

Introduction to C

Programming Workshop in C (67316)

Fall 2018

Lecture 8

13.11.2018

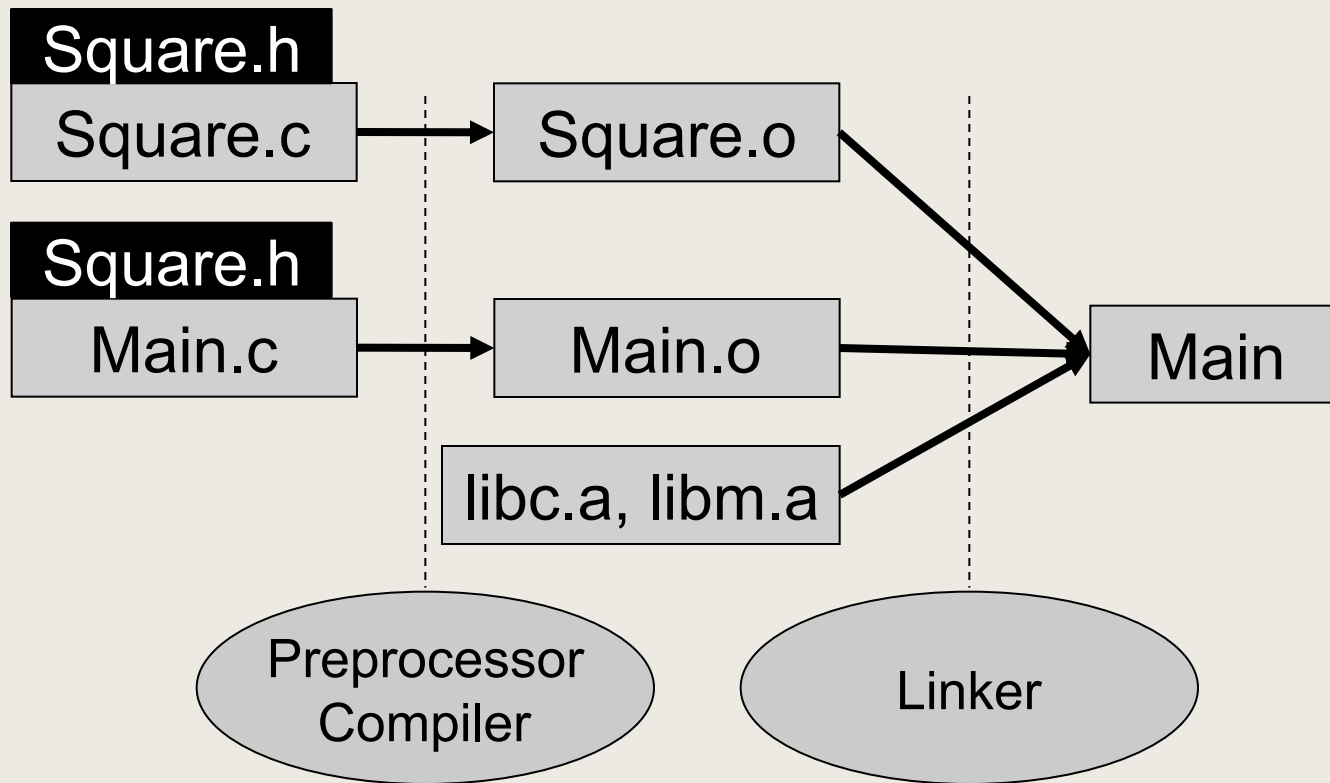
Multiple file project management

The whole process

```
$ gcc -c -Wall Square.c -o Square.o
```

```
$ gcc -c -Wall Main.c -o Main.o
```

```
$ gcc Square.o Main.o libc.a libm.a -o Main
```



Make

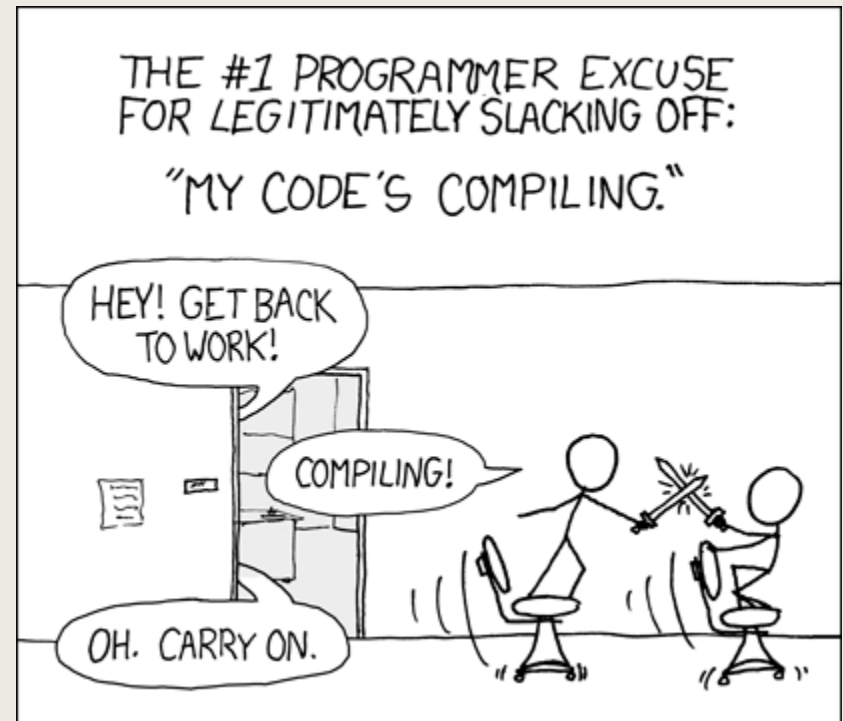
What is it?

- **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- Build only out-of-date files (use timestamps)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

```
// main.c  
#include "read.h"  
#include "list.h"  
...
```

```
// list.c  
#include "list.h"  
...
```

```
// read.c  
#include "read.h"  
...
```

```
// list.h  
...
```

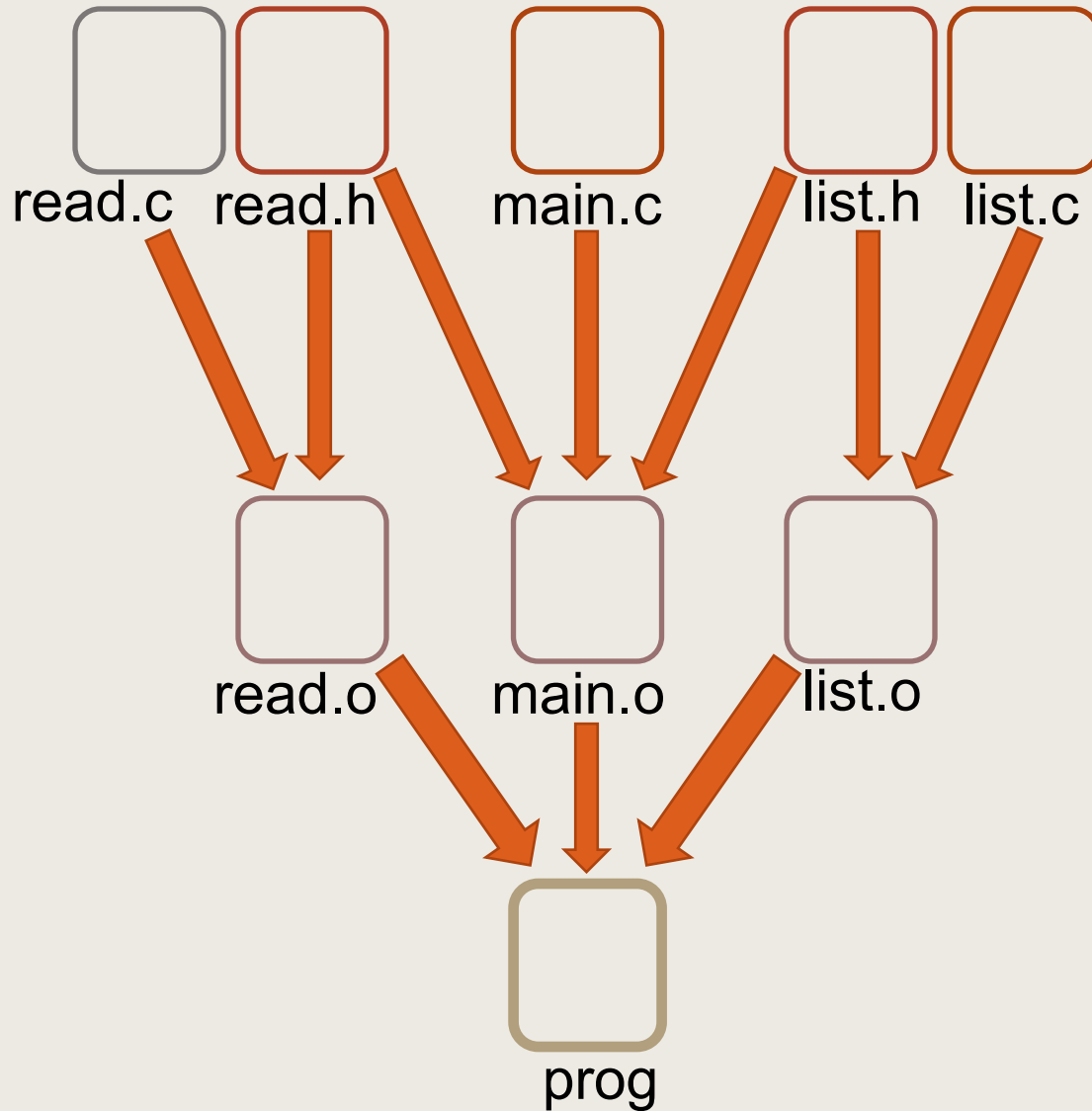
```
// read.h  
...
```

Compilation & linkage

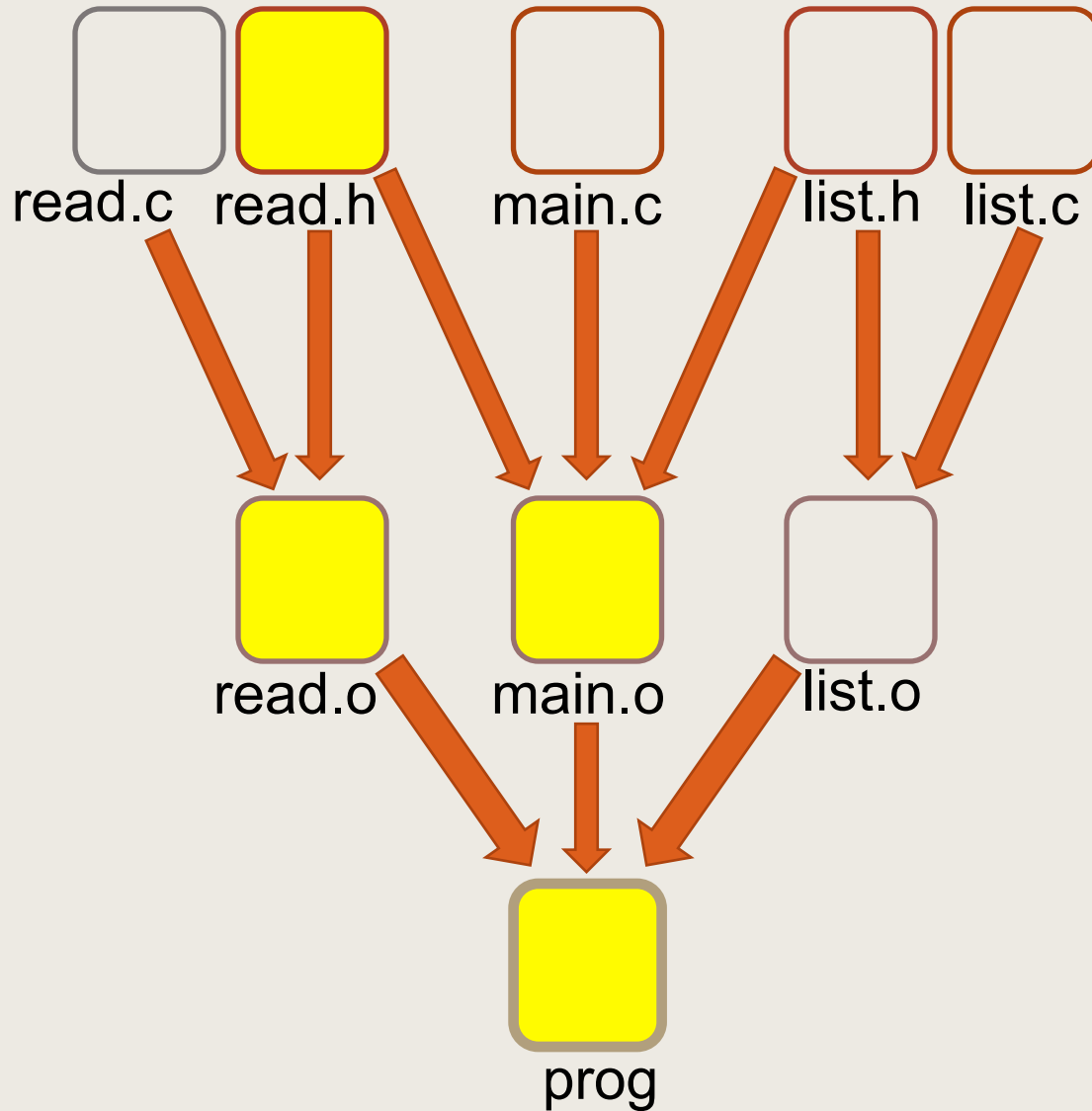
If only one file is modified, will we have to recompile all over again?

- No!
- **Dependencies tree**

Dependencies Tree



read.h change implication



Makefile

Aim: **build only out-of-date files**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

Format:


`#comment`

`target: dependencies`

`[tab] system command`

`[tab] system command`

`...`



Beware of the
essential **tab**!

Running make examples:

`> make prog`

`> make`

`> make -f myMakefile`

Modules & Header files

Square.h

```
// declaration
int area (int x1, int y1, int x2, int y2);
int length (int x1, int y1, int x2, int y2);
...
```

Square.c

```
#include "Square.h"
#include <math.h>
// implementation
int area (int x1,int y1,int x2, int y2){
    return length(x1,y1,x2,y1) *
           length(x1,y2,x1,y2);
}
...
```

MyProg.c

```
#include "Square.h"
int main()
{
    // usage
    area (2,3,5,6);
}
```

makefile names

make looks automatically for : `makefile`, `Makefile`

Override by using `-f` :
`make -f MyMakefile`

Makefile - version 1

A very simple Makefile

prog:

 gcc -Wall square.c main.c -o prog

Beware of the
essential **tab**!

This is what you would type to compile and
link the program

Makefile - version 2, macros

Macros are similar to variables
Upper case by convention

```
CC = gcc
```

```
CCFLAGS = -Wall
```

```
prog:
```

```
$(CC) $(CCFLAGS) square.c main.c -o prog
```

We still run the same terminal
command... because there are
no dependencies for prog

Makefile - Version 3 - using dependencies

```
CC = gcc
```

```
CCFLAGS = -Wall
```

```
prog: square.o main.o
```

```
    $(CC) square.o main.o -o prog
```

```
main.o: main.c square.c square.h
```

```
    $(CC) $(CCFLAGS) -c main.c
```

```
square.o: square.c square.h
```

```
    $(CC) $(CCFLAGS) -c square.c
```

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 (it does not execute the commands)

Implicit Rules

We saw “explicit rules” so far, e.g:

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

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

Implicit rules (many kinds):

- 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`)

Makefile - version 4, implicit rules

```
CC = gcc
```

```
CCFLAGS = -c -Wall
```

```
prog: square.o main.o
```

```
    $(CC) square.o main.o -o prog
```

```
%.o : %.c
```

```
    $(CC) $(CCFLAGS) $*.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 in

`$<` - 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

Makefile - version 5, one list of files

```
CC = gcc
```

```
CCFLAGS = -c -Wall
```

```
LDFLAGS = -lm
```

```
# User defined classes and modules. (no file suffixes)
```

```
CLASSES = square main
```

```
# Prepare object and source file list using pattern substitution  
func.
```

```
OBJS = $(patsubst %, %.o, $(CLASSES))
```

```
SRCS = $(patsubst %, %.c, $(CLASSES))
```

```
prog: $(OBJS)
```

```
    $(CC) $(OBJS) $(LDFLAGS) -o prog
```

```
%.o : %.c
```

```
    $(CC) $(CCFLAGS) $*.c
```

Makefiles: all, clean, .PHONY

A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request

```
# Makes all progs
```

```
all: prog1, prog2
```

```
    shell_command
```



Any command
(tar, diff...)

```
...
```

```
# Removing the executables and object files
```

```
clean:
```

```
    rm prog1 prog2 *.o
```

```
backup:
```

```
    zip backup.zip *.h *.cc
```

```
# Not really a file name
```

```
.PHONY: all, clean, backup
```

Using Wildcards

Automatic Wildcard (*,?) expansion in:

- Targets
- Prerequisites
- Commands

clean:

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

```
objects = *.o # not good
```

Automatic makefiles (cmake)

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

But...

- You often need to modify cmake, such as add library
- For this purpose, you need to understand what's going on when compiling
- So, practice **writing your own makefiles**

Libraries

http://www.adp-gmbh.ch/cpp/gcc/create_lib.html

Libraries

A library is a collection of functions that you may want to use

- written and compiled by you
- or by someone else

Examples:

- C's standard libraries
- Math library
- Graphic libraries

Libraries may include many different object files

Two Kinds of Libraries

Static libraries:

- linked with your executable at compilation time
- standard unix suffix: .a (windows: .lib)

Shared libraries:

- loaded by the executable at run-time
- standard unix suffix: .so (windows: .dll)



static vs. shared

Static libraries pros:

1. Independent of the presence/location of the libraries
2. Independent of the versions of the libraries
3. Less linking overhead on run-time

Shared libraries pros:

1. Smaller executables
2. Multiple processes share the code
3. No need to re-compile executable when libraries are changed
4. The same executable can run with different libraries

Static Libraries

Using a utilities library

```
#include <utils.h> // Library header
int main ()
{
    foo(); // foo is a function of the
           // 'utils' library
    ...
}
```

Compiling static with libraries

Path to the library
header files
usually in CCFLAGS

Compilation:

```
gcc -Wall -c -I /usr/lib/include/  
main.c -o main.o
```

Path to the library file
usually in LDFLAGS

Linking:

```
gcc main.o -L /usr/lib/bin/  
-lutils -o app
```

library name:
libutils.a

Static libraries – creating your own

Creating the library **libutils.a**:

```
ar rcs libutils.a data.o stack.o list.o
```

- **ar** is like tar – archive of object files
- **rcs** are 3 relevant flags (read ‘ar’ man pages), of which ‘s’ indicates: create ‘symbol-table’ for the linker

Using (linking with) the static library **libutils.a**:

```
gcc main.o another.o -L. -lutils -o prog
```

This links to the code in **libutils.a**

Libraries in makefile

```
LIBOBJECTS = data.o stack.o list.o
```

```
libutils.a: ${LIBOBJECTS}
```

```
    ar rcs libutils.a ${LIBOBJECTS}
```

```
...
```

```
OBJECTS = main.o another.o
```

```
CC = gcc
```

```
prog: ${OBJECTS} libutils.a
```

```
    ${CC} ${OBJECTS} -L. -lutils -o prog
```

Order is important – put libraries at the end

```
gcc main.o -L. -lutils driver.o -o app
```

- The linker links the library files from left to right
- If driver tries to use references from `libutils.a` they may not be linked!
This is because unused library references may be dropped at linkage time
- This is also true for dependencies between different libraries*

* there is a way to resolve cyclic dependency

The linking process: objects vs. static libraries

Objects:

- The entire object file is linked to the executable, even when its functions are not used
- Two function implementations – will cause error

Libraries:

- Just symbols (functions) which are not found in the obj files are linked
- Two function implementations – first in the obj file, and second in library – the first will be used
- Order is important – compiler may discard unused references when linking the library

Dynamic Libraries

Dynamic Libraries

Dynamic library compilation and creation:

```
gcc -Wall -fPIC -c utils.c
```

```
gcc -shared utils.o -o libutils.so
```

- PIC – position independent code
- On windows you will need to use in your code:
__declspec(dllimport) and
__declspec(dllexport)

(feel free to read more about it 😊)

Dynamic Libraries

Usage: (linking to...)

```
gcc -Wall main.c -L. -lutils
```

You will need to set the environment variable

LD_LIBRARY_PATH=.

so the shared library can be found at runtime, unless it is in the system's path (the already set value)

Dynamic Libraries

Why do we need to set the dynamically linked library at link (compile) time?

- The **header file** is used to define the prototypes which are required to **compile** the code which uses the library
- The linker will just **check that linking is possible at link time** (make sure the functions it needs are actually in the linked library or it will raise an error)

Dynamic Libraries

Why do we need to set the dynamically linked library at link (compile) time?

- The **actual linking is done during run time**, so it will check where the functions are again, as the library itself might have changed since the program was compiled
- For this, it needs to **know how to find the library at runtime** (should be in the dynamic library search path, e.g. c:\windows\system, or defined in **LD_LIBRARY_PATH**)

Static variables

Static variables

- **static** keyword has two meanings, depending on where it is declared
- **Outside a function**, **static** variables/functions only visible within that file, not globally. The static limits the scope to the file.
- **Inside a function**, **static** variables
 - are local to that function
 - are **initialized only once** during program initialization
 - do not get reinitialized with each function call
 - provide private permanent storage within a single function

Static variables **outside** a function

```
static char buffer[BUFFER_SIZE];  
static int counter;  
int storeInBuffer()  
{  
    ...  
}  
int getFromBuffer()  
{  
    ...  
}
```

Static variables **inside** a function

```
int getUniqueID()
{
    static int id = 0; // called once
    id++;
    return id;
}

int main()
{
    int i = getUniqueID(); // i = 1
    int j = getUniqueID(); // j = 2
}
```

Static variables in a function

- ❖ Static variable duration is the entire program running time
- ❖ Static variable visibility is the scope where it's declared
- ❖ Static variables in a function keep their value for the next call to the function
- ❖ Memory is allocated on global space (initialized and uninitialized data)
- ❖ Static variables (like global variables) are initialized as 0 if not initialized explicitly
- ❖ Static variables can only be initialized using constant literals



Inter module scope rules

[module] = [translation unit] = [.c file]

Programming modules in C

- C programs do not need to be monolithic
- Module: interface and implementation
 - interface: header files
 - implementation: auxiliary source/object files
- Same concept carries over to external libraries

Modules & Header files

Square.h

```
// declaration
int area (int x1, int y1, int x2, int y2);
int length (int x1, int y1, int x2, int y2);
...
```

Square.c

```
#include "Square.h"
#include <math.h>
// definition/implementation
int area (int x1,int y1,int x2, int y2){
    return length(x1,y1,x2,y1) *
           length(x1,y2,x1,y2);
}
...
```

MyProg.c

```
#include "Square.h"
int main()
{
    // usage
    area (2,3,5,6);
}
```


Function declaration vs. definition

Square.h

Declaration

```
int area (int x1, int y1, int x2, int y2);  
int length (int x1, int y1, int x2, int y2);  
...
```

Square.c

Definition

```
#include "Square.h"  
  
int area (int x1, int y1, int x2, int y2){  
    return length(x1,y1,x2,y1) *  
           length(x1,y2,x1,y2);  
}  
...
```

Function declaration vs. definition

- Declaration of a function → the function exists somewhere in the program but the memory is not allocated for them
 - the program knows what are the arguments to the function, its data types, the order of arguments and the return type
- Definition of a function results in memory allocation

- What about variables?
- How would you declare a C variable without defining it?

Variable declaration vs. definition

Square.h

Declaration

```
int area (int x1, int y1, int x2, int y2);  
int length (int x1, int y1, int x2, int y2);  
extern int var;
```

Square.c

Definition

```
#include "Square.h"  
int var;  
int area (int x1, int y1, int x2, int y2){  
    return length(x1,y1,x2,y1) *  
           length(x1,y2,x1,y2);  
}  
...
```

Extern

- extends the visibility to the whole program
- functions declarations and definitions include extern by default:

```
int foo(int arg1, char arg2);  
extern int foo(int arg1, char arg2);
```

```
extern int var;  
int main(void)  
{  
    var = 10;  
    return 0;  
}
```

var declared but not
defined → linker
error

Static and extern variables

static variable on the global scope

- Available only in the current module

extern variable

- May be defined outside the module

file1.c

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

file2.c

```
extern int y; // import y (from file1.c )  
extern int x; // import x (from file1.c)  
int myFunc2() {  
    extern int z; // import z (from file1.c)  
    y = 5;  
    x = 3; // linker error  
    // x visible only in file1.c (static)  
}
```

Static functions

static function available only in current module

funcs.h :

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

funcs.c :

```
static void Func1()  
{  
    ...  
}
```

main.c:

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

Visibility and duration

❑ 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

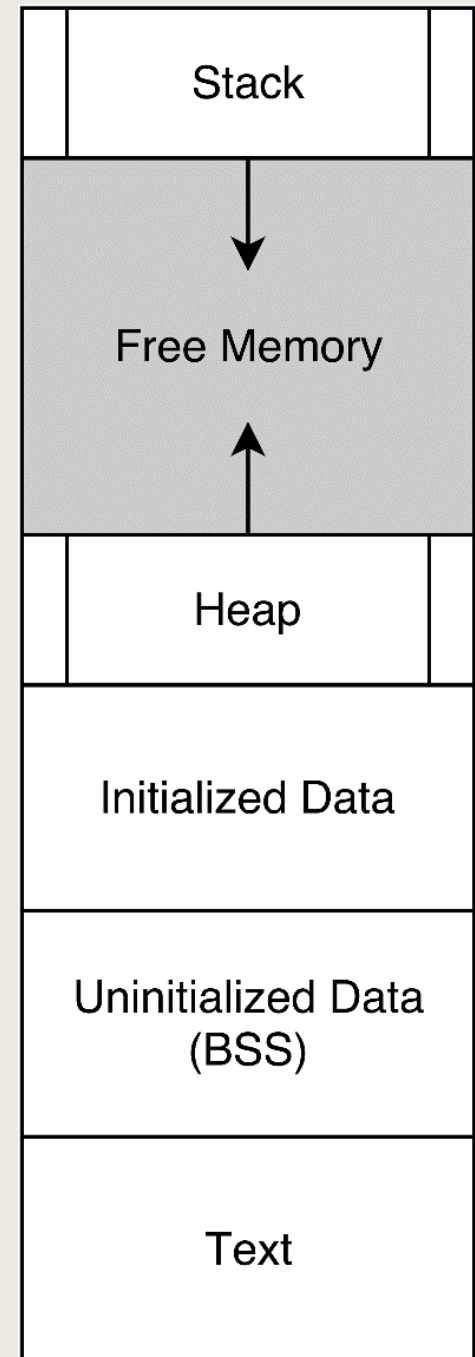
- **Global** declaration – visible throughout the translation unit, starting from the declaration location
- **Local** declarations (inside {}) – visible in their block (starting from the declaration)
- Can be hidden by inner scope declarations
- Scope is related to the compiler, not the linker

Visibility, duration and linkage

❑ Duration (a.k.a Lifetime)

the amount of time that it is guaranteed that the memory for an object is allocated (accessible)

- **Functions** - the entire running time of the program
- **Globals** - the entire running time of the program
- **Locals** – until their scope ends
- **Dynamic** – we control it
- **Static** – next slides



Duration - example

```
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
}
```

And what about visibility?

Visibility, duration and linkage

Linkage

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

- It associates the name with an entity **OUTSIDE** its block
- Variables with no linkage are only accessible from within the same block

Visibility, duration and linkage

❑ External linkage

- ❑ Names are associated with the same object throughout the program, **across translation units**
- 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

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
}
```

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

- Enumeration (or enum) is a user defined data type in C
- Used to assign names to integral constants
- 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
```

- Are an alternative to #define

enum

- enum names can have the same value

```
#include <stdio.h>
enum State {Working = 1, Failed = 0, Freezed = 0};

int main()
{
    printf("%d, %d, %d", Working, Failed, Freezed);
    return 0;
}
```

- by default the compiler assigns values starting from 0
- we can assign values to some name in any order. The unassigned names will get the value of a previous name plus one



enum vs. macro

- we can also use macros to define constants

```
#define Working 0  
#define Failed 1  
#define Freezed 2
```

// alternatively with enum

```
enum State {Working, Failed, Freezed};
```

- enum follow scope rules
- the values are assigned automatically

macro vs. const vs. enum for constants definition

```
#define BUFFER_SIZE 256  
const int buffer_size = 256;  
enum { buffer_size = 256 };
```

- in principle all can be used, the preference is personal
- C arrays can only be initialized using macro defined constants
- this is the major reason for a recommendation to use macros in C and const variables in C++ (not an issue with VLAs)
- const variables are typed, properly scoped, can take an address
- #define is in global scope, prone to conflicting usages
- enum is properly typed (integers only) and scoped, initialized by default, can't take address
- more here:
<https://stackoverflow.com/questions/1637332/static-const-vs-define> and
here: <https://stackoverflow.com/questions/1674032/static-const-vs-define-vs-enum>

switch

switch statement

- substitute long if statements that compare a variable to integer value

```
switch (n)
{
    case 1: // code to be executed if n = 1;
        break;
    case 2: // code to be executed if n = 2;
        break;
    default: // code to be executed if n doesn't match any cases
}
```

- the expression has to be constant value:

```
switch (1+2+3) // good
switch (a+b) // not good
```

switch statement (cont.)

- duplicate case values are not allowed
- the default statement is optional and can be placed anywhere. it will be still executed if no match is found
- the break statement is optional
- all statements following a matching case execute until a break is reached
- nesting is allowed



enum and switch go together

```
int main() {  
    enum Color {BLACK,RED,GREEN,BLUE};  
  
    enum Color my_color;  
    scanf("%d", &my_color);  
  
    switch (my_color)  
    {  
        case RED: printf("your favorite color is Red");  
                break;  
        case GREEN: printf("your favorite color is Green");  
                break;  
        default: printf("you did not choose any color");  
    }  
    return 0;  
}
```