Software Security

In This Course, 'C'ing is Believing

Mohamed Sabt

Univ Rennes, CNRS, IRISA

2022 / 2023

Part I

Introduction

Basic Concepts

- This lecture provides definitions for the specific terminology and the concepts used when describing the C programming language.

- A C program is a sequence of text files (typically header and source files) that contain declarations.
  - They undergo translation to become an executable program, which is executed when the OS calls its main function

- Certain words in a C program have special meaning,
  - they are keywords.
  - Others can be used as identifiers.

Phases of Translation

- The C source file is processed by the compiler *as if* the following phases take place, in this exact order.

- Phase 4:
  - Preprocessor is executed

- Phase 7:
  - Compilation takes place: the tokens are syntactically and semantically analyzed and translated as a translation unit.

- Phase 8:
  - Linking takes place: Translation units and library components needed to satisfy external references

Main Function

- Every C program coded to run in a hosted execution environment contains the definition (not the prototype) of a function called main.

- The main function is called at program startup,
  - after all objects with static storage duration are initialized.
  - It is the designated entry point to a program that is executed in *hosted* environment (that is, with an operating system).

- Inside the main function, if the control reaches the terminating }, the value returned to the environment is the same as if executing return 0;

Memory Model

- The data storage (memory) available to a C program is one or more contiguous sequences of *bytes*.
  - Each byte in memory has a unique *address*.

- A *byte* is the smallest addressable unit of memory. It is defined as a contiguous sequence of bits
  - The number of bits in a byte is accessible as CHAR_BIT.

- The types char, unsigned char, and signed char use one byte for both storage and value representation.

Object Representation

- Except for bit fields, objects are composed of contiguous sequences of one or more bytes, each consisting of CHAR_BIT bits
  - can be copied with memcpy into an object of type unsigned char[n], where n is the size of the object.
  - The contents of the resulting array are known as *object representation*.

- When objects of integer types (short, int, long, long long) occupy multiple bytes, the use of those bytes is implementation-defined, the two dominant implementations are
  - *big-endian* (POWER, Sparc, Itanium) stores the most significant byte at the lowest address.
  - *little-endian* (x86, x86_64): stores the least significant byte at the lowest address.

Part II

Preprocessing

Preprocessor

- The preprocessor is executed at translation phase 4, before the compilation. The result of preprocessing is a single file which is then passed to the actual compiler.

- The preprocessing directives control the behavior of the preprocessor.
- Each directive occupies one line and has the following format
  - *# character*
  - preprocessing instruction
  - arguments (depends on the instruction)
  - line break

Capabilities

- **conditionally** compile of parts of source file (controlled by directive #if, #ifdef, #ifndef, #else, #elif and #endif.
- **replace** text macros while possibly concatenating or quoting identifiers (controlled by directives #define and #undef, and operators # and ##)
- **include** other files (controlled by directive #include)
- cause an **error** (controlled by directive #error)
- **file name and line information** available to the preprocessor (controlled by directives #line)

Conditional Inclusion

- **#if, #elif**
  - The expression is a constant expression, using only constants and identifiers, defined using #define directive.
  - Any identifier, which is not literal, non defined using #define, evaluates to *0*.

- **#ifdef, #ifndef**
  - Checks if the identifier was defined using #define directive.
  - #ifndef identifier is essentially equivalent to #if !defined identifier

Replacing text macros

- **#define identifier replacement-list**
  - The #define directives define the identifier as a macro, that is they instruct the compiler to replace all successive occurrences of identifier with replacement-list

- **Two types of macros:**
  - **Object-like macros**
  - **Function-like macros**

- **#undef directive**
  - #undef directive undefines the identifier, that is it cancels the previous definition of the identifier by #define directive.

Source File Inclusion

- **#include <filename>**
  - Searches for the file in implementation-defined manner.
  - Typical implementations search only standard include directories.

- **#include “filename”**
  - Searches for the file in implementation-defined manner.
  - Typical implementations first search the directory where the current file resides and, only if the file is not found, search the standard include directories as with #include <filename>

Implementation Defined Directive

- **#pragma pragma_params**
  - Behaves in an implementation-defined manner
  - Unless pragma_params is one of the standard pragmas

- **Standard Pragmas**
  - **#pragma STDC FENC_ACCESS on/off**
  - **#pragma STDC FP_CONTRACT on/off**
  - **#pragma STDC CX_LIMITED_RANGE on/off**

- **Non-Standard Pragmas**
  - **#pragma once:** similar to include guards
  - **#pragma pack(arg):** sets the current alignment to value *arg* for structures

Part III

Statements / Expressions

Statements

- Statements are fragments of the C program that are executed in sequence.

- There are five types of statements:
  - Compound statements.
  - Expression statements.
  - Selection statements.
  - Iteration statements.
  - Jump statements.

Labels

- Any statement can be *labeled*, by providing a name followed by a colon before the statement itself.

- There are three types of labels:
  - Target for goto.
  - Case label in a switch statement.
  - Default label in a switch statement.

- Labels must be unique within the enclosing function.
- Label declaration has no effect on its own
  - does not alter the flow of control,
  - or modify the behavior of the statement that follows in any way.

Statements Types 1/2

- Compound statements:
  - A compound statement, or *block*, is a brace-enclosed sequence of statements and declarations.
  - Each compound statement introduces its own block scope.

- Expression Statements:
  - An expression followed by a semicolon is a statement.
  - An expression statement without an expression is called a *null statement*.

- Selection Statements (if, switch)
  - The selection statements choose between one of several statements depending on the value of an expression.

Statements Types 2/2

- Iteration statements (while, do-while, for):
  - The iteration statements repeatedly execute a statement.

- Jump Statements (break, continue, return, goto):
  - The jump statements unconditionally transfer flow control.

Expressions

- An expression is a sequence of *operators* and their *operands*, that specifies a computation.

- Primary expressions
  - Constants and literals,
  - Declared identifiers.

- Constants and literals: Constant values of certain types may be embedded in the source code of a C program using specialized expressions known as literals:
  - Integer constants,
  - Character constants,
  - Floating constants,
  - String literals.

Value Categories

- Each expression in C is characterized by two independent properties: a type and a value category.

- Every expression belongs to one of three value categories:
  - lvalue,
  - non-lvalue object (rvalue),
  - function designator.

Lvalue Expressions 1/2

- The name of this value category ("left value") is historic:
  - reflects the use of lvalue expressions as the left-hand operand of the assignment operator.
- Lvalue expressions can be used in the following *lvalue contexts*:
  - as the operand of the address-of operator
  - as the operand of the pre/post increment and decrement operators.
  - as the left-hand operand of the member access (dot) operator.
  - as the left-hand operand of the assignment operators.

- Lvalues:
  - Identifiers,
  - String/Compound literals
  - The result of member access
  - The result of subscription operator
  - The result of indirection operator

Lvalue Expressions 2/2

- Lvalue expressions undergo lvalue conversion, which models the memory load of the value of the object from its location.

- Modifiable lvalue expressions
  - Only modifiable lvalue expressions may be used as arguments to increment/decrement, and as left-hand arguments of assignment and compound assignment operators.

Non-lvalue Expressions

- non-lvalue object expressions are the expressions of object types that do not designate objects, but rather values that have no object identity or storage location.
  - The address of a non-lvalue object expression cannot be taken.

- Non-lvalue object expressions include
  - Integer, character and floating constants,
  - All operators not specified to return lvalues

Integer Constant

- An integer constant is an non-lvalue expression of the form
  - Decimal: 10
  - Octal: 012
  - Hex: 0xA
  - Binary: 0b1010

- Suffixes:
  - Unsigned: character U
  - Long: character L

Compound Literals

- Constructs an unnamed object of specified type in-place, used when a variable of array, struct, or union type would be needed only once.
  - Syntax: (type) {initializer-list}
  - The value category of a compound literal is lvalue (its address can be taken).

- The unnamed object to which the compound literal evaluates has
  - static storage duration if the compound literal occurs at file scope
  - automatic storage duration if the compound literal occurs at block scope

- Although the syntax of a compound literal is similar to a cast, the important distinction is that a cast is a non-lvalue expression while a compound literal is an lvalue.

Exercise

```c
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#define MAX 10
struct point {
    uint8_t x, y;
};

int main(void) {
    uint8_t i;
    struct point * tab[MAX];
    for (i = 0 ; i < MAX ; i++) {
        tab[i] = &(struct point){i,2*i};
    }
    for (i = 0 ; i < MAX ; i++) {
            printf("%d\n", tab[i]->x);
        }
}
```

Part IV

Objects

Identifiers

- Identifiers can denote the following types of entities:
  - Objects
  - Functions
  - Struct
  - union
  - enumeration tags,
  - their members,
  - typedef names,
  - labels,
  - macros.

- Each object, function, and expression in C is associated with a type.

Declaration

- Each identifier (other than macro) is only valid within a part of the program called its scope,
  - and belongs to one of four kinds of name spaces.

- Some identifiers have linkage which makes them refer to the same entities when they appear in different scopes or translation units.

Scope

- Each identifier that appears in a C program is *visible* (that is, may be used) only in some portion of the source code called its *scope*.

- Within a scope, an identifier may designate more than one entity only if the entities are in different name spaces.

- C has four kinds of scopes:
  - Block scope
  - File scope
  - Function scope
  - Function prototype scope

Block Scope

- The scope of any identifier declared inside a compound statement.

- Block-scope variables have no linkage and automatic storage duration by default.
- Note that storage duration for non-VLA local variables begins when the block is entered, but until the declaration is seen, the variable is not in scope and cannot be accessed.

File Scope

- The scope of any identifier declared outside of any block or parameter list begins at the point of declaration and ends at the end of the translation unit.

- File-scope identifiers have external linkage and static storage duration by default.

Function Prototype Scope

- The scope of a name introduced in the parameter list of a function declaration that is not a definition ends at the end of the function declaration.

Point of Declaration

- The scope of structure, union, and enumeration tags begins immediately after the appearance of the tag in a type specifier that declares the tag.
  - Question: what is the consequence of that?

- The scope of any other identifier begins just after the end of its declarator and before the initializer, if any.

- Unlike C++, C has no struct scope: names declared within a struct/union/enum declaration are in the same scope as the struct declaration (except that data members are in their own member name space).

Namespaces

- When an identifier is encountered in a C program, a lookup is performed to locate the declaration that introduced that identifier and that is currently in scope.

- C allows more than one declaration for the same identifier to be in scope simultaneously if these identifiers belong to different categories, called *name spaces*:
  - Label name space: all identifiers declared as labels.
  - Tag names: all identifiers declared as names of structs, unions and enumerated types.
  - Member names: all identifiers declared as members of any one struct or union.
  - All other identifiers, called *ordinary identifiers* to distinguish from (1-5) (function names, object names, typedef names, enumeration constants).

- At the point of lookup, the name space of an identifier is determined by the manner in which it is used.
- It is common practice to inject struct/union/enum names into the name space of the ordinary identifiers using a typedef declaration

Storage Duration

- ***automatic*** storage duration.
  - The storage is allocated when the block in which the object was declared is entered and deallocated when it is exited by any means.
  - One exception is the VLAs; their storage is allocated when the declaration is executed, not on block entry, and deallocated when the declaration goes out of scope, not when the block is exited.
  - All function parameters and non-static block-scope objects have this storage duration.

- ***static*** storage duration.
  - The storage duration is the entire execution of the program, and the value stored in the object is initialized only once, prior to main function.
  - All objects declared static and all objects with either internal or external linkage have this storage duration.

- ***allocated*** storage duration.
  - The storage is allocated and deallocated on request, using dynamic memory allocation functions.

- ***Thread*** storage duration.
  - The storage duration is the entire execution of the thread in which it was created, and the value stored in the object is initialized when the thread is started.

Linkage

- Linkage refers to the ability of an identifier (variable or function) to be referred to in other scopes. If a variable or function with the same identifier is declared in several scopes, but cannot be referred to from all of them, then several instances of the variable are generated.

- ***no linkage***. The identifier can be referred to only from the scope it is in.
  - All function parameters and all non-extern block-scope variables (including the ones declared static) have this linkage.

- ***internal linkage***. The identifier can be referred to from all scopes in the current translation unit.
  - All static file-scope identifiers (both functions and variables) have this linkage.

- ***external linkage***. The identifier can be referred to from any other translation units in the entire program.
  - All non-static functions, all extern variables (unless earlier declared static) and all file-scope non-static variables have this linkage.

Question

- In headers, what kind of linkage and scope all variables/functions should be?

Lifetime

- Every object in C exists, has a constant address, retains its last-stored value over a portion of program known as this object's *lifetime*.

- For the objects that are declared with automatic, static, and thread storage duration, lifetime equals to their storage duration.

- For the objects with allocated storage duration, the lifetime begins when the allocation function returns (e.g., malloc)

Objects

- C programs create, destroy, access, and manipulate objects.
- Every object has
  - size (can be determined with sizeof)
  - alignment requirement (can be determined by _Alignof)
  - storage duration (automatic, static, allocated, thread-local)
  - lifetime (equal to storage duration or temporary)
  - effective type
  - value (which may be indeterminate)
  - optionally, an identifier that denotes this object
- Objects are created by declarations, allocation functions, string literals, compound literals

Effective Type

- Every object has an *effective type*, which determines which lvalue accesses are valid and which violate the strict aliasing rules.
- If the object was created by a declaration, the declared type of that object is the object's *effective type*.

- If the object was created by an allocation function, the first write to that object through an lvalue that has a type other than character type, at which time the type of that lvalue becomes this object's *effective type*

Alignment

- Every object type has a property called *alignment requirement*, which represents the number of bytes between successive addresses at which objects of this type can be allocated.
- The valid alignment values are non-negative integral powers of two.
- The alignment requirement of a type can be queried with _Alignof

- In order to satisfy alignment requirements of all members of a struct, padding may be inserted after some of its members.

Question 1

- What will be the outputs?

```c
#include <stdio.h>
#include <stdalign.h>


struct S {
    char a; // size: 1
    char b; // size: 1
};


struct X {
    int n;  // size: 4
    char c; // size: 1,
};

int main(void)
{
    printf("sizeof(struct S) = %zu\n", sizeof(struct S));
    printf("alignof(struct S) = %zu\n", alignof(struct S));
    printf("sizeof(struct X) = %zu\n", sizeof(struct X));
    printf("alignof(struct X) = %zu\n", alignof(struct X));
}
```

Question 2

- Find the problem?

```
1   int i = 7;
2   char* pc = (char*)(&i);
3
4   if (pc[0] == '\x7') {
5       puts("This system is little-endian");
6   } else {
7       puts("This system is big-endian");
8   }
9
10  float* pf = (float*)(&i);
11  float d = *pf;
```

Part V

Declaration / Initialization

Declarations

- A *declaration* is a C language construct that introduces one or more identifiers into the program and specifies their meaning and properties.
- Declarations may appear in any scope.
  - Each declaration ends with a semicolon (just like a statement) and consists of two (until C23) three (since C23) distinct parts.

- specifiers-and-qualifiers
  - type specifiers
  - zero or one storage-class specifiers: auto, register, static, extern, _Thread_local
  - Zero or more type qualifiers: const, volatile, restrict, _Atomic
  - (only when declaring functions), zero or more function specifiers: inline, and _Noreturn
  - Zero or more alignment specifiers: _Alignas

- declarators-and-initializers

Declarators

- Identifier
- * qualifiers-declarator
- Noptr declarator[static qualifiers expression]
- Noptr declarator[qualifiers *]
- Noptr declarator(parameters)

- Qualifiers: const, volatile and restrict

Storage Class Specifiers

- auto -- automatic duration and no linkage
  - The auto specifier is only allowed for objects declared at block scope (except function parameter lists). It indicates automatic storage duration and no linkage, which are the defaults for these kinds of declarations.
- register -- automatic duration and no linkage; address of this variable cannot be taken
  - The register specifier is only allowed for objects declared at block scope, including function parameter lists.
- static -- static duration and internal linkage (unless at block scope)
  - The static can be used with functions at file scope and with variables at both file and block scope, but not in function parameter lists.
- Extern -- static duration and external linkage (unless already declared internal)
  - If extern appears on a redeclaration of an identifier that was already declared with internal linkage, the linkage remains internal.

_Alignas

- Appears in the declaration syntax as one of the type specifiers to modify the alignment requirement of the object being declared.

- _Alignas specifier only needs to appear on the definition of an object, but if any declaration uses _Alignas, it must specify the same alignment as on the definition.

Redeclaration

- A declaration cannot introduce an identifier if another declaration for the same identifier in the same scope appears earlier, except that

- Declarations of objects with linkage (external or internal) can be repeated
- Non-VLA typedef can be repeated as long as it names the same type.
- struct and union declarations can be repeated:

Pointer Declaration

- Pointer is a type of an object that refers to a function or an object of another type, possibly adding qualifiers.
  - Pointer may also refer to nothing, which is indicated by the special null pointer value.

- A pointer whose value is null does not point to an object or a function (dereferencing a null pointer is undefined behavior).

- The qualifiers that appear between * and the identifier qualify the type of the pointer that is being declared.
- Any pointer to object can be cast to pointer to object of a different type.

Array Declaration

- Array is a type consisting of a contiguously allocated nonempty sequence of objects with a particular *element type*.

- In each function call to a function where a parameter of array type uses the keyword static, the value of the actual parameter must be a valid pointer to the first element of an array with at least as many elements as specified by expression.
- If qualifiers are present, they qualify the pointer type to which the array parameter type is transformed.

- Arrays of constant known size
  - Arrays of constant known size can use array initializers to provide their initial values
- Variable-Length Arrays (VLA)
  - If the size is *, the declaration is for a VLA of unspecified size. Such declaration may only appear in a function prototype scope
  - VLA must have automatic or allocated storage duration (what is the consequence?)
- Arrays of unknown size
  - If expression in an array declarator is omitted, it declares an array of unknown size.
  - Within a struct definition, an array of unknown size may appear as the last member (as long as there is at least one other named member)

Initialization

- A declaration of an object may provide its initial value through the process known as *initialization*.

- For each declarator, the initializer, if not omitted, may be one of the following:
  - Expression
  - {initializer-list}
- Implicit Initialization:
  - Objects with automatic storage duration are initialized to indeterminate values.
  - with static and thread-local storage duration are zero-initialized

Array Initialization

- When initializing an object of array type, the initializer must be either a string literal or be a brace-enclosed list for array members.

- Arrays of known size and arrays of unknown size may be initialized (but not VLA).
  - When initializing an array of unknown size, the largest subscript for which an initializer is specified determines the size of the array being declared.

- All array elements that are not initialized explicitly are zero-initialized.

Question

- How can you initialize all the elements of an array to zero?
- How can you initialize the 100^{th} element to 5, and the remaining elements to zero?

Definitions

- A *definition* is a declaration that provides all information about the identifiers it declares.

- For functions, a declaration that includes the function body is a function definition.
- For objects, a declaration that allocates storage (automatic or static, but not extern) is a definition, while a declaration that does not allocate storage (external declaration) is not.
- For structs and unions, declarations that specify the list of members are definitions

Tentative Definitions

- A *tentative definition* is an external declaration without an initializer.

- *A tentative definition* is a declaration that may or may not act as a definition.
  - If an actual external definition is found earlier or later in the same translation unit, then the tentative definition just acts as a declaration.

- If there are no definitions in the same translation unit, then the tentative definition acts as an actual definition with the initializer = 0 (or, for array, structure, and union types, = {0}).
- One Definition Rule
  - If an identifier with linkage is used in any expression, there must be one and only one external definition for that identifier somewhere in the entire program (or the translation unit).

Exercise

```
3   int x;
4   int x;
5
6   int main(void) {
7       int y;
8       int y;
9       return 0;
10  }
11
```

Part VI

Functions

Functions

- A function is a C language construct that associates a compound statement (the function body) with an identifier (the function name).

- Functions may accept zero or more *parameters*, which are initialized from the *arguments* of a function call operator, and may return a value to its caller by means of the return statement.

- Each function that is actually called must be defined only once in a program, unless the function is inline.

- each function definition must appear at file scope, and functions have no access to the local variables from the caller.

Declaration

- The return type
  - Must be non-array type or the type void.
  - It cannot be cvr-qualified.

- If a function declaration appears outside of any function, the identifier it introduces has file scope and external linkage, unless static is used.

- Parameters:
  - The parameters in a declaration that is not part of a function definition do not need to be named.
  - Any parameter of array type is adjusted to the corresponding pointer type.
  - The special parameter list that consists entirely of the keyword void is used to declare functions that take no parameters.

Notes on Declaration

- the declarators f() and f(void) have different meaning.

- f(void) is declares a function that takes no parameters.
- f() declares a function that takes *unspecified* number of parameters.

_Noreturn Specifier

- The _Noreturn keyword appears in a function declaration.
- It specifies that the function does not return by executing the return statement or by reaching the end of the function body.
- This specifier is typically used through the convenience macro noreturn, which is provided in the header stdnoreturn.h.

Inline Specifier

- The intent of the inline specifier is to serve as a hint for the compiler to perform optimizations.
  - The compilers can (and usually do) ignore presence or absence of the inline specifier for the purpose of optimization.
- Any function with internal linkage may be declared static inline with no other restrictions.
- The inline definition that does not use extern is not externally visible and does not prevent other translation units from defining the same function.
- At most one translation unit may also provide a regular, non-inline non-static function, or a function declared extern inline.