Lecture Notes for Lecture 12 of CS 5001 (Foundations of CS) for the Fall, 2018 session at the Northeastern University Silicon Valley Campus.

Modularizing C Programs Into Separate Compilation Units

Philip Gust,
Clinical Instructor
Department of Computer Science

Lecture 11 Review

- C programs that are loaded into memory have separate memory segments for different types of information
- Segments include text (program code), constants, global and static variables, dynamic memory, and local variables.
- Dynamic memory segment ("heap") is a pool of memory that programmers can allocate and use.
- Programmer determines lifetime of allocated heap storage, requesting it when needed, and freeing it afterwards.
- Allocated memory can be for any kind of data, including basic types, pointers, structs, and arrays.
- Heap functions malloc(), free(), and realloc() allocate a block of memory, free it for future use, and resize an existing block.

Separate Compilation Units

- In this lecture, we will learn about how to modularize program functionality into multiple files to make code easier to create and manage.
- In many cases, each file contain the data definitions and functions for a single data type, with the main function and associated logic in their own files.
- We will also learn about the concept of sharing declarations for the the functions in a given file with functions in other files using external storage declarations.
- Finally we will see how to put declarations for data types and external declarations into their own files and include them in files that require them using the C pre-processor.

Separate Compilation Units

Why Divide Programs Into Separate Files?

- There is a practical limit on the amount of code that can be effectively managed as a single C program file.
- Putting groups of data types (structs, typedefs, enums, constants, variables, and functions) in separate files makes it easier to share and reuse them.
- Having separate files makes it possible for multiple developers to work on different parts of the same program.
- Separating unit tests into their own files allows the test code to be managed separately from the code being tested.
- Dividing functionality into separate files also makes code easier to maintain: only need to modify subset of files.

Separate Compilation Units

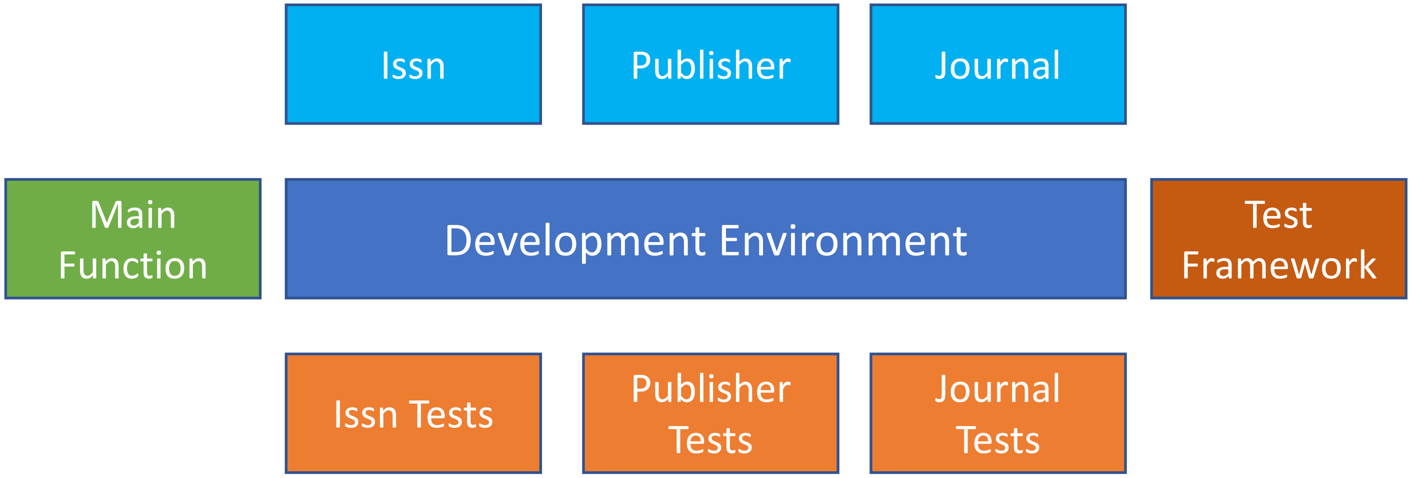
Example: Publication Data Types

- We will look at how to divide the publication related structs from earlier examples into separate compilation units.
- Here are some candidate data types to maintain in separate files:
    - Issn
    - Publisher
    - Journal
    - Unit tests
    - Main program

Separate Compilation Units

Example: Publication Data Types

- Development of application data types and unit tests.



Separate Compilation Units

Sharing Code and Declarations

- Suppose that we separate the data types into files as follows:
    - All Issn related information into the a file “issn.c”,
    - All Publisher related information into a file named “publisher.c”,
    - All Journal related information into a file “journal.c”
    - All test functions into files “issn_test.c”, “publisher_test.c” and “journal_test.c”
    - The main function into a file “main.c”

Separate Compilation Units

Sharing Code and Declarations

- When compiling "journal.c" the compiler will report errors because Publisher and Issn are unknown.
- When compiling "publisher.c" the compiler will report errors because Journal is unknown.
- When compiling test functions, the compiler will report errors because the type(s) being tested are unknown.
- When compiling the main function, the compiler will report errors because the test functions it calls are unknown.

Separate Compilation Units

Sharing Code and Declarations

- When all the code and declarations are in a single C file, the compiler has all the information it requires to translate the source code and create an executable program.
- If the code is separated into multiple files, a way is needed to make declarations for functions and data in one file available to functions in files that refer to them.
- C provides a way to share declarations for functions and data in one file available to functions in files that refer to them using *include files* and the C pre-processor.

Separate Compilation Units

Include Files and the C Preprocessor

- The C preprocess is a phase that the C complier that runs before the source code is actually compiled.
- The purpose is to perform text inclusions, substitutions, and conditional processing on the C source code that customizes the source code prior to compilation.

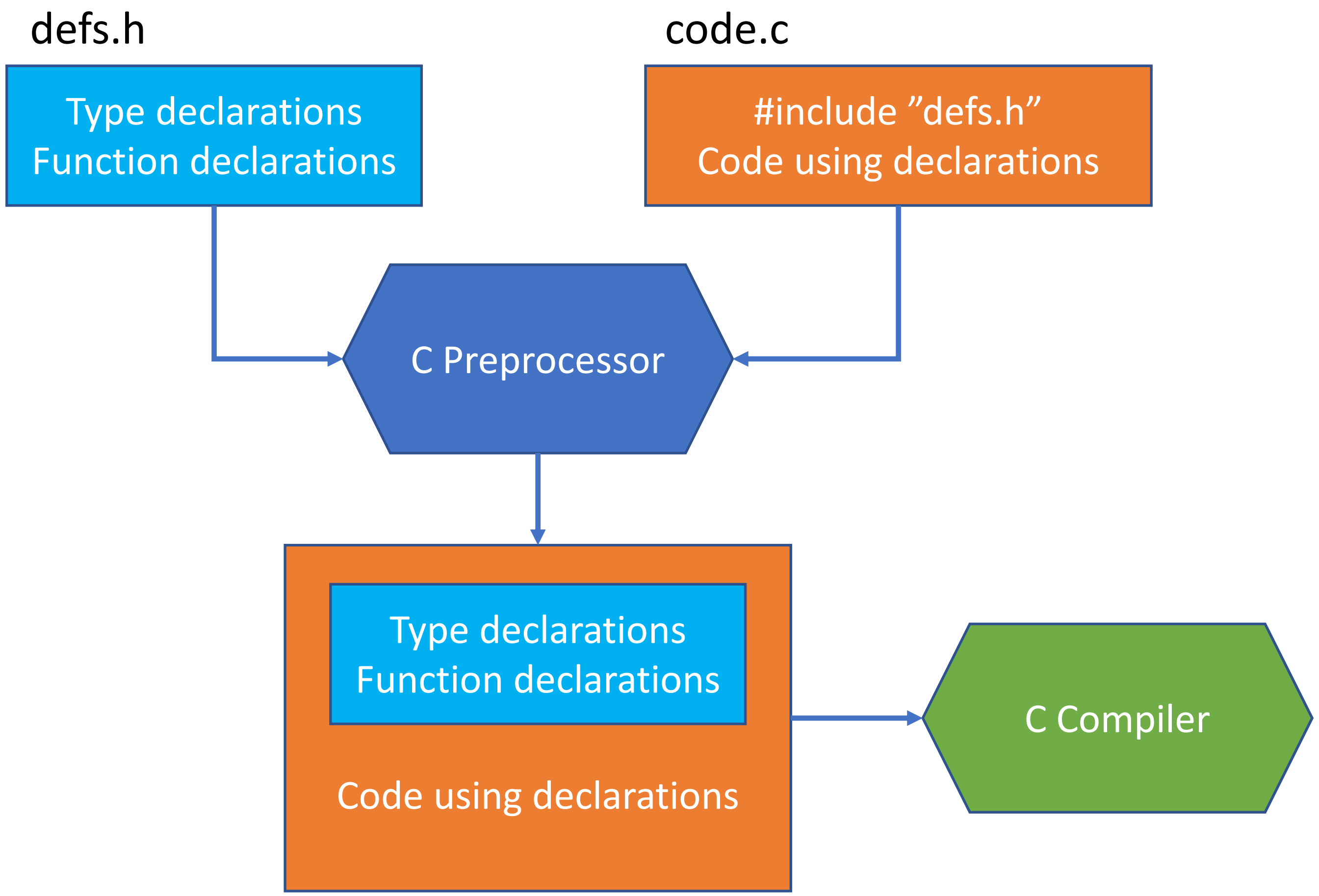
Separate Compilation Units

Include Files and the C Preprocessor

- One of those steps is to textually “include” external files that contain needed type, function, and variable declarations at the point where an *#include* preprocessor directive occurs.
- This allows the declarations to be written once, and included in multiple C files, instead of having to enter them for each C file that requires them.
- Include files traditionally have the file extension ”.h”.

Separate Compilation Units

Include Files and the C Preprocessor



Separate Compilation Units

Sharing Code and Declarations

- For Issn, we create two files:
  - **issn.h** contains the type declaration for Issn, and declarations (but not their definitions) for Issn functions and any variables
  - **issn.c** contains the definitions for the functions and any variables that declared in issn.h. It also has a #include “issn.h” preprocessor directive that includes the declarations in this file.

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Header File* command to create **issn.h**:

```
#ifndef ISSN_H
#define ISSN_H

/** Represents an 8-digit journal id: nnnnnnnC */
typedef uint32_t Issn;

/** Sentinel value used when the ISSN is unknown */
extern const Issn ISSN_UNKNOWN;
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Header File* command to create **issn.h**:

```
/**
 * Get string representation of ISSN.
 * @param issn the ISSN value
 * @param issnChars array to receive the ISSN characters
 * @return pointer to ISSN chars as nnnn-nnnC
 */
char *issnToString(Issn issn, char issnChars[]);
/**
 * Parse a string representation of an Issn
 * @param issnChars characters of ISSN as nnnn-nnnC
 * @return issn or ISSN_UNKNOWN if parse failed
 */
Issn parseIssn(const char issnChars[]);

#endif
```

Separate Compilation Units

Sharing Code and Declarations

- The C compiler supports defining preprocessor constants, variables and functions used only while the C preprocessor is working on a file, and do not appear in the source code sent to the C compiler.
- The C preprocessor also supports conditional processing of input text based on the value of preprocessor constants and variables.
- In the previous example, these preprocessor directives ensure that the file is included only once by a C source file or another file that is also being included:

```
#ifndef ISSN_H
#define ISSN_H

….

#endif
```

Separate Compilation Units

Sharing Code and Declarations

- C separates the declaration of a function or variable from its definition. This statement declares the function issToString() but does not define it.

```
char *issnToString(Issn issn, char issnChars []);
```

- Declaring a variable without defining it requires the use of the keyword *extern*. This statement declares the global variable ISSN_UNKNOWN, does not define it:

```
extern const Issn ISSN_UNKNOWN;
```

- All declared functions and variables must still be defined once in some compiled file.
- The use of the extern keyword is optional for functions but is required for variables to distinguish between their declaration and their definition.

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Source File* command to create **issn.c**:

```
#include ”issn.h”
/** Sentinel value used when the ISSN is unknown */
const Issn ISSN_UNKNOWN = 0;       // value is not a valid ISSN

/**
  * Get string representation of ISSN
  * @param issn the ISSN value
  * @param issnChars array to receive the ISSN characters (OUT)
  * @return pointer to ISSN chars
  */
char *issnToString(Issn issn, char issnChars []) {  // char  *issnChars
     // make use of underlying uint32_t type of ISSN internally
     sprintf(issnChars, “%04x-%04x”,  issn >> 16, issn & %0xFFFF);
     if (issnChars[8] == 'a') {
               issnChars[8] = 'X';  // issn uses ‘X’ rather than ‘a’ for 10
     }
     return issnChars;
}
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Source File* command to create **issn.c**:

```
/** Parse string representation of Issn.
 * @param issnChars characters of ISSN as nnnn-nnnC
 * @param issnChars array to receive the ISSN characters
 * @return issn or ISSN_UNKNOWN if parse failed
 */
Issn parseIssn(const char issnChars[]) {
    unsigned d[8];
    int n = sscanf(issnChars, "%1u%1u%1u%1u-%1u%1u%1u%1u",
                                &d[0],&d[1],&d[2],&d[3],&d[4],&d[5],&d[6],&d[7]);
    if ((n == 7) && (issnChars[8] == 'X')) {
        d[n++] = 0xa;
    }
    Issn issn = 0;
    if (n == 8) {
        for (int i = 0; i < 8; i++) {
            issn = issn<<4 | d[i];
        }
    }
    return issn;
}
```

Separate Compilation Units

Sharing Code and Declarations

- We can do the same for Publisher and Journal:
  - **publisher.h** contains the type declaration for Publisher, and declarations (but not their definitions) for Publisher functions
  - **publisher.c** contains the definitions for the functions that were declared in publisher.h. It also has a #include “publisher.h” preprocessor directive that includes the declarations in this file.
  - **journal.h** contains the type declaration for Journal, and declarations (but not their definitions) for Journalfunctions
  - **journal.c** contains the definitions for the functions that were declared in journal.h. It also has a #include “journal.h” preprocessor directive that includes the declarations in this file.

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Header File* command to create **publisher.h**:

```
#ifndef PUBLISHER_H
#define PUBLISHER_H
#include "journal.h"

/** Struct that defines a Journal */
typedef struct {            // “anonymous struct”
    char name[100];                     // name of publisher
    Journal **journals;             // array of journal pointers
    unsigned int nJournals;         // number of journals
    unsigned int maxJournals;       // maximum number of journals
} Publisher;                // only known by its typedef name
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New* -> *Header File* command to create **publisher.h**:

```
/**
 * Create and initialize a publisher.
 * @param name the name of the publisher
 * @return the Publisher
 */
Publisher *newPublisher (const char *name);

/**
 * Delete the publisher.
 * @param publisher the publisher to delete
 */
void deletePublisher(Publisher *publisher);
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New* -> *Header File* command to create **publisher.h**:

```
/**
 * Add a journal to a publisher.
 * @param publisher the publisher
 * @param journal the journal to add
 */
void addJournalToPublisher(Publisher *publisher, Journal *journal);

/**
  * Print a Publisher only without its journals.
  * @param publisher the publisher to print
  */
void printPublisherOnly(const Publisher *publisher);
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New* -> *Header File* command to create **publisher.h**:

```
/**
  * Print a Publisher and its journals.
  * @param publisher the publisher to print
  */
void printPublisher(const Publisher *publisher);

#endif  /* PUBLISHER_H_ */
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Header File* command to create **journal.h**:

```
#ifndef JOURNAL_H
#define JOURNAL_H
#include “issn.h”
#include “publisher.h”
/** Struct that defines a Journal */
typedef struct Journal {// “anonymous struct”
    char name[100];              // journal name
    Issn issn;                   // defined type for the ISSN of journal
    Publisher *publisher;        // defined type for the journal publisher
} Journal;                   // also known by its typedef name
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Header File* command to create **journal.h**:

```
/**
 * Create and initialize a journal for a publisher.
 * @param name the journal name
 * @param issn the journal issn
 * @param publisher the journal publisher
 * @return the journal
 */
Journal *newJournal(const char *name, Issn issn, Publisher *publisher);

/**
 * Delete the journal.
 * @param journal the journal to delete
 */
void deleteJournal(Journal *journal);
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Header File* command to create **journal.h**:

```
/**
  * Print a Journal only without publisher info.
  * @param journal the journal to print
  */
void printJournalOnly(const Journal *journal);

/**
  * Print a Journal and its publisher
  * @param journal the journal to print
  */
void printJournal(const Journal *journal);

#endif /* JOURNAL_H_ */
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New* -> *Source File* command to create **journal.c**:

```
#include ”publisher.h”
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/** Create and initialize a journal for a publisher.
 * @param name the journal name
 * @param issn the journal issn
 * @param publisher the journal publisher
 * @return the journal
 */
Journal *newJournal(const char *name, Issn issn, Publisher *publisher) {
    Journal *journal = malloc(sizeof(Journal));
    strcpy(journal->name, name);
    journal->issn = issn;
    journal->publisher = publisher;

    addJournalToPublisher(publisher, journal);
    return journal;
}
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Source File* command to create **journal.c**:

```
/** Delete the journal.
 * @param journal the journal to delete
 */
void deleteJournal(Journal *journal) {
    free(journal);
}

/** Print a Journal only without publisher info.
  * @param journal the journal to print
  */
void printJournalInfo(const Journal *journal) {
    char issnChars[10];
    char* issnString = issnToString(journal->issn, issnChars);
    printf("journal name: '%s'\nISSN: %s\n", journal->name, issnString);
}
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New* -> *Source File* command to create **journal.c**:

```
/**
  * Print a Journal and its publisher
  * @param journal the journal to print
  */
void printJournal(const Journal *journal) {
     printJournalInfo(journal);
     printPublisherInfo(journal->publisher);
}
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New* -> *Source File* command to create **publisher.c**:

```
#include ”publisher.h”
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/**
 * Create and initialize a publisher.
 * @param name the publisher name
 * @return the publisher
 */
Publisher *newPublisher(const char *name) {
    Publisher *publisher = malloc(sizeof(Publisher));
    strcpy(publisher->name, name);
    publisher->nJournals = 0;// no journals
    publisher->maxJournals = 2;    // initially two slots available
    publisher->journals = malloc(publisher->maxJournals * sizeof(Journal*));
    return publisher;
}
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New* -> *Source File* command to create **publisher.c**:

```
/**
 * Delete the publisher.
 * @param publisher the publisher to delete
 */
void deletePublisher(Publisher *publisher) {
     free(publisher->journals);       // first free dynamic array
     free(publisher);
}
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New* -> *Source File* command to create **publisher.c**:

```
/**
 * Add a journal to a publisher.
 * @param publisher the publisher
 * @param journal the journal to add
 */
void addJournalToPublisher(Publisher *publisher, Journal *journal) {
    if (publisher->nJournals >= publisher->maxJournals) {  // out of space
        publisher->maxJournals *= 2;  // double available size
        publisher->journals =                // grow storage to new available size
            realloc(publisher->journals, publisher->maxJournals * sizeof(Journal*));
    }
    publisher->journals[publisher->nJournals++] = journal;
}
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Source File* command to create **publisher.c**:

```
/**
 * Print a Publisher only without its journals.
 * @param publisher the publisher to print
 */
void printPublisherInfo(const Publisher *publisher) {
    printf("publisher name: ’%s’\n", publisher->name);
}

/**
 * Print a Publisher and its journals.
 * @param publisher the publisher to print
 */
void printPublisher(const Publisher *publisher) {
    printPublisherInfo(publisher);
    for (int jnl = 0; jnl < publisher->nJournals; jnl++) {
        printJournalOnly(publisher->journals[jnl]);
    }
}
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Source File* command to create **main.c**:

```
#include ”publisher.h”
#include “journal.h”

/** Test dynamic allocation of publisher and journal */
int main(void) {
    Publisher *ngs = newPublisher("National Geographic Society");

    Journal *nationalGeographic =
            newJournal("National Geographic", 0x00279358, ngs);
    Journal *nationalGeographicExplorer =
            newJournal("National Geographic Explorer", 0x15413357, ngs);
    Journal *nationalGeographicKids =
            newJournal("National Geographic Kids", 0x15423042, ngs);
```

Separate Compilation Units

Sharing Code and Declarations

- Use Eclipse *New -> Source File* command to create **main.c**:

```
        printJournal(nationalGeographic);
        printJournal(nationalGeographicExplorer);
        printJournal(nationalGeographicKids);
        printPublisher(ngs);

        deleteJournal(nationalGeographic);
        deleteJournal(nationalGeographicExplorer);
        deleteJournal(nationalGeographicKids);
        deletePublisher(ngs);
    }
```

Separate Compilation Units

Compile Module Local Variables, Constants, Functions

- By default, all C global variables and functions are visible globally throughout the the program.
- It is sometimes useful to declare variables and functions that are local to just a single C file where they are used. The C language uses the special keyword *static* to accomplish this:

  ```
  static Publisher **publishers = NULL;   // pointer to publisher pointer array
  ```

- Both variables and function can be declared static to ensure they are not visible outside the file where they are defined.