CS 24000 - Programming In C

Week Six: Review for Midterm 1

Zhiyuan Li
Department of Computer Science
Purdue University, USA
Other Unary operators

• The *indirection operator “*” is also called the “pointer dereference” operator

  * E

  – E is a pointer to

    • a piece of data, e.g. scalar, array, structure, etc
      – If E points to a simple variable (i.e. scalar, array element, structure member that is a scalar, etc, then *E has an lvalue

    • or a function

  – What is &*p?

  – Let’s examine &*p and p and see if they are the same (run cast.c)
• The `!` operator (the negation operator)
  – Operand must be of *arithmetic type* or be a *pointer*, and the result is 1 if the value of its operand equal to 0, and 0 otherwise. The type of the result is `int`.
  – Thus, `!p` has the value of 0 for all non-NULL pointer `p`.
• Unary `+` and `–`
  – Their mathematical meaning is obvious
  – The nuance concerning the data size of type promotion will be discussed later in the semester
• What is the value of `-(x)` and `-- x`
• What is the value of `!(x)` and `!!x`
Arithmetic binary operations (by precedence levels)

- **Multiplicative Operators**
  - *, /, and %
  - Left associative

- **Additive Operators**
  - +, -
  - Left associative

- **Note on pointer arithmetic**
  - A special kind of addition concerning pointers
  - p + int_expression
  - int_expression is first converted to an address offset by multiplying it by the size of the object to which the pointer points.
  - The sum is a pointer of the same type as the original pointer, and points to another object in the same array, appropriately offset from the original object.
    - We have explained before
• The next level is Shift Operators
  \[ E_1 \gg E_2 \]
  E1 (interpreted as a bit pattern) right-shifted E2 bits
  \[ E_1 \ll E_2 \]
  E1 (interpreted as a bit pattern) left-shifted E2 bits

• Left associative

• Integral type for both E1 and E2
  • \[ a + b \gg c + d \]
  • Both additions are performed first before doing >>

• The impact on the sign bit and the type promotion will be discussed later in the semester

• We may use shift operations extract control bits from control words in future labs/projects
Relational and Logic operations

**Listed by Precedence Levels**

- **Relational operators:** `<`, `<=`, `>`, `>=`
  - Compare any two types and return 1 for true and 0 for false
- **Equality operators** `==`, `!=`
  - `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value.
- **bit-wise operators**
  - bit-wise AND
  - bit-wise xor
  - bit-wise OR
- **Logical AND OR** `&&` higher than `||`
  - Operands: 0 is false, all other values true
  - Result: return 0 or 1
Short-circuit evaluation of logical expressions

• Evaluation order follows the precedence levels and appropriate associativity

• Evaluation terminates as soon as the final truth value can be determined, as in Java
  • w/o performing the rest of the operations

\[
1 \land \lor 0 \rightarrow 0 \quad 1 \lor 0 \rightarrow 1 \quad !42 \rightarrow 0
\]

\[
2 > 10 \land (a < b \lor c < d) \rightarrow 0 \quad \text{Not performing } \&, \<, \lor
\]
\[
a != 0 \land c/a > 4 \quad \text{if } a \text{ is } 0, \text{ then } c/a \text{ not performed}
\]
\[
p \land \ast p
\]
  • If \( p \) turns out to be a null pointer, return 0 without evaluation \( \ast p \)
• Next level, conditional operator
  \[E1 \ ? \ E2 : E3\]
• \(E1\) is first evaluated (possibly generating side effects)
• If \(E1\) evaluates to true then evaluate \(E2\), whose result will be the result of the entire expression
• If \(E1\) evaluates to false then evaluate \(E3\), whose result will be the result of the entire expression
• Further lower level, assignment operators
  =, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=
  – These are binary operators
  – Left operand must have an lvalue
• \( E_1 \text{ op=} E_2 \) is equivalent to \( E_1 = E_1 \text{ op} (E_2) \) except that \( E_1 \) is evaluated only once
  – The equivalence is true only if \( E_1 \) does not have any side effect
  – Is “\(*p++ += 1;\)” same as “\(*p++ = *p++ + 1;\)”?
Statements

• Many statements in C are like those in Java
The switch statement allows multi-way branching, it takes an integer valued expression and a number of constant-labeled branches.

Syntax

```java
switch (expression) {
    case const-expr: statements
    case const-expr: statements
    default: statements
}
```

Without the break, the execution would have continued to the following statements.
Example: use of *break* and *default*

```c
#include <stdio.h> /* students should experiment with overlapping cases and no breaks */

int main() {

    int i, odd=0, even=0;
    for (i=0;i<10;i++)
        switch ( i ) {
            case 0: case 2: case 4: case 6: case 8: case 10:
                even++;
                break; /* important to break */
            case 1: case 3: case 5: case 7: case 9:
                odd++;
                break; /* important to break */
            default: odd++;
        }
    printf("odd numbers \t%d\n", odd);
    printf("even numbers \t%d\n", even);
}
```

Break and continue

- **break** leaves the current loop or **switch**, **continue** goes to the top of the loop

```java
while (1) {
    for(int i=0; i<10; i++) {
        if (i&1) continue;
        if (i==8) break;
    }
}
```
Goto and labels

goto can jump to any label in the current function
goto should be used very carefully and rarely

```c
while (1) {
    for(int i=0; i<10; i++) {
        if (i&1) continue;
        if (i==8) goto error;
    }
}

error:
    printf("oops");
```
Revisit a previous example of pointer expressions (keep track of pointers)

```c
#include <stdio.h>
main() {
    int c = 0, in = 0;
    char buf[2048]; char *p = buf;
    char x[10][10];

    while((c = getchar()) != EOF)
        *p++=c;
    *p++ = '\0'; p = buf;
    *( buf + 1) = 'c';
    *( x[0] + 1) = 'd';
    p[0] = 'a'; *buf = 'b';
    printf("*buf is \t %c\n", *buf);
    p++[0] = 'b';
    p++[0] = 'c';
    printf("p[0] is \t %c\n", p[0]);
    printf("p[1] is \t %c\n", p[1]);
    printf("p[2] is \t %c\n", p[2]);
    printf("p[0] address is \t %p\n", p);
    printf("x[0][1] is \t %c\n", x[0][1]);
    printf("buf address is \t %p\n", buf);
    printf("buffer is \t %s\n", buf);
}
```
• We now continue to discuss malloc() and related functions
• Lab 4, Lab 5, Project 1 and Project 2 will use these extensively
What is `size_t`?

- In `malloc()`, we request an amount of memory measured in a number of bytes.
- How large can this number, `s`, be?
  - Different system may have a different limit.
  - To make the program more portable, instead of declaring `s` to be of type `int` or `long` or any other primitive type, we use the type `size_t`.
  - What is exactly `size_t` is defined in another header file.
  - We can use `sizeof(size_t)` to see what it is.
    - E.g. on my machine `sizeof(size_t)` is 8, i.e. `s` itself can take 8 bytes.
malloc(size_t s)

- Allocates \( s \) bytes and returns a pointer to the allocated memory.
- Memory is not cleared (i.e. contents not set to 0)
- Returned value is a pointer to alloc’d memory or \textbf{NULL} if the request fails
- You must cast the pointer

\[
p = (\text{char}*) \text{malloc}(10); /* allocated 10 bytes */
\]
\[
\text{if}(p == \text{NULL}) \{ /*panic*/ \}
\]
**free(void* p)**

- Frees the memory space pointed to by `p`, which *must* have been allocated with a previous call to `malloc`, `calloc` or `realloc`.
- If memory was not allocated before, or if `free(p)` has already been called before, undefined behavior occurs.
- If `p` is `NULL`, no operation is performed.
- `free()` returns nothing /* run free.c below */

```c
char *mess = NULL;
mess = (char*) malloc(100);
free(mess);  *mess = 43;
```

FREE DOES NOT SET THE POINTER TO NULL
**free(void* p)**

- Frees the memory space pointed to by `p`, which **must** have been allocated with a previous call to `malloc`, `calloc` or `realloc`.

- If memory was not allocated before, or if `free(p)` has already been called before, undefined behavior occurs.

- Let’s compile and run `free2.c` below.

```c
#include <stdlib.h>
#include <stdio.h>
main () {
    char *mess = NULL;
    int *mint = NULL;
    mess = (char*) malloc(100);
    printf("mess is \t%p\n", mess);
    printf("&mess is \t%p\n", &mess);
    free(&mess);  
}
```
Compile and run free3.c

```c
#include <stdlib.h>
#include <stdio.h>
main () {
    char *mess = NULL;
    char *mint = NULL;
    mess = (char*) malloc(100);
    printf("mess is \t%p\n", mess);
    printf("&mess is \t%p\n", &mess);
    mint = mess;
    free(mess);
    free(mint);
}
```