Coding Review (I2P 2019)

Basic Syntaxes of C v1.0

Review of common syntaxes and concepts.

Operators

1. Explain the following code.

   ```
   int a;
   a = 1;
   ```

   Answer:

   We declare the variable `a` and assign the value `1` to it.

   Note that `=` is the assignment operator, not the equal sign we used to see in math.

   In math textbooks, the above operation is often written as:

   $$a := 1$$

   In algorithm textbooks, the operation is often notated as:

   $$a \leftarrow 1$$

2. What is the result of the following code?

   ```
   #include <stdio.h>

   int main(void) {
       int a = 0;
       if (a = 5)
           puts("a is 5");
       return 0;
   }
   ```

   Answer:

   ```
   a is 5
   ```

   `==` is not `=`, The `if` condition is satisfied for anything else than `0`. The code above is like:

```
if (a = 5)  // a becomes 5
if (a)  // Read from a
if (5)
True
```

3. What is the result of the following code?

```
#include <stdio.h>

int main(void)
{
    int a = 1, b = 2;
    a = b = 3;
    printf("%d %d\n", a, b);
    return 0;
}
```

Answer:

```
3 3
```

The right operand is evaluated first, so it can be seen as:

```
a = (b = (3)) // (a, b) == (1, 2)
a = (3) // (a, b) == (1, 3)
3 // (a, b) == (3, 3)
```

4. What is the result of the following code?

```
#include <stdio.h>

int main(void) {
    int a = -1;
    puts("a is -1");
    if (-3 < a < 3)
        puts("-3 < a < 3 is True");
    else
        puts("-3 < a < 3 is False");
    if (-3 < a < 0)
        puts("-3 < a < 0 is True");
    else
        puts("-3 < a < 0 is False");
    return 0;
}
```

Answer:

```
a is -1
-3 < a < 3 is True
-3 < a < 0 is False
```

When evaluating an expression, be aware of the operator precedence and the associativity.

The result of expression is `1` if True, otherwise `0`.

In this part, it can be seen as:

```
if ((-3 < a) < 3)
if ((-3 < -1) < 3)
if (1 < 3)  // (-3 < -1) is True
if (1)  // (1 < 3) is True
True
```

```
if ((-3 < a) < 0)
if ((-3 < -1) < 0)
if (1 < 0) // (-3 < 1) is True
if (0) // (1 < 0) is False
False
```

To achieve the correct result, consider using `if (-3 < a && a < 0)`.

5. What is the result of the following code?

```
#include <stdio.h>
#define MAX 5

int main(void) {
    int i, a[MAX] = {1, 2, 3, 4, 5};
    for (i = 0; i < 1000000; i++) {
        if (i < MAX & a[i] < 3) {
            printf("%d\n", a[i]);
        }
    }
    return 0;
}
```

Answer:

Undefined Behavior. (May result in segmentation fault)

Assume `i` is 10. The if condition is evaluated as:

```
if ((i < MAX) & (a[i] < 3))
if ((10 < 5) & (a[i] < 3))
if (0 & (a[i] < 3))
if (0 & (a[10] < 3))
if (0 & (??? < 3))  // ??? here indicates an invalid memory access.
Undefined Behavior.
```

If `&` is changed to `&&`, the code's result will be:

```
1
2
```

since `&&` has short-circuit evaluation.

6. What is the result of the following code?

```
#include <stdio.h>
#define MAX 100

int main(void) {
    int i, top = -1, len = 0;
    int a[MAX], b[MAX];
    for (i = 1; i < 4; i++) {
        a[++top] = i;
        b[len++] = i;
    }
    for (i = 0; i < len; i++) {
        printf("a[%d] is %d\n", i, a[i]);
        printf("b[%d] is %d\n", i, b[i]);
    }
    return 0;
}
```

Answer:

```
a[0] is 1
b[0] is 1
a[1] is 2
b[1] is 2
```

```
a[2] is 3
b[2] is 3
```

Be aware of pre-increment and post increment.

- `++a` can be seen as:

  ```
  a += 1
  return a;
  ```

- `a++` can be seen as:

  ```
  int tmp = a;
  a += 1;
  return tmp;
  ```

By the way, you've just implemented the Stack data structure!

7. What is the result of the following code?

```
#include <stdio.h>

int main(void) {
    printf("The size of char is %zu\n", sizeof(char));
    printf("The size of short is %zu\n",sizeof(short));
    printf("The size of int is %zu\n", sizeof(int));
    printf("The size of long is %zu\n", sizeof(long));
    printf("The size of long long is %zu\n", sizeof(long long));
    printf("The size of unsigned long long is %zu\n", sizeof(unsigned long
long));
    printf("The size of float is %zu\n", sizeof(float));
    printf("The size of double is %zu\n", sizeof(double));
    return 0;
}
```

Answer:

The answer should be something like:

```
The size of char is 1
The size of short is 2
The size of int is 4
The size of long is 4
```

```
The size of long long is 8
The size of unsigned long long is 8
The size of float is 4
The size of double is 8
```

The numbers may vary depending on your compiler and hardware.

The C99 standard only defines the minimum size of each variable type. So the maximum size may vary depending on implementation.

Conditional Statement

1. What is the result of the following code?

```
#include <stdio.h>

int main(void) {
    int a = 3;
    if (a < 0);
    {
        puts("a < 0?");
    }
    return 0;
}
```

Answer:

```
a < 0?
```

A semicolon after `if`, `while`, `for`, ... can be seen as a `{}`. To avoid bugs, it may be better to just use `{}` instead of `;` for empty `if`s or `for`s.

2. What is the result of the following code?

```
#include <stdio.h>

int main(void) {
    int a = 3;
    switch(a) {
        case 1: puts("1");
        case 2: puts("2");
        case 3: puts("3");
        case 4: puts("4");
        case 5: puts("5");
        default: puts("Default");
    }
    return 0;
}
```

Answer:

```
3
4
5
Default
```

The rationale on `switch` is it compares the target variable with each case and continue executing code after the matched case. If no `break` is specified before the next case, it'll continue executing and won't jump out the `switch`.

We'll use `case ?: ... break;` most of the time.

3. What is the result of the following code?

```
#include <stdio.h>

int main(void) {
    int a, b = 0, c;
    c = ((a=3<4)*4>5?7:(b=5)<<1%10?117:87);
    printf("%d %d %d\n", a, b, c);
    return 0;
}
```

Answer:

```
1 5 117
```

If you didn't get the answer right, it's totally okay. The code above is a piece of bad code.

Keep in mind that:

- `... ? ... : ...` (The short-hand notation of `if`) may be suitable in some case, but often it just decrease the readability.
- If you're not sure with the operator precedences, adding redundant parentheses is totally fine.

4. What is the result of the following code?

```
#include <stdio.h>

int main(void) {
play:
    puts("play");
    goto play;
    return 0;
}
```

Answer:

Print `play` forever.

```
play
play
play
...
```

Abusing `goto` may easily make your program unmaintainable. But it might be useful when you want to break out a deep, nested loop.

Loops

1. What is the result of the following code?

```
#include <stdio.h>

int main(void) {
    int i = 0;
    while (i < 3) {
        printf("%d\n", i);
        i++;
    }
    for (int j = 0; j < 3; j++) {
        printf("%d\n", j);
    }
    return 0;
}
```

Answer:

```
0
1
2
0
1
```

```
2
```

The `for` loop above works exactly the same as the `while` loop above, they are interchangeable.

2. What is the result of the following code?

```
#include <stdio.h>

int main(void) {
    int T = 3;
    while (T--)
        printf("T is %d\n", T);
    printf("final T is %d\n", T);
    T = 3;
    while (--T)
        printf("T is %d\n", T);
    printf("final T is %d\n", T);
    return 0;
}
```

Answer:

```
T is 2
T is 1
T is 0
final T is -1
T is 2
T is 1
final T is 0
```

`while (T--)` may be useful when you want to repeat doing something for exactly `T` times.

3. What is the result of the following code?

```
#include <stdio.h>

int main(void) {
    int T = 0;
    do {
        printf("%d\n", T);
    } while (T--);
    int T2 = 0;
    printf("%d\n", T2);
    while(T2--) {
        printf("%d\n", T2);
    }
    return 0;
}
```

Answer:

```
0
0
```

The `do-while` loop above works exactly the same as the `printf` + `while` loop above, they are interchangeable.

The `do-while` loop may be rarely seen, but it can be useful when asking for inputs.

The following code read inputs until the user enters a positive number. If the input is invalid, the program will continue to ask for a valid input.

```
do {
    puts("Please enter a positive number");
    scanf("%d, &input);
} while (input < 0);
// Continue the program.
```

4. What is the result of the following code?

```
#include <stdio.h>

int main(void) {
    int i, j;
    for (i = 0; i <= 100; i += 10) {
        printf("i is %d\n", i);
        for (j = 0; j < 10; j++) {
            if (j % 2) continue;
            printf("j is %d\n", j);
        }
        if (!(i % 10)) break;
    }
    printf("final i is %d\n", i);
    printf("final j is %d\n", j);
    return 0;
}
```

Answer:

```
i is 0
j is 0
j is 2
j is 4
j is 6
j is 8
final i is 0
final j is 10
```

Three points worth noting:

- `continue` ignores the current iteration and continue running the `for` loop. You can think of it as jumping to the `}` part of the `for`.
- `break` directly breaks (exits) the `for` loop.
- There's a final `j++` before testing the exit condition. This can be useful to determine whether `break` is executed in the `for` loop.

  ```
  for (int i = 0; i < N; i++) {
      if (...)
          break;
  }
  if (i == N) {
      puts("break didn't occur");
  }
  ```

Scope and Lifetime of Variables

We'll only discuss variables in a single file.

These concepts in multiple files are much more complicated, they'll be covered in the final project.

1. What is the result of the following code?

```
#include <stdio.h>

int a;

void func(int a) {
    printf("%d\n", a);
    {
        extern int a;
        printf("%d\n", a);
    }
    printf("%d\n", a);
    a = 4;
    printf("%d\n", a);
}
```

```
int main(void) {
    printf("%d\n", a);
    int a = 1;
    printf("%d\n", a);
    {
        int a = 2;
        printf("%d\n", a);
    }
    printf("%d\n", a);
    {
        extern int a;
        printf("%d\n", a);
    }
    printf("%d\n", a);
    a = 3;
    printf("%d\n", a);
    func(a);
    printf("%d\n", a);
    return 0;
}
```

Answer:

```
0
1
2
1
0
1
3
3
0
3
4
3
```

Some points worth noting:

- The initial value of global variables is 0.
- Local (auto) variables has indeterminate initial value if not specified, and if they have the same name, the local variable is accessed instead of the global variable (variable scope).
- `extern` is used mostly in multiple files, to access variables in another file. In this case, it is used to access the global variable.
- A variable can only be accessed in the brackets (`{}`) it is declared in (variable lifetime), when outside the bracket, the variable is released.
- The rules in the `main` function is the same in other functions.

2. What is the result of the following code?

```
#include <stdio.h>

int a = 117;

void func1() {
    int a = 0;
    a++;
    printf("func1 a is %d\n", a);
}

void func2() {
    static int a = 0;
    a++;
    printf("func2 a is %d\n", a);
}

void func3() {
    static int a;
    a = 0;
    a++;
    printf("func3 a is %d\n", a);
}

int main(void) {
    for (int i = 0; i < 3; i++)
        func1();
    for (int i = 0; i < 3; i++)
        func2();
    for (int i = 0; i < 3; i++)
        func3();
    printf("main a is %d\n", a);
    return 0;
}
```

Answer:

```
func1 a is 1
func1 a is 1
func1 a is 1
func2 a is 1
func2 a is 2
func2 a is 3
func3 a is 1
func3 a is 1
func3 a is 1
main a is 117
```

`static` variables' value persist throughout the entire lifetime of the program. However, its scope is still limited in the brackets `{}`.

Compiler Preprocessor (`#define`)

1. What is the result of the following code?

```
#include <stdio.h>

int a = 0;
#define a b
int a = 1;

int main(void) {
    printf("a is %d\n", a);
    printf("b is %d\n", b);
#undef a
    printf("a is %d\n", a);
    printf("b is %d\n", b);
    return 0;
}
```

   Answer:

```
a is 1
b is 1
a is 0
b is 1
```

   `#define` only affects tokens. (e.g. variables)

2. What is the result of the following code?

```
#include <stdio.h>

int a = 10;

int func() {
    printf("%d\n", a);
}

#define a 117;

int main(void) {
    int b = a + 3;
    printf("%d\n", b);
    func();
    return 0;
}
```

Answer:

```
117
10
```

Some points worth noting:

- trailing semicolon in `#define` has its meaning.

  In the case above:

  ```
  int b = a + 3;
  int b = 117; +3;
  ```

  The `+3;` has no affect on `b`, it's just like `1;`.

- `#define` does not follow your program flow, since the replacement is done before compile.

3. What is the result of the following code?

```
#include <stdio.h>
#define ADD(X, Y) X+Y

int main(void) {
    int result;
    result = 2 * ADD(1, 2) * 4;
    printf("%d\n", result);
    return 0;
}
```

Answer:

```
10
```

You can see `#define` as replacing tokens. So if you want it to get the answer `24`, you should add parentheses `#define ADD(X, Y) (X+Y)`.

4. What is the result of the following code?

```
#include <stdio.h>
#define ever (;;)
#define forever(x) \
    for ever { puts(x); }

int main(void) {
    forever("alone");
    return 0;
}
```

Answer:

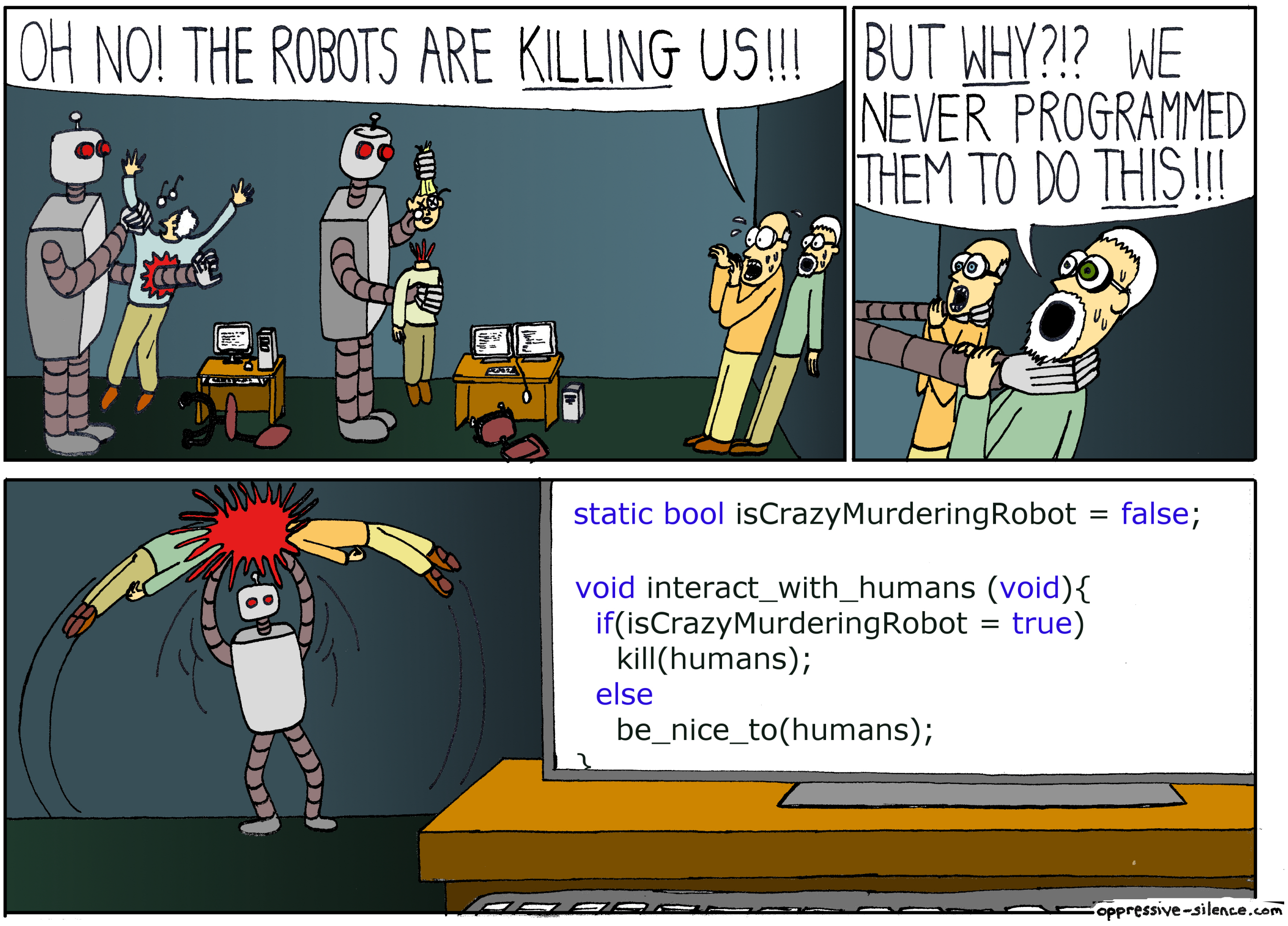
Print `alone` forever.

```
alone
alone
alone
...
```

`for(;;)` runs forever.

For the next assignment, we'll review some basic recursions.

Epilogue


OH NO! THE ROBOTS ARE KILLING US!!!
BUT WHY?!? WE NEVER PROGRAMMED THEM TO DO THIS!!!
oppressive-silence.com

```
static bool isCrazyMurderingRobot = false;

void interact_with_humans (void){
  if(isCrazyMurderingRobot = true)
    kill(humans);
  else
    be_nice_to(humans);
}
```

Photo Credit: on Oppressive Silence site by Ethan Vincent.

If there's any typo, please discuss on iLMS or email `j3soon@gapp.nthu.edu.tw`, I appreciate your help.