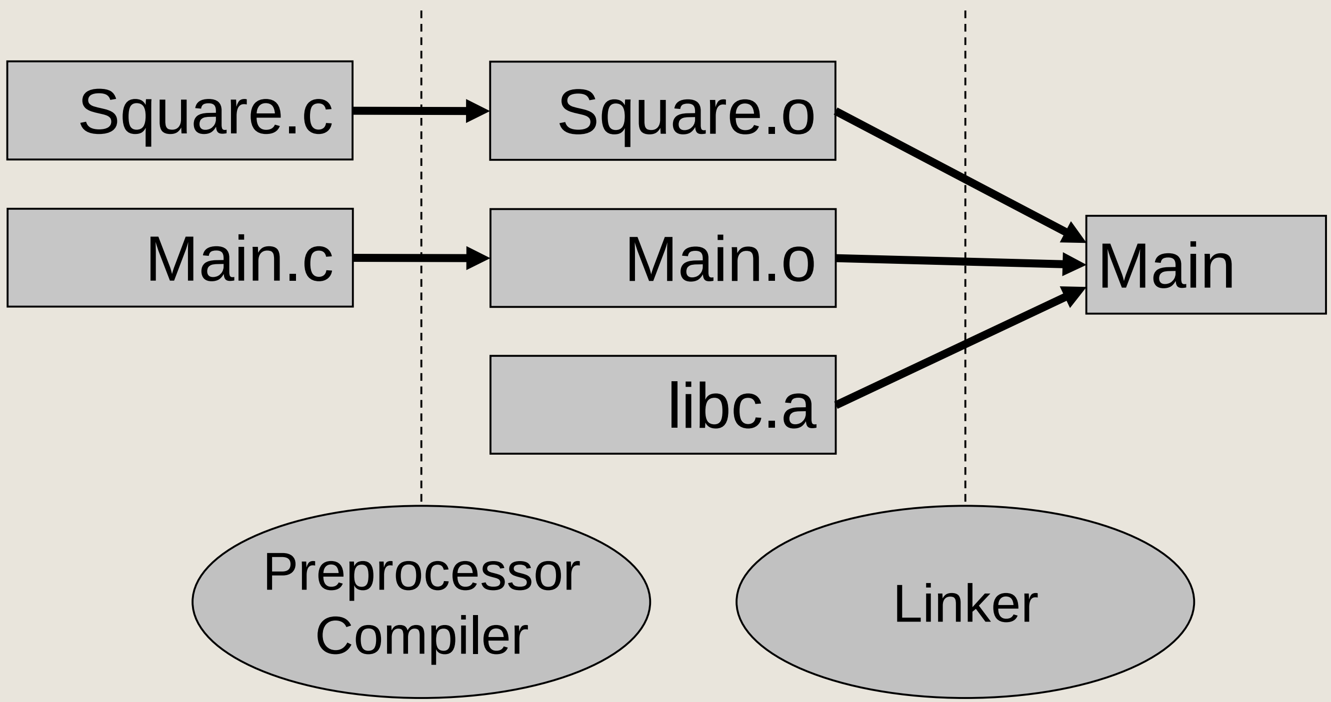
- Creates an object file for each code file (.c -> .o)
- Each .o file contains opcode of the C code of its *translation unit*.
- Unresolved references (function/ variables defined in other files) still remain



Linking

Combines several object files into an executable file

- Link the actual functions to their calls.
- No unresolved references.

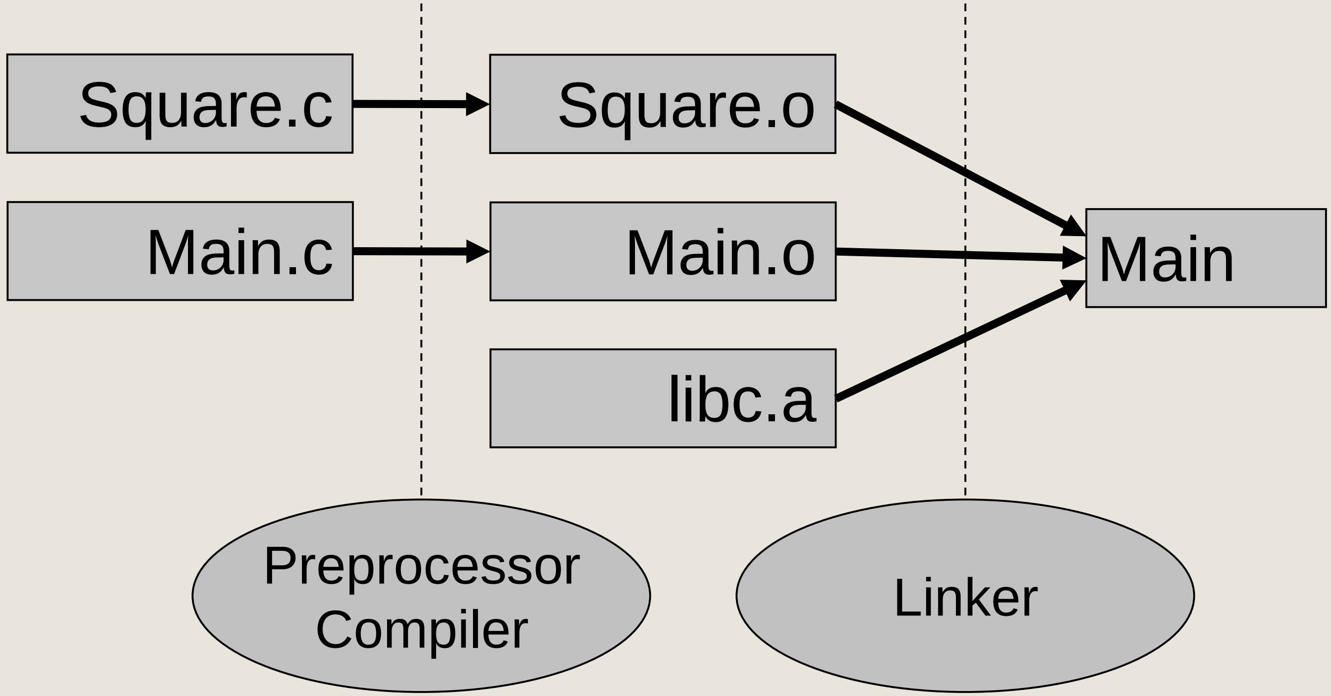


The whole process:

$ gcc –c Square.c –o Square.o

$ gcc –c Main.c –o Main.o

$ gcc Square.o Main.o –o Main



Make

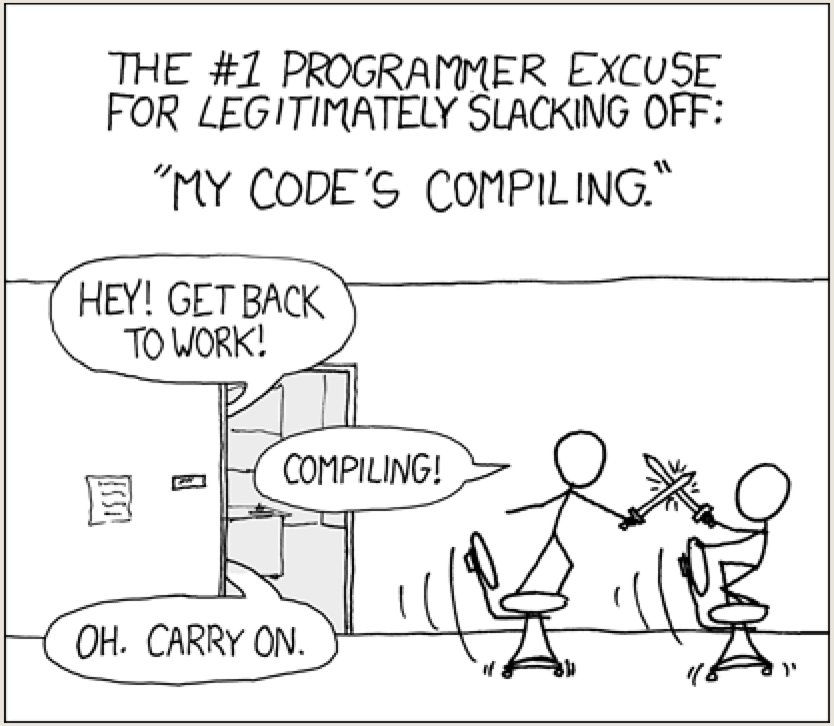
What is it?

- **Automatic** tool for projects management (not just c/c++)

What is it good for?

- Faster compilation/linkage => more productivity!
- Less boring work for the programmer => Less errors!

- man/google/**gnu** make



Make and Makefiles

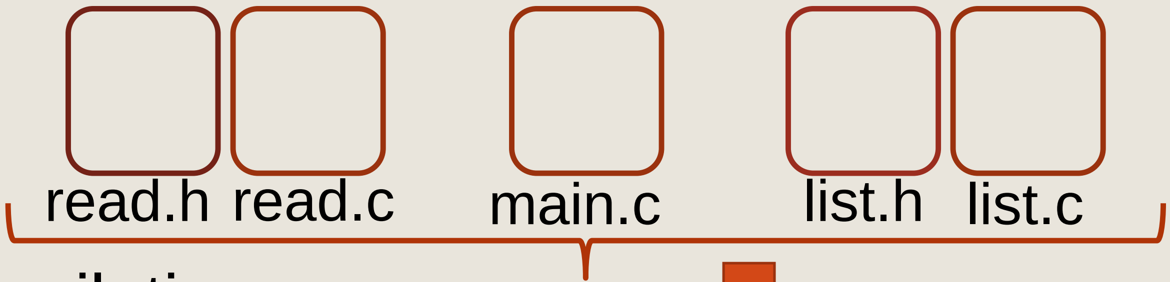
Make is a program who’s main aim is to update other programs in a “smart” way.

“smart” =

- Build only out-of-date files (use timestamps).
- Use the dependency graph for this.

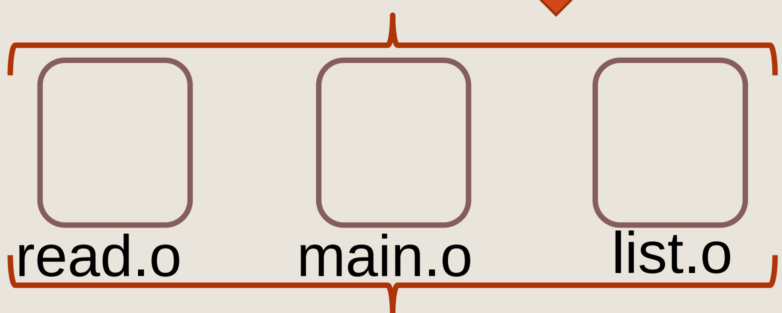
You tell make what to do by writing a *makefile*

Compilation & linkage



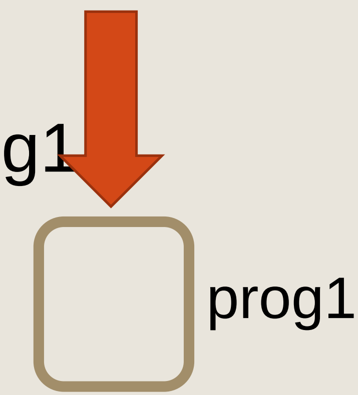
Compilation:

gcc -c read.c main.c list.c

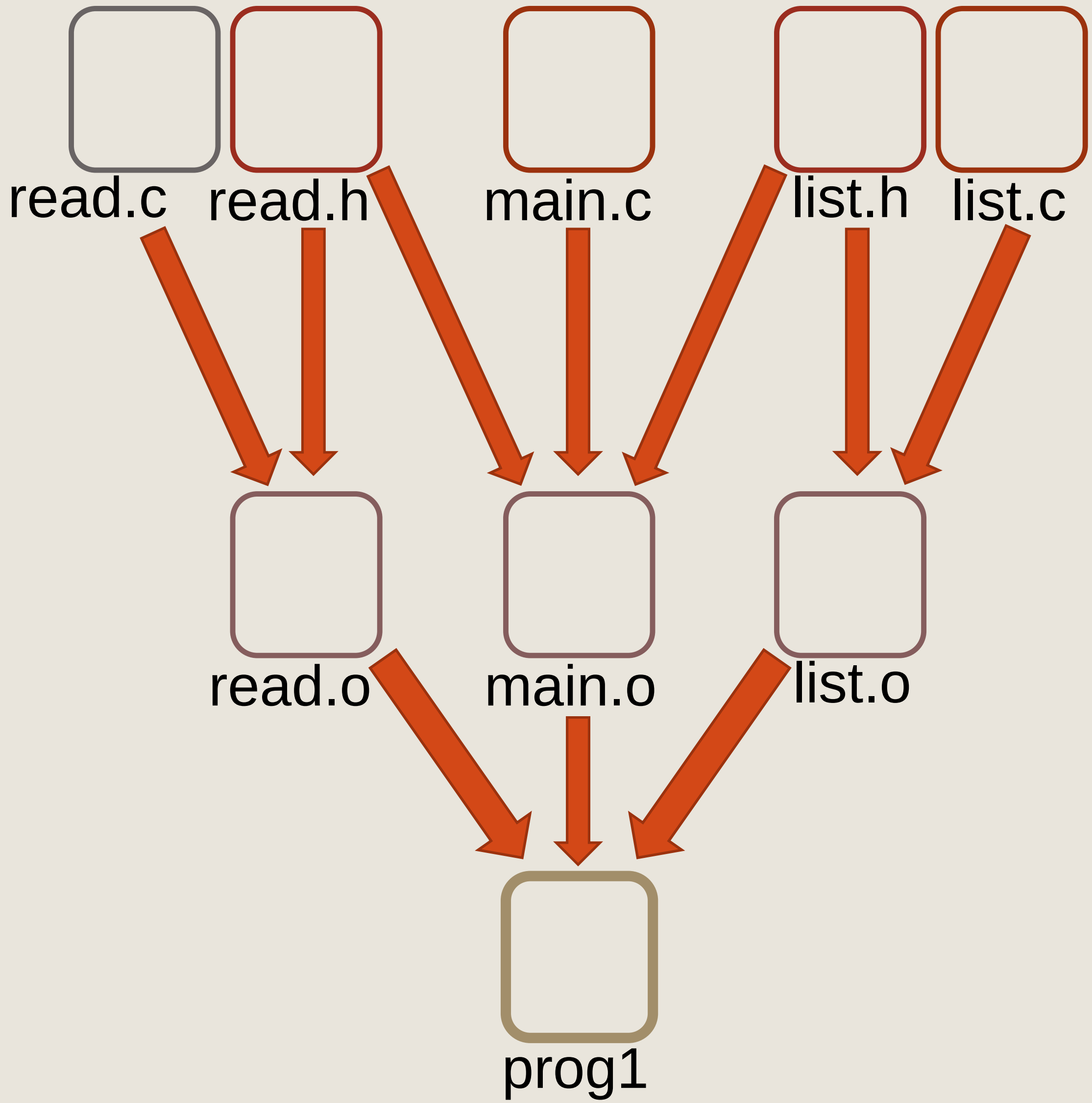


Linkage:

gcc read.o main.o list.o -o prog1



Dependencies Tree

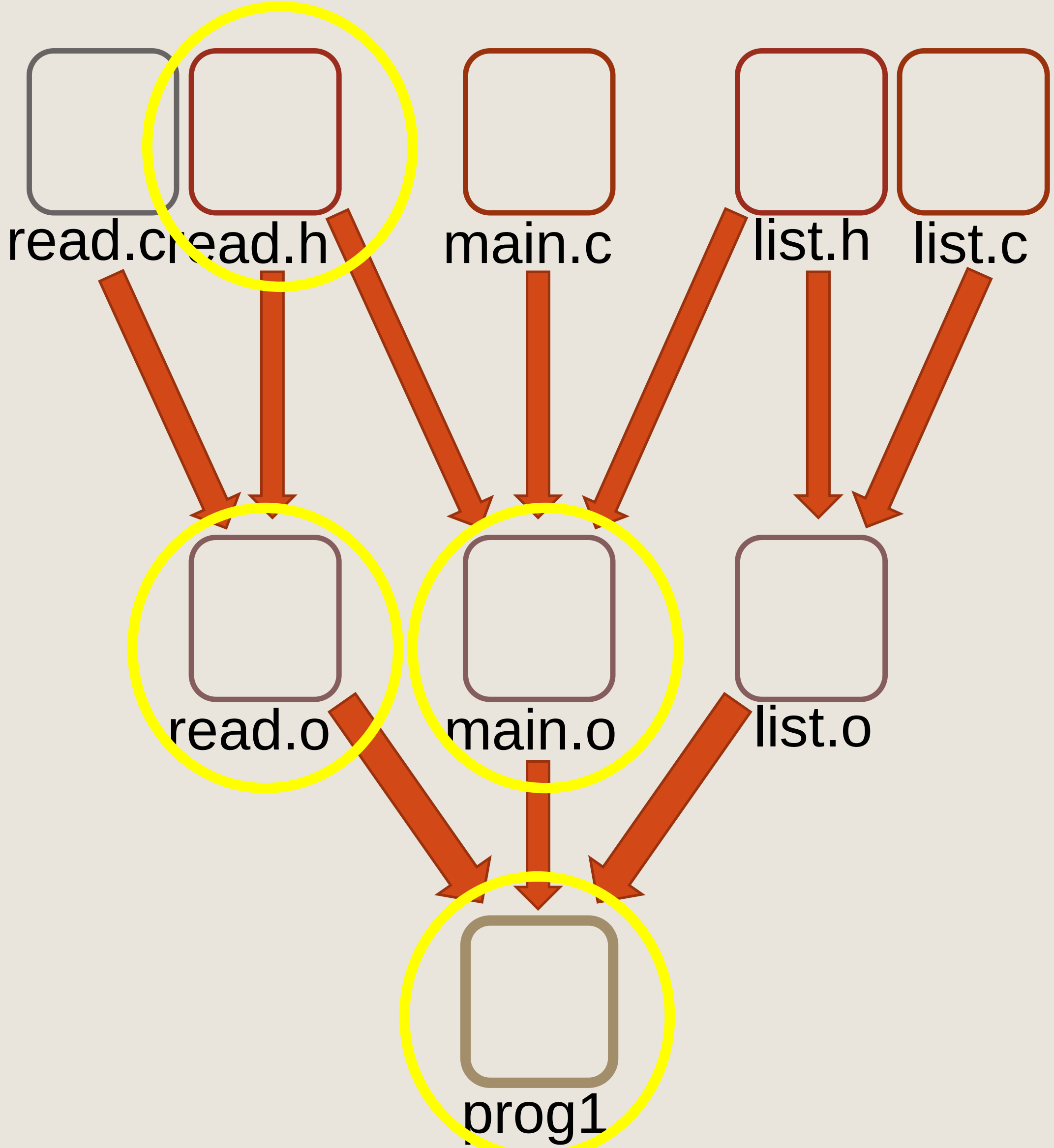


Compilation & linkage

If only one file is modified, will we have to recompile all over again?

- No!
- **Dependencies tree**

Example



Makefile

Aim: Build only out-of-date files (use timestamps)

Makefile contains:

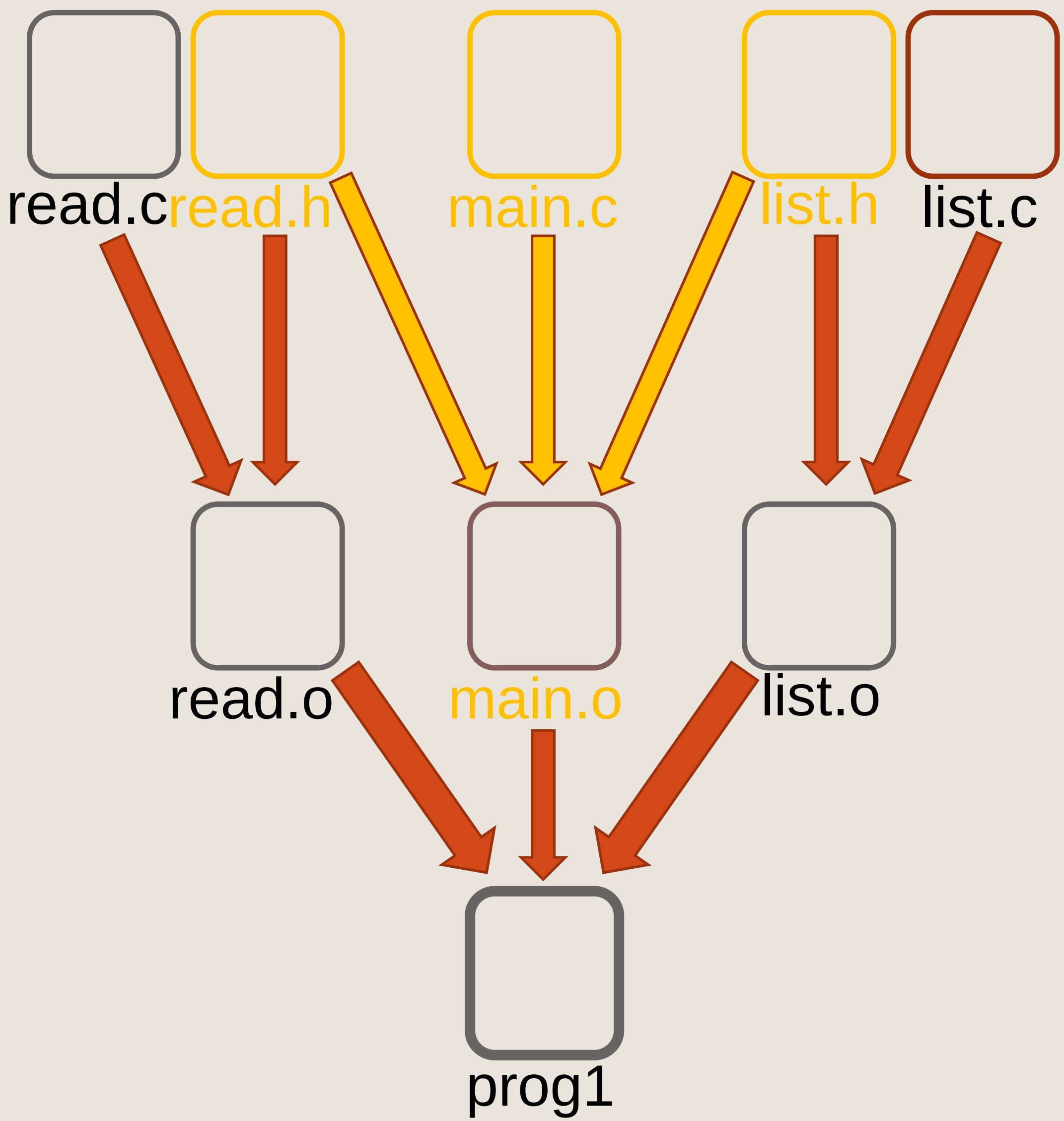
- List of dependencies (no cycles)
- "Recovery" scenario when any file is modified

```
main.o: main.c list.h read.h
	gcc -c main.c
```

Beware of the essential **tab**!

In words: if any of the files {main.c, list.h, read.h} was modified after main.o, the command "gcc -c main.c" will be performed

Dependencies Tree



Makefile

Format:

```
#comment
 target: dependencies
	[tab]    system command
	[tab]    system command
...
```

A very simple Makefile:

```
mywc: mywc.c
	gcc -Wall mywc.c -o mywc
```

Makefile - example

```
prog1: read.o main.o list.o
        gcc main.o read.o list.o –o prog1

main.o: main.c read.h list.h
        gcc -c main.c

read.o: read.c read.h
        gcc -c read.c

list.o: list.c list.h
        gcc -c list.c
```

Running make

- make prog1
- make main.o

How Make works?

Given a target:

1. Find the rule for it
2. Check if the rule prerequisites are up to date
   - By “recursive” application on each of them
3. If one of the prerequisites is newer than target run the command in rule body

- Use the flag <-n> to print the commands being performed
- Circular dependencies are dropped

Makefiles: macros

Macros are similar to variables

- Upper case by convention

Examples:

- OBJECTS = read.o list.o main.o

```
prog1: $(OBJECTS)
        gcc $(OBJECTS) -o  prog1
```

- CC=gcc

```
main.o: main.c
        $(CC) –Wall main.c
```

Implicit Rules

- When no explicit rule defined, an implicit rule will be used.
- Not always sufficient (e.g. doesn't check .h files update)

We saw "explicit rules" so far, e.g:

```
list.o: list.c list.h
        gcc -c list.c
```

Implicit rules (many kinds):

- Built-in - create ".o" files from ".c" files.

```
foo: foo.o bar.o
        gcc -o foo foo.o bar.o
```

No need to tell make to create o from c

- If we would like to write this explicitly (not needed!)

```
%.o : %.c
        gcc -c $< -o $@
```

$@ - file for which the match was made (e.g. list.o)
$< - the matched dependency (e.g. list.c)

More Automatic Variables (read at home)

Set automatically by Make, depend on current rule

`$@` - target

`$^` - list of all the prerequisites

- including the directories they were found

`$<` - first prerequisite in the prerequisite list.

- $< is often used when you want just the .c file in the command, while the prerequisites list contains many .h too

`$?` - all the prerequisites that are newer than the target

Many Others …….

Makefiles: Explicit/implicit rules

One more example for implicit rule:

```
%.class: %.java
        javac $<
```

- Result: For every “.java” file that was modified, a new “.class” file will be created.

Makefiles: all, clean, .PHONY

```
#Makes all progs
all: prog1, prog2
        shell command
...
# Removing the executables and object files
clean:
        rm prog1 prog2 *.o

# Not really a file name
.PHONY: clean
```

Any command (tar, diff…)

makefile names

make looks automatically for : `makefile, Makefile`

Override by using –f :

```
make –f MyMakefile
```

Using Wildcards

Automatic Wildcard (*,?) expansion in:

- Targets
- Prerequisites
- Commands

```
clean:
 rm -f *.o  # good

objects = *.o  # no good
```

Automatic makefiles

In most modern IDEs there is no need to write makefiles, the process is done for you automatically based on your project structure,

But…

In this course we want you to understand what's going on when compiling. So, **write your own makefiles**