MPCS 51040 – C Programming

Lecture 6 – Recursion, Linked List

Dries Kimpe

May 1, 2017

Operator Precedence and Associativity

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-Left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Associativity defines the order in which adjacent operators *with the same precedence level* are evaluated.

- a−b−c => (a−b)−c
- q && r || s => (q && r) || s

Does not say in which order the operator *operands* have to be evaluated!

precedence.c

Evaluation Order

```
// Unspecified function
// evaluation order;
// Associativity only
// defines addition order.
f1()+f2()+f3();

// bad; no sequence point
a[i]=b[i++];

// bad; no sequence point
// however, all side effects
// done *before* entering
// function
f(i++,i,j++);

// OK; , is seq point
i++,a=i;
```

Order of operations of the operands of *almost all* operators is not defined by the language. The compiler is free to evaluate operands in any order (and does not have to be consistent).

Only the sequential-evaluation (,), logical-AND (&&), logical-OR (||), conditional-expression (? :), and function-call operators constitute **sequence points** and therefore guarantee a particular order of evaluation for their operands.

(sequence point: all side effects of previous evaluations have occurred)

The function-call operator is the set of parentheses following the function identifier. The sequential-evaluation operator (,) is guaranteed to evaluate its operands from left to right. (Note that the comma operator in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.)

Boolean Short-circuit Evaluation

```c
// call to f never happens
0 && f();

// OK but avoid due to short circuit
// with side-effects;
// logical operator is seq point
q && r || s--;
```

Logical operators also guarantee evaluation of their operands from left to right. However, they evaluate the smallest number of operands needed to determine the result of the expression. This is called "short-circuit" evaluation. Thus, some operands of the expression may not be evaluated.

Expert topic; Better to avoid relying on specific sequencing whenever possible.

Recursion

```c
void repeat(unsigned int i)
{
    if (!i)
        return;
    // do something
    puts("X");
    repeat(i-1);
}
```

Recursion:

- Recursion happens when a function calls itself.
- Each invocation of the function receives its own copy of local (automatic) variables (variables go on the stack).
- Local variables exists until the function returns.
  - Watch out for memory (stack) usage!
- Recursion is a form of looping.
- Code on left: **tail** recursion

Recursion needs to end: *base case*.

Recursion

Typical implementation (compiler detail)

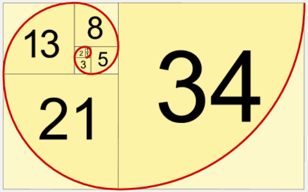
somelocalvar (cur)
i (cur)
(return address)
somelocalvar (prev)
i (prev)
(return address)
...

stack

- A stack structure is perfect to keep track of function call data
  - We only need access to the data (local variables, parameters, ...) of the *current* function.
  - When returning, we want to restore the previous situation
  - Multiple concurrent invocations of the function should be possible
- Most processors have built-in support for keeping track of strack structures (`push` and `pop` operation)

Recursion

Example



Calculate fibonacci numbers: $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0, F_1 = 1$

Solution

The base case and recursion step are explicit in the mathematical definition.

Calculate fibonacci numbers (`fibonacci.c`)

Recursion

Example

Recursion is very useful for divide-and-conquer type algorithms: split the problem in smaller problems and try to solve the smaller problem.

Example

Count the number of occurrences of a number in an array.

Solution

- *Base case (conquer): array of size 1*
- *Divide: split in two arrays, add counts.*

```
unsigned int count (const int * array, unsigned int size, int val);
```

Implement count().

Unit Testing

In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure.

Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process. It forms the basis for component testing. ***Ideally, each test case is independent from the others****. Substitutes such as method stubs, mock objects, fakes, and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended.*

Unit Test Frameworks

Unit Test Frameworks

All unit tests share some common elements:

- Need to report the results
- Need to make it simple to add tests
  - Including setup and teardown of test environments

Because of this, every programming language typically has one or more unit test frameworks/libraries available.

In this course, we will be using `cunit`.

`http://cunit.sourceforge.net/`;
See 'example code' for a quick start.

CUnit

Available as a library (pre-installed on linux.cs.uchicago.edu)

Test Assertions

- CU_ASSERT_FATAL
- CU_ASSERT_TRUE_FATAL
- CU_ASSERT_FALSE_FATAL
- CU_ASSERT_EQUAL_FATAL
- CU_ASSERT_NOT_EQUAL_FATAL
- CU_ASSERT_PTR_EQUAL_FATAL
- CU_ASSERT_PTR_NOT_EQUAL_FATAL
- CU_ASSERT_PTR_NULL_FATAL
- CU_ASSERT_PTR_NOT_NULL_FATAL
- CU_ASSERT_STRING_EQUAL_FATAL
- CU_ASSERT_STRING_NOT_EQUAL_FATAL
- CU_ASSERT_NSTRING_EQUAL_FATAL
- CU_ASSERT_NSTRING_NOT_EQUAL_FATAL
- CU_ASSERT_DOUBLE_EQUAL_FATAL
- CU_ASSERT_DOUBLE_NOT_EQUAL_FATAL

`_FATAL` versions immediatately end the current test function. Beware of memory leaks, uninitialized variables, etc. that might result from the code skipped due to a `_FATAL` assert!

See varstring_unit.c in hw4 directory
See arb_unittest.c in hw5 directory

Unit Testing

Definition

Unit test: automated piece of code which checks single assumptions about the behaviour of small pieces of code (often single logical functions).

In software industry, experience with testing code is considered a big benefit!

```
uint32_t myhtonl(uint32_t in);

// Subset: see documentation
CU_ASSERT_TRUE(somexpr);

CU_ASSERT_EQUAL(myhtonl(10),
                htonl(10));

// Note: for cunit, remember to
// link with cunit library
```

See `http://cunit.sourceforge.net/index.html`

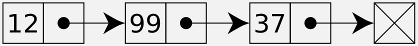
- As your projects become more complex, unit testing will save you work. *Especially true for HW6!*
- They provide (some) level of confidence that certain functions behave as expected, simplifying debugging.
  - Very useful when switching to a new compiler or new machine!

Demo: testmyhton

What are linked lists

Linked List

A linked list is a data structure (i.e. it stores data). There are multiple kinds (single, double, circular, skiplist, . . .) but the principle is the same: a list *node* holds a pointer to another node belonging to the same list.



Typical operations:

- Query the size of the list
- Insert an item at a specific position
- Remove an item
- Search for an item
- . . .

The type used to represent the list might or might not be a pointer (cfr. hw5). In general: the exact interface does not matter and might differ between implementations; The available operations and their time&space complexity will not.

Why linked lists?

Why not use arrays?

Linked lists and arrays serve different purposes and have different strengths and weaknesses. Some examples:

- Arrays are less flexible
  - In order to add elements, you might have to create a new array and copy all existing elements.
    - This would invalidate any pointers to existing array elements!
  - Adding elements in the middle creates similar issues
- Arrays require contiguous memory blocks
- Linked lists are flexible, but generally are slower to access and have higher overhead.
  - Might need to traverse the list to get to the n^{th} element.
  - We need to store link information

Forward Declarations and Incomplete types

Linked Lists in C

```
1  // Struct without tag
2  typedef struct {
3      int a;
4  } MyStruct;
5
6  MyStruct a;
7
8  // Need tag to refer to self
9  struct Link
10 {
11     int data;
12     struct Link * next;
13 };
```

- The struct on line 2 does not have a tag (which is OK)
- In order to link to other structs of the same type, a tag is needed (line 9)

Generic Data Structures

```c
// Single-linked list storing
// void * pointers
struct ListItem
{
    void * data;
    struct ListItem * next;
};
```

- Other than `void *`, there is no good way to provide 'generic' data structures.
- Decide (and document) if you are storing a value type or not. Consequences if not storing value types!
  - Destruction, copy, initialization
  - Operations such as testing for equality or partial order.
- By using `void *`, we no longer can rely on the compiler to catch type errors.

Linked list implementation demo

Linked Lists

There are many different kinds of linked lists (see Chapter 5 in the O'Reilly book); Considered on-topic:

- Singly-linked list
- Doubly-linked list
- Circular linked list

There are other minor variations; For example:

- Explicitly storing and maintaining the size of the list
- Keeping track of the list tail and/or head
- Availability of non-optimal operations (such as moving backwards in a singly-linked list)

Make sure you understand the complexity of the particular implementation of a data structure before using it!
(You need it anyway to determine the complexity of your own code)

Value Types

Value Types and Reference Types

In general, there are two kinds of types (not meaning 'C datatype' here but logical type): *value* type and *reference* type.

```
// Example value type
int a = 10;
b = a;
++b; // b != a

// Example reference type
FILE * f = fopen(...);
FILE * f2 = f;
fgetc(f);
// f2 is modified as well!
```

- value types (example: POD types in C++) are fully contained in the memory the underlying (C) type holds. They do not refer to outside data.
- Because of this, value types can be created, copied and 'destroyed' without special considerations. (Compare with FILE * which needs to be properly closed)
- The underlying C type for a reference type does not hold (or only partially holds) the state for the logical type. It cannot be copied by assignment (since this would not copy the external state).

Modular Code

Example

Good code can easily be reused. An often overlooked property is that ideally, changes in the implementation details of the code should not affect users of that code.

```
// Library for employee management
// header
struct Person
{
    int age;
};

// User of library
struct Person p;
// bad!
printf ("%i", p.age);
```

This is not good reusable code.

- Can never change struct Person (for example store age elsewhere)
- (Example) Can not add access control or logging.

Person: Improved

```
// in header:
struct Person { int age; }
void setAge (Person * p, int age);

// in main
struct Person p;
setAge(&p, 21);
```

Possible issues:

- Can look in header and access age directly.
- Can assume Person to be value type.

Modular Code

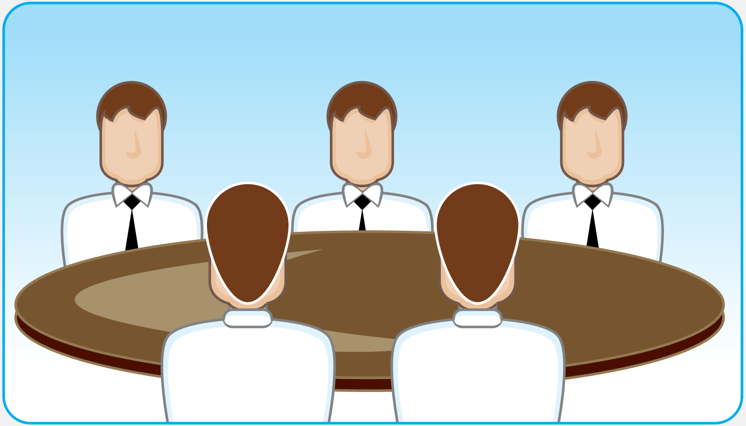
Better: Encapsulation

```
// In java:
//    p.setAge(10);

// Update age
void setAge (struct Person * p, int a);

// Better
void setAge (Person * p, int age);

// Even better
void setAge (Person p, int age);
```

- Object-oriented principle: encapsulation
  - Information hiding
  - Only provide information needed (can't be 'abused')
- C does not support C++/Java syntax
  - but we can emulate it

Quiz

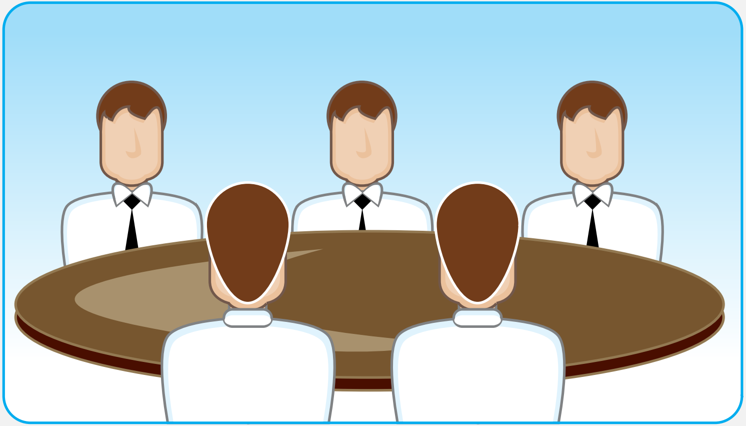
Discussion



(review Quiz questions)

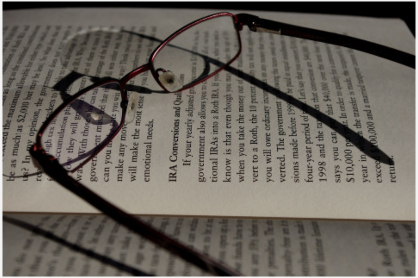
Homework 5

Discussion



What data structures did you use?

Reading Assignment



Reading Assignment

O'Reilly Mastering Algorithms in C:

Required Chapter 5, 6, 7