Lecture Notes for Lecture 10 of CS 5001 (Foundations of CS) for the Fall, 2018 session at the Northeastern University Silicon Valley Campus.

Defining Types: Structs

Philip Gust,
Clinical Instructor
Department of Computer Science

Lecture 9 Review

- An enumeration is a closed set of symbolic values that represents a fixed set of choices.
- Symbolic values make the program easier to design, and ensure the code is more readable and easier to maintain and extend.
- Enum types have their own name space and require using ‘enum’ qualifier for the type wherever instances are defined.
- Using typedef with enum eliminates the need for an ‘enum’ qualifier, by creating an alias in the regular type name space.
- Enum symbolic values have also have default numeric values that can also be specified expliclity.
- Enum symbolic values must be unique within a program, but symbols in different enums can have the same numeric value.
- Enums can be used as loop, array, and switch indexes.

Struct

- The final way to create a new type is as an aggregation of heterogeneous types.
- This aggregation allows types to model complex real-world data that includes related information of different types about an entity.
- This lecture will present the facility that the C language for defining aggregations of heterogeneous types, known as a *struct*.

Struct

- A struct is an aggregation of heterogeneous types that are specified by declaring fields of the struct.
- Example:

```
/** Typedef for ISSN: nnnn-nnnN */
typedef uint32_t Issn;

/** Struct that defines a Journal */
struct  Journal {
    char name[100];         // journal name
    Issn issn;              // defined type for the ISSN of journal
    char publisher[100];    // name of publisher
};
```

- Like enum, struct type names are defined in their own name space, and declaring struct variables requires using ‘struct’ as part of the type:

```
struct Journal myJournal;
```

Struct

- At first having struct type identifiers in their own name space, separate from identifiers for function, variable, const, and typedef, seemed like a good idea to C language developers.
- In practice having a separate name space for struct types had limited value.
- C++, Java, and other programming languages put the equivalent of struct types identifiers in the same namespace other identifiers.
- Most C programmers achieve the same effect by using typedef to create an alias that eliminates the need to use the struct type.

Struct

- Creating a typedef for a struct allows the typedef type to be used instead of the struct type.

```
typedef struct Journal Journal;      // struct Journal is struct type
struct Journal journal1;             // using Journal struct type
Journal journal2;                    // using Journal typedef
```

- The struct and typedef can be combined into one declaration.

```
// combine struct and typedef definitions into one declaration
typedef struct Journal { … } Journal;
```

- The struct type name can be left off if is is not needed. This is known as an "*anonymous struct.*"

```
// omit struct type name and only refer to typedef type name
typedef struct { … } Journal;
```

Struct

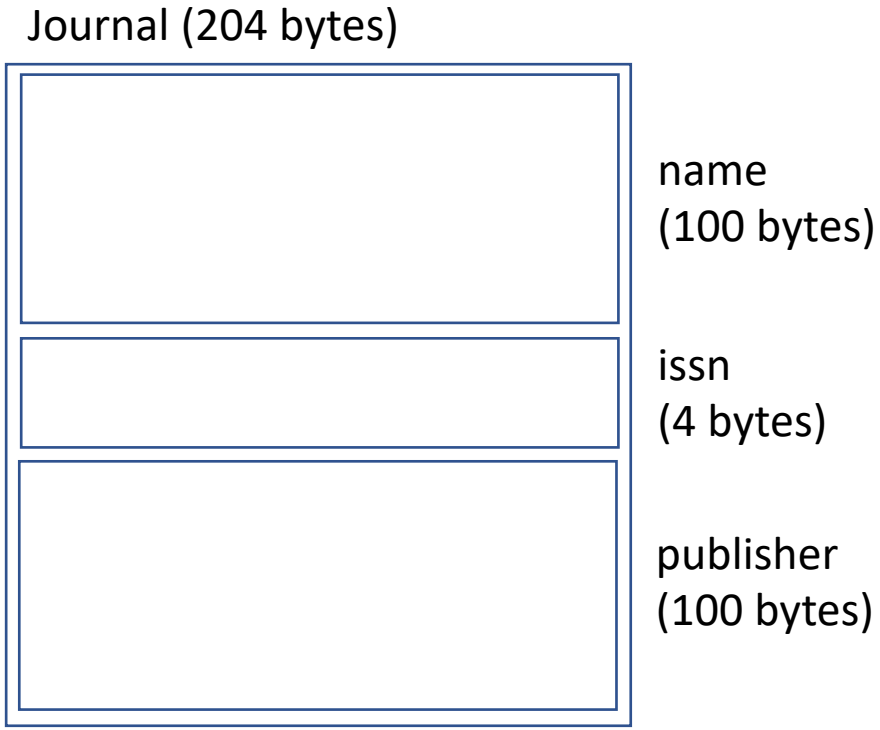
- Example: struct declaration omits struct type and uses only typedef type.

```
/** Typedef for ISSN: nnnn-nnnN */
typedef uint32_t Issn;

/** Struct that defines a Journal */
typedef struct {        // “anonymous struct”
    char name[100];         // journal name
    Issn issn;              //  defined type for the ISSN of journal
    char publisher[100];    // name of publisher
} Journal;                  // only known by its typedef name
```

Struct

- Journal struct memory structure



Struct

- Defining and initializing Journal struct with initializer list

```
/**
 * Get string representation of ISSN.
 * @param issn the ISSN value
 * @param issnChars array to receive the ISSN characters
 * @return pointer to ISSN chars
 */
char *issnToString(Issn issn, 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 for check digit
    }
    return issnChars;
}
```

Struct

- Defining and initializing Journal struct with initializer list

```
/** Create and print a Journal */
int main(void) {
    Journal myJournal= {                // positional initializer list
        “National Geographic”,                  // name
        0x00279358,                             // issn (as uint32_t)
        ”National Geographic Society”           // publisher
    };
    char issnChars[10];
    char* issnString = issnToString(myJournal.issn, issnChars);
    printf(”Name: ’%s’\nISSN: %s\nPublisher: ’%s’\n”,
            myJournal.name, issnString, myJournal.publisher);
}
```

Struct

- Defining and initializing Journal struct with C99 initializer list.

```
/** Create and print a Journal */
int main(void) {
    Journal myJournal= {                // designated initializer list (C99)
        .issn = 0x00279358,                                 // issn (as uint32_t)
        .name = “National Geographic”,                      // name
        .publisher = ”National Geographic Society”   // publisher
    };
    char issnChars[10];
    char* issnString = issnToString(myJournal.issn, issnChars);
    printf(”Name: ’%s’\nISSN: %s\nPublisher: ’%s’\n”,
            myJournal.name, issnString, myJournal.publisher);
}
```

Struct

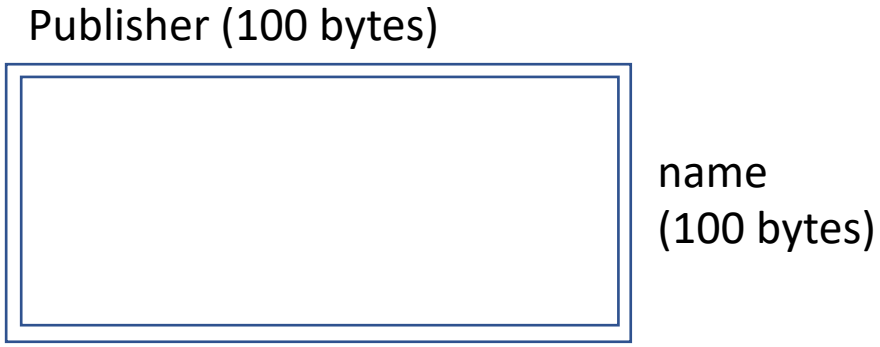
- A struct can have fields of any type including another struct.
- Suppose that instead of just a publisher name string, a publisher is represented by another struct:

```
/** Struct that defines a Publisher */
typedef struct  Publisher{
    char name[100];         // publisher name
} Publisher;
```

- This might be done to facilitate adding other publisher fields in the future.

Struct

Publisher struct memory structure



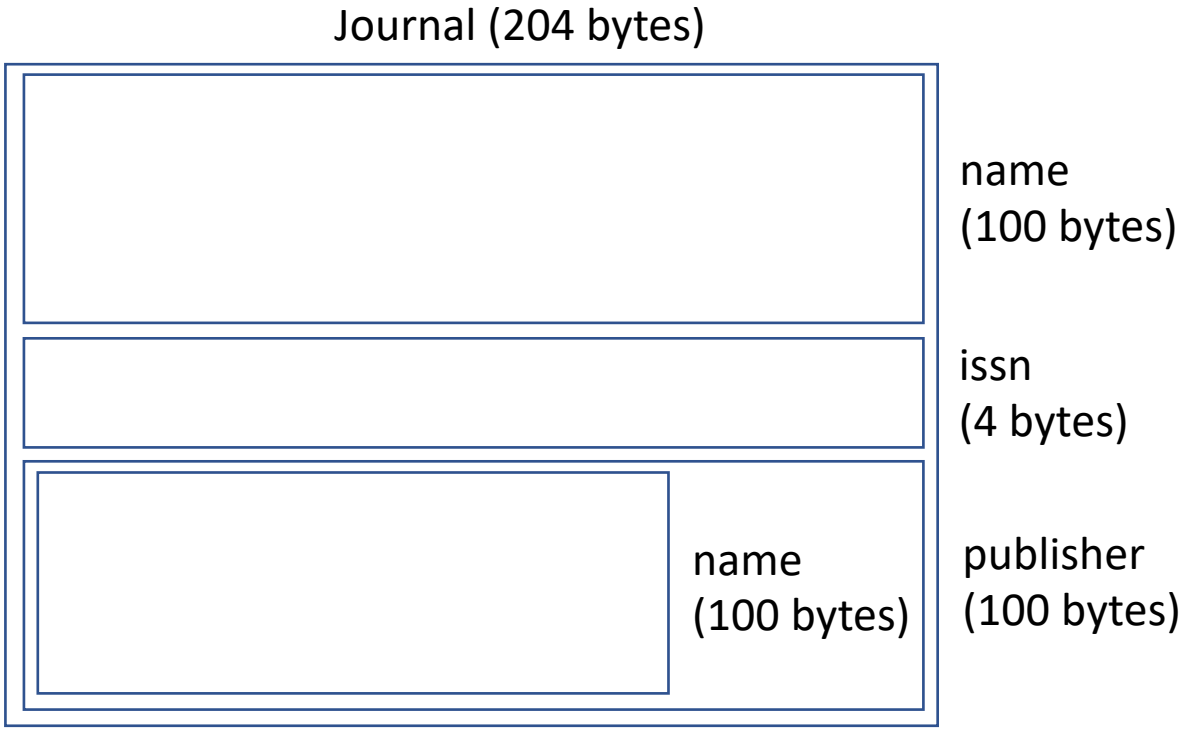
Struct

- Here is how the Journal struct changes to embed the Publisher struct:

```
/** Struct that defines a Journal */
typedef struct  Journal {
    char name[100];          // journal name
    Issn issn;               // ISSN of journal
    Publisher publisher      // the publisher
} Journal;
```

Struct

Journal with nested Publisher struct memory structure



Struct

Here is how initialization of the Journal struct changes

```
/** Create and print a Journal */
int main(void) {
    Journal myJournal = {              // positional initializer list
        “National Geographic”,                  // name
        0x00279358,                             // issn (as uint32_t)
        {”National Geographic Society”}         // publisher
    };
    char issnChars[10]
    char* issnString = issnToString(myJournal.issn, issnChars);
    printf(”Name: ’%s’\nISSN: %s\nPublisher: ’%s’\n”,
            myJournal.name, issnString, myJournal.publisher.name);
}
```

Struct

Defining and initializing a Journal struct in C99

```
/** Create and print a Journal */
int main(void) {
    Journal myJournal = {                // designated initializer list (C99)
        .name = “National Geographic”,
        .issn = 0x00279358,
        .publisher = {  .name = ”National Geographic Society” }
    };
    char issnChars[10]
    char* issnString = issnToString(myJournal.issn, issnChars);
    printf(”Name: ’%s’\nISSN: %s\nPublisher: ’%s’\n”,
            myJournal.name, issnString, myJournal.publisher.name);
}
```

Struct

Copying Structs

- A struct can be copied by assigning to another struct. C copies each field of the struct and any nested structs.

```
Journal myJournal = {              // positional initializer list
      “National Geographic”,                  // name
      0x00279358,                             // issn (as uint32_t)
      {”National Geographic Society”}         // publisher
};
char issnChars[10]
char* issnString = issnToString(myJournal.issn, issnChars);
printf(”Name: ’%s’\nISSN: %s\nPublisher: ’%s’\n”,
          myJournal.name, issnString, myJournal.publisher.name);

Journal yourJournal = myJournal;
char* issnString = issnToString(yourJournal.issn, issnChars);
printf(”Name: ’%s’\nISSN: %s\nPublisher: ’%s’\n”,
          yourJournal.name, issnString, yourJournal.publisher.name);
```

Struct

Struct as Parameter

- C passes and returns structs by *value*, just like basic types. The struct is copied to a local struct parameter variable. Changes made to local copy do not impact called value.
- In many cases, structs should be passed by reference using a struct pointer parameter, and passing the address of the struct to the function.

Struct

Printing a Journal From a Function

```
/**
 * Print a Journal
 * @param journal the journal to print
 */
void printJournal(const Journal *journal) {
    char issnChars[10];
    char* issnString = issnToString((*journal).issn, issnChars);
    printf("Name: '%s'\nISSN: %s\nPublisher: '%s'\n",
            (*journal).name, issnString, (*journal).publisher.name);
}
```

- Note how journal pointer is dereferenced to access its fields

Struct

Printing a Journal From a Function

```
/** Create and print a Journal */
int main(void) {
    Journal journal = {                    // positional initializer list
            .name = “National Geographic”,
            .issn = 0x00279358,
            .publisher = { .name = ”National Geographic Society” }
    };
    printJournal(&journal);      // pointer to pass journal by reference
    return EXIT_SUCCESS;
}
```

Struct

C provides a ‘->’ operator to access fields of struct pointer

```
/**
 * Print a Journal
 * @param journal the journal to print
 */
void printJournal(const Journal *journal) {
    char issnChars[10]
    char* issnString = issnToString(journal->issn, issnChars);
    printf(”Name: ’%s’\nISSN: %s\nPublisher: ’%s’\n”,
            journal->name, issnString, journal->publisher.name);
}
```

- Note how ‘->’ operator dereferences journal pointer and access fields.

Struct

Printing a Journal From a Function

- Now that we have a separate Publisher struct, we should also create a printPublisher function.
- Adding fields to the Publisher struct in the future should only impact Publisher methods, and not Journal methods.

```
/**
  * Print a Publisher
  * @param publisher the publisher to print
  */
void printPublisher(const Publisher *publisher {
    printf(”Publisher: ’%s’\n”,  publisher->name);
}
```

- Note how publisher pointer is dereferenced to access its name field

Struct

Here is an updated version of printJournal()

```
/**
 * Print a Journal
 * @param journal the journal to print
 */
void printJournal(const Journal *journal) {
    char issnChars[10];
    char* issnString = issnToString(journal->issn, issnChars);
    printf("Name: '%s'\n", journal->name);
    printf("ISSN: %s\n", issnString);
    printPublisher(&(journal->publisher));  // pointer to journal publisher field
}
```

- Note journal publisher field is passed by reference as pointer

Struct

Struct as Return Value

- C can return a struct by value from a function.

```
/**
  * Get a Journal from a source.
  * @return a journal
  */
Journal getJournal(void) {
    Journal journal = {                    // positional initializer list
        .name = “National Geographic”,
        .issn = 0x00279358,
        .publisher = { .name = ”National Geographic Society” }
    };
    return journal;
}
```

Struct

Printing a Journal From a Function

```
/** Create and print a Journal */
int main(void) {
    Journal journal = getJournal();   // copy journal returned by value
    printJournal(&journal);      // pointer to pass journal by reference
    return EXIT_SUCCESS;
}
```

Structs

Struct Pointing to Struct

- If there are a number of journals from a publisher, it it wasteful to copy the publisher information for each journal.
- If we update information about a journal such as adding or modifying a field, every journal for that publisher is impacted.
- Instead, we can have a single instance of each Publisher, and modify the Journal instances for each Publisher to point to it, rather than embed their own copies of the Publisher.

Struct

Here is how Journal struct changes to point to a Publisher struct:

```
/** Struct that defines a Journal */
typedef struct  Journal {
    char name[100];          // journal name
    Issn issn;               // ISSN of journal
    Publisher *publisher     // pointer to publisher
} Journal;
```

Struct as Parameters

Here is how the publisher and journal are initialized:

```
/** Create and print a Journal */
int main(void) {
    Publisher publisher = {
        .publisher = { .name = ”National Geographic Society” }
    }
    Journal journal = {                    // positional initializer list
        .name = “National Geographic”,
        .issn = 00279358,
        .publisher = &publisher
    };
    printJournal(&journal);      // pointer to pass journal by reference
    return EXIT_SUCCESS;
}
```

Struct as Parameters

Here is an updated version of printJournal() calling printPublisher() with the journal publisher pointer field.

```
/**
 * Print a Journal
 * @param journal the journal to print
 */
void printJournal(const Journal *journal) {
    char issnChars[10];
    char* issnString = issnToString(journal->issn, issnChars);
    printf("Name: '%s'\n", journal->name);
    printf("ISSN: %s\n", issnString);
    printPublisher(journal->publisher);  // journal publisher pointer field
}
```