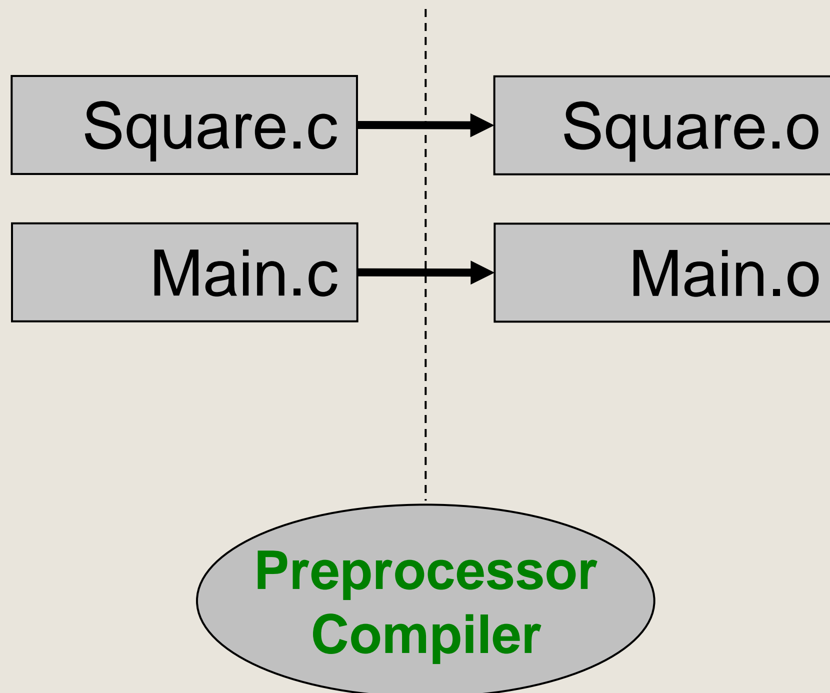
Revisiting building

Preprocessing + Compiling

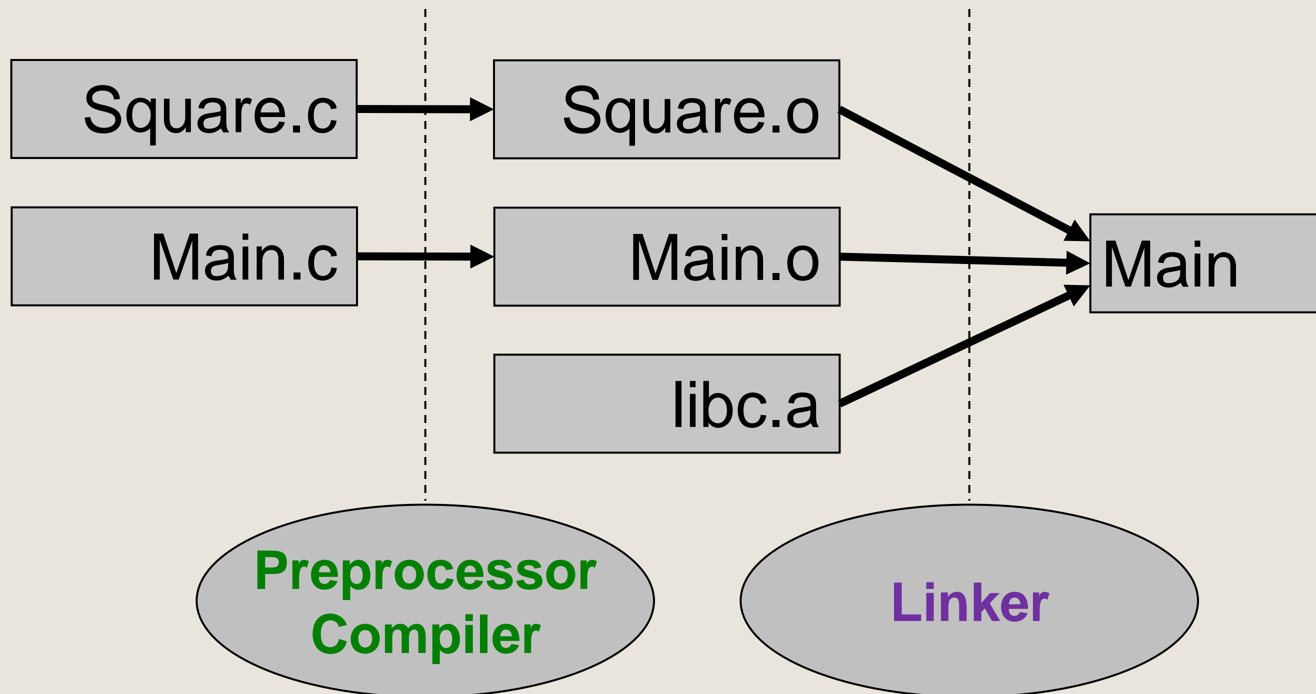
- Creates an object file for each code file (.c -> .o)
- Each .o file contains code of the functions and structs and global variables declarations
- Unresolved references still remain



Linking

Combines several object files into an executable file

- No unresolved references

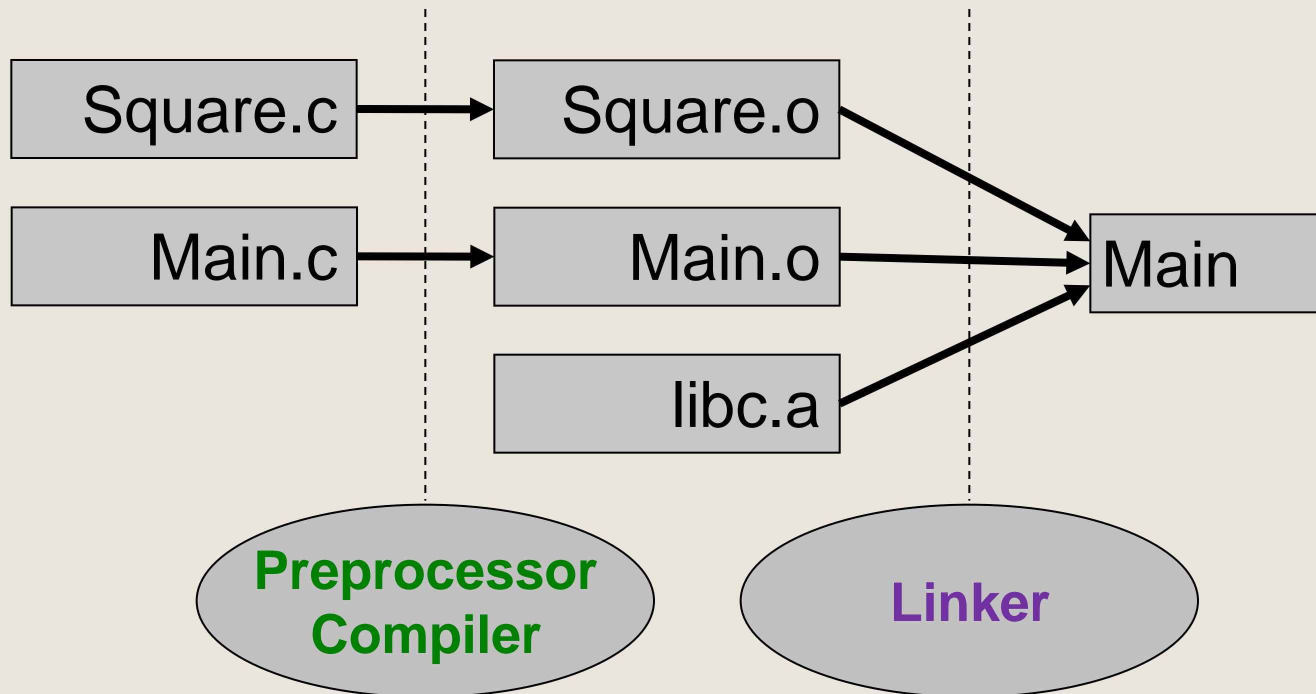


The whole process in linux (MSVS is doing the same process):

$ **gcc –c** Square.c –o Square.o

$ **gcc –c** Main.c –o Main.o

$ **gcc** Square.o Main.o –o Main

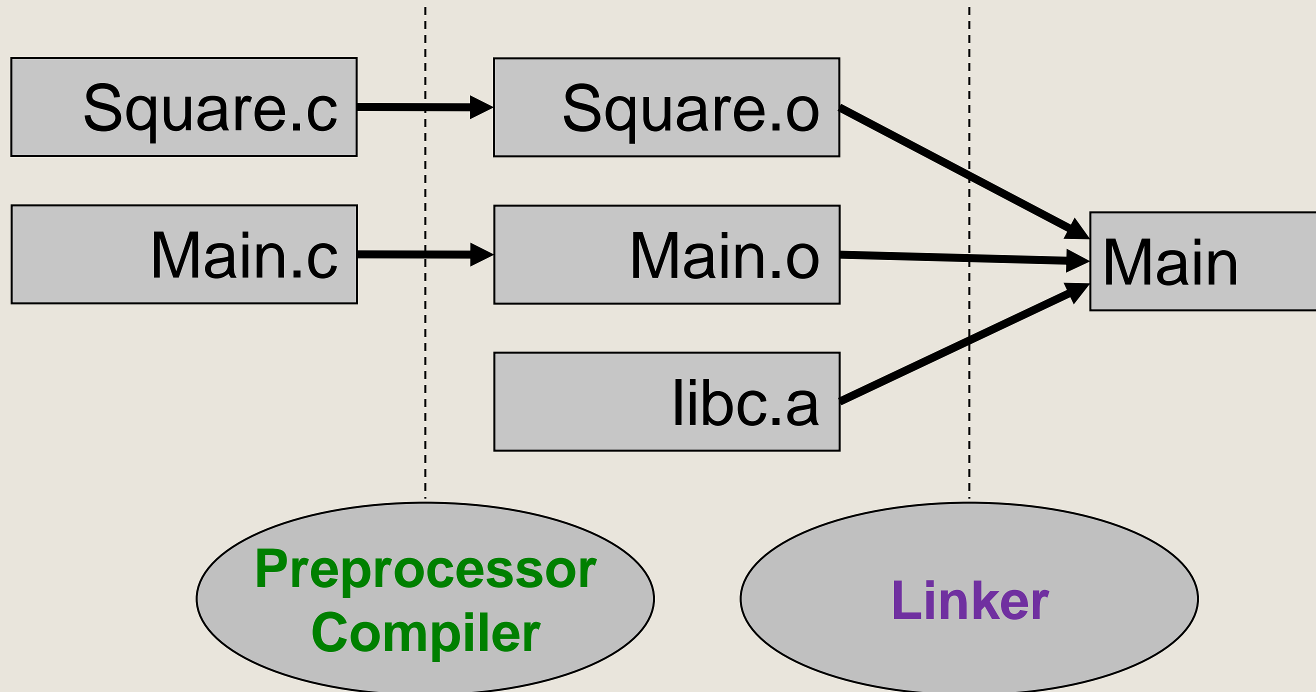


Do all files get compiled every time we build the program???

$ **gcc –c** Square.c –o Square.o

$ **gcc –c** Main.c –o Main.o

$ **gcc** Square.o Main.o –o Main



Do all files get compiled every time we build the program? – No!

Regular projects in Visual studio decide which files need to be compiled and which compiled files can be reused.

There are tools to write explicit or implicit buidling rules. For example, the ‘make’ tool which is used often in Linux.

The ‘make’ tool uses ‘Makefiles’. Many libraries of code include a Makefile.

Inter module scope rules

Visibility, duration and linkage

- **Translation unit** – a “.c” file + its included headers

- **Visibility** – the lines in the code where an object (variable, function, typedef) is accessible through a declaration.

Scopes - example

```
int func1( void );
int a = 3;
int b = 1;

void func2( )
{
  int a;
  b = 2;.
}
```

Visibility, duration and linkage

Object declaration scope: The maximal region the object can be visible 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.

Visibility, duration and linkage

Duration: 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 (remember, stack?).
- **Dynamic** – Until we free it.

Static variables in a function

Keep their value for the next call to the function.

That is, globals (duration is the entire program running time) with scope limited to the function.

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

Duration - example

```
int a; // all running time.
static int c; // all running time.

int func1( void ); // all running time.
static void func2() // all running time.
{

   int b; // until func2 ends.
   static int e; // all running time.

}
```

Visibility, duration and linkage

- **Linkage:** The mapping:

  Name (of variables / functions) ➔

  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.
    - All global objects declared as `static` have internal linkage.
  - **No linkage:** the object is unique to its scope
    - All locals have no linkage (unless declared with `extern`, next slide)

`extern` variables

- May be defined (defined=compiler allocates memory for it) outside the module

file1.c

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

file2.c

```c
extern int x; // should be imported
extern int y; // should be imported
int myFunc2()
{
   extern int z; // z from file1.c
   x = 5;
   y = 42;
}
```

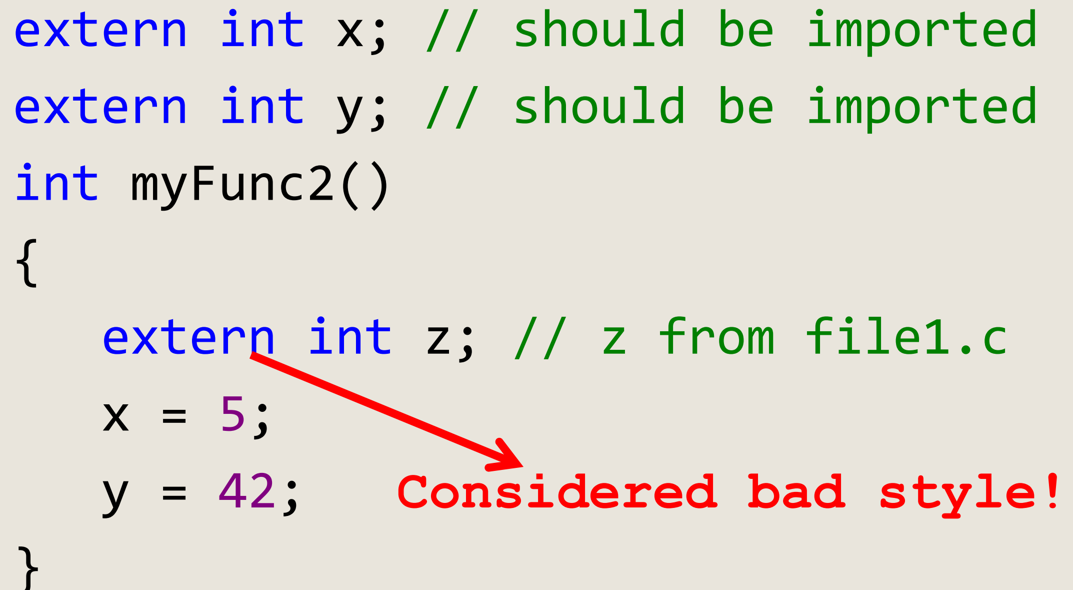
extern variables

- May be defined (defined=compiler allocates memory for it) outside the module

file1.c

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

file2.c



`extern` & `static` variables

`extern` variable

- May be defined (defined=compiler allocates memory for it) outside the module

`static` variable on the global scope

- Available only in the current module

file1.c

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

file2.c

```
extern int x; // should be imported
extern int y; // should be imported
int myFunc2()
{
    extern int z; // z from file1.c
    x = 5;
    y = 42; // link error!!!
}
```

Globals: try to avoid!

If you have to use globals, this is the right way

file.h

```c
extern int my_global;
```

file1.c

```c
#include “file.h”

// only once, init is good
int my_global= 11;
static int private_global;
myFunc1() {
  ++my_global;
  private_global= 33;
```

file2.c

```c
#include “file.h”

int myFunc2()
{
  my_global+= 3;
}
```

`static` functions on the global scope: 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();
}
```

extern functions? It is the default…

funcs.h :

```
// Both are the same
extern void Func1();
void Func2();
```

main.c:

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

Examples

```
// func1 has external linkage
int func1( void );

// func2 has internal linkage
// d has no linkage
static void func2( int d );
```

Examples

```
// a has external linkage and
// defined=created=memory is allocated)
int a;

// b has external linkage
extern int b;

// c has internal linkage and
// defined=created=memory is allocated)
static int c;
```

Examples

```
int a; // a has external linkage
       // and defined=created=memory is allocated
extern int b; // b has external linkage
static int c; // c has internal linkage and
              // defined=created=memory is allocated

int func1( void ) // func1 has external linkage
{
 int b = 2; // This b has no linkage and
            // hides the external b declared above
 static int e; // e has no linkage
 extern int c; // It is the same as ‘c’ above and
               // retains internal linkage
 extern int a; // It is the same as ‘a’ above and
               // retains external linkage
}
```

Examples

```
int a; // a has external linkage
       // and defined=created=memory is allocated
extern int b; // b has external linkage
static int c; // c has internal linkage and
              // defined=created=memory is allocated

int func1( void ) // func1 has external linkage
{
```

extern inside functions is considered bad style!!!

```
extern int c; // It is the same as ‘c’ above and
              // retains internal linkage
extern int a; // It is the same as ‘a’ above and
              // retains linkage
}
```

Error handling methods in C

The problem:

```
include <stdio.h>
void sophisticatedAlgorithm (char* name)
{
   FILE * fd = fopen (name); // using the file
                             // for an algorithm
   // ...
}
int main()
{
   char name[100];
   scanf ("%s", name);
   sophisticatedAlgorithm (name);
   // ...
}
```

OOP (java / C++) solution:

```
try
 {
    FileInputStream fstream = new
                   FileInputStream(name);
    DataInputStream in = new
                   DataInputStream(fstream);
    while (in.available() !=0)
    {
        System.out.println (in.readLine());
    }
    in.close();
 }
 catch (Exception e)
 {
   System.err.println("File input error");
 }
```

How can it be done in C?

Using assert (a short reminder):

```c
#include <assert.h>
// Sqrt(x) - compute square root of x
// Assumption: x non-negative
double Sqrt(double x )
{
    assert( x >= 0 ); // aborts if x < 0
    //...
}
```

If the program violates the condition, then

```
assertion "x >= 0" failed: file "Sqrt.c",
line 7 <exception>
```

Using assert:

- Terminates the program continuation.
- Good for debugging and logic examination.
- User can not decide what to do in case of error.

Return values:

```
#include <stdio.h>
int sophisticatedAlgorithm (char* name)
{
  FILE * fd = fopen (name);
  // indicate an abnormal termination of
  // the function
  if( fd == NULL ) return -1;
  ...
  // indicate a normal termination of
  // the function
  return 0;
}
```

Return values:

```
int main()
{
    int ret = sophisticatedAlgorithm(name);
    if(ret == -1)
    {
        // the exceptional case
    }
    else
    {
        // the normal case
    }
}
```

Return values:

- User can decide what to do in case of error.

But:

- Error handling and legitimate return values are mixed.
- Requires checking after each function call.

Using global variable – The standard library approach:

The idea: Separate between function return code and error description.

- Sometimes, functions return just 0 in case of success or -1 in case of error or some other binary output.
- A global variable holds the specific error code (or message) describes the occurred error.

Example:

```
#include <stdio.h> // for perror
#include <stdlib.h>
#include <errno.h> // for the global variable
                   // errno
#include <string.h> // for strerror
const char *FILE_NAME =
"/tmp/this_file_does_not_exist.yarly";
```

Example:

```
int main( int argc, char **argv )
{
   FILE * fp;
   fp = fopen( FILE_NAME,”r”);
   if( fp == NULL)
   {
      // Error
      perror( "Error opening file" );
      printf( "Error opening file: %s\n",
              strerror( errno ) );
   }
   return EXIT_SUCCESS;
}
```

Output:

```
Error opening file: No such file or directory
Error opening file: No such file or directory
```

Globals:

- User can decide what to do in case of error.
- Error handling and legitimate return values are separated.

But, still:

- Requires checking after each function call.

C exceptions:

google: C exceptions will lead to many useful C libraries that implement some kind of exceptions, very similar to java/c++