Kahoot (1)

lec2-kahoot-decl.c

```c
/* This is a simple printf example. */

int main(void) {
    printf("hello, world.\n");
    return 0;
}
```

Which statement describes this program the best?

Kahoot (2)

Every Unix program returns a exit code. When a program encounters an error it will return:

Kahoot (3)

lec2-kahoot-struct.c

```c
struct tuple {
    int first;
    int second;
}

int main(void) {
    tuple t;
    t->first = 1;
    return 0;
}
```

How many errors are there in this program?

Kahoot (4)

lec2-kahoot-mult-defs.c

```c
#include <stdio.h>

int main(void) {
    printf("hello!\n");
    return 0;
}

int main(void) {
    printf("hello, world.\n");
    return 0;
}
```

What sentence describes this program best?

Kahoot (5)

lec2-kahoot-printf.c

```c
#include <stdio.h>

int main(void) {
    printf("I like %c\n", "C");
    return 0;
}
```

We compile this program with `gcc lec2-kahoot-printf.c`
Which statement is the most accurate?

Kahoot (6)

lec2-kahoot-sum.c

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Sum: %d\n", argv[1] + argv[2]);
    return 0;
}
```

This program should print the sum of the two arguments.
What can you say about this program?

Errata and Questions from Lab

- Inconsistent `*` placement in `stack.h` and `stack.c`
- Inconsistent parameter naming in `stack.h` and `stack.c`
- `Coding_standard`: `make tarball` is the recommended way to create the tar file.
- `Coding_standard` still mentioned `AUTHOR` file.
- `#define STACK_SIZE 100` defines a constant (in a rather crude way).
- Things that start with a # are preprocessing directives.
- They are processed by cpp.
- No semicolon at the end of a preprocessor directive!
- Semicolon at the end of struct declaration!
- Requirements state: stack size is limited to a fixed number.
- The stack API pushes and pops integers.
- Although you cannot store -1 on the stack.

Printing to different output streams

- The first assignment asks you to print statistics to standard error.
- Every UNIX process gets three streams for free:
  - `stdin` (0)
  - `stdout`(1)
  - `stderr` (2)
- `stderr` is often used for diagnostic messages and warnings.
- `printf` prints to `stdout`
- Want to use a different stream?
  → Use `printf` little brother `fprintf()`
- `fprintf` takes the stream to print to as the first argument.
- Otherwise works the same way as `printf`

fprintf example

fprintf.c

```c
#include <stdio.h>
/* printing to stdout and stderr */

int main(void) {
   printf("hello, world.\n");
   fprintf(stderr, "This line is sent to stderr\n");
   fprintf(stdout, "standard output again\n");
   return 0;
}
```

Exciting, let's check the output of this program!

Variable definitions

- General syntax to declare a new variable:
  `type name;`
  - Define an integer variable: `int i;`
  - Define an integer array: `int data[10];`
- Declare and set an initial value:
  `type id = expr;`
  - Define and init an integer variable: `int i = 0;`
  - Define and init an int array variable:
    `int data[] = {1,2,3};`
- Declare multiple variables at once.
  - Syntax: `type id, id;`
  - Define two integers: `int i1, i2;`
- And you can initialize them as well:
  - Syntax: `type id = expr, id = expr;`
  - Example: `char c1 = ’a’, c2 = ’b’;`
- Function parameters are variables as well!
  `int main(int argc, char* argv[])`

Uninitialized variables

dontrunme.c

```c
int main(void) {
    int a;
    return a;
    a = 42;
    return 0;
}
```

- Note: you can return in the middle of a function body.
- *Don't use a variable before it is initialized!*

Uninitialized variables (2)

Don't use uninitialized variables!

Uninitialized variables (3)

dontrunme.c

```c
int main(void) {
    int a;
    return a;
    a = 42;
    return 0;
}
```

- Why doesn't C just initialise the variable a for us?
- **Performance**: setting variables to default values takes time.

Composite types: `struct`'s

- Applications often model things with multiple attributes.
- Examples: a person or a stack.
- A structure neatly organizes these attributes in a single place.
- Like a "class" in Java or Python.
- But without methods and inheritance.
- Structure type definition:

```
struct label {
    type member1;
    type member2;
    ..
}; // <--- don't forget the semi colon!
```

- Defines a struct type "`struct label`".
- Declare a variable `x` with: `struct label x;`
- Access `member1` in the struct `x` with: `x.member1`

struct example

person.c

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

struct person {
    char *name;
    int age;
    int nums[3];
};

int main(void) {
    struct person joe = malloc(sizeof(struct person));
    joe->name = "joe";
    joe->age = 34;
    joe->nums[0] = 12;
    joe->nums[1] = 13;
    joe->nums[2] = 42;
    printf("Name: %s, age: %d, lucky nums: %d, %d, %d\n", joe->name, joe->age
           joe->nums[0], joe->nums[1], joe->nums[2]);
    free(joe);
    return 0;
}
```

Person example C and Java

person.h

```
struct person {
   char *name;
   int age;
   int nums[3];
};
```

Person.java

```
public class Person {
    String name;
    int age;
    int[] nums;

    public Person() {
        name = "joe";
        age = 34;
        nums = new int[3];
    }
}
```

Person example C and Java (2)

personmain.c

```c
#include <stdio.h>
#include "person.h"
int main(void) {
   struct person joe;  // joe is the actual struct object
   joe.name = "joe";
   joe.age = 34;
   joe.nums[0] = 12; joe.nums[1] = 13; joe.nums[2] = 42;
   printf("Name: %s, age: %d, lucky nums: %d, %d, %d\n", joe.name, joe.age,
           joe.nums[0], joe.nums[1], joe.nums[2]);
   return 0;
}
```

PersonExample.java

```java
public class PersonExample {
    public static void main(String args[]) {
        Person joe = new Person();  // Create and assign reference to joe
        joe.nums[0] = 12; joe.nums[1] = 13; joe.nums[2] = 42;
        System.out.printf("Name: %s, age: %d, nums: %d, %d, %d\n", joe.name,
                joe.age, joe.nums[0], joe.nums[1], joe.nums[2]);
    }
```

Person example with reference to struct

personmain-ref.c

```
#include "person.h"
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    struct person *joe = malloc(sizeof(struct person)); // joe is ref to struct
    if (!joe) {
        return 1;
    }
    joe->name = "joe";
    joe->age = 34;
    joe->nums[0] = 12;
    joe->nums[1] = 13;
    joe->nums[2] = 42;
    printf("Name: %s, age: %d, lucky nums: %d, %d, %d\n", joe->name, joe->age
           joe->nums[0], joe->nums[1], joe->nums[2]);
    free(joe);
    return 0;
}
```

Automated workflow

make-example/hello.c

```c
int a = 124;
```

make-example/world.c

```c
#include <stdio.h>
extern int a;

int main(void) {
    printf("hello %d\n", a);
    return 0;
}
```

Automated workflow with scripts

- A solution: write a script to automate the compilation.
- `compile.sh` contains the gcc commands.
- Much better than typing in the commands.
- There is a lot of repetition in the script.
- And it will always compile every source file.

Automated workflow with Makefiles

make-example/Makefile

```
CC=gcc
test: hello.o world.o
        $(CC) -o $@ $^

clean:
        rm -f *.o test a.out
```

- A Makefile specifies the dependency relations between files.
- The format is:

  ```
  target: dependencies ...
      action_rules
  ```

- Create a target with: `make <target>`
- Runs the commands to create that target.
- Builtin rules for common file types.
- `action_rules` must be indented with a tab character!

Korte intro Makefile: http://liv.science.uva.nl/dauto1.html#wat-doet-make-1-eigenlijk

References: a lightspeed introduction

This slide is just to help you read the framework code.

- The first assignment has functions that take references to structures as arguments and return structures.
- Unlike Java, C uses a special syntax to indicate a reference is used.
- `int stack_push(struct stack *stack, int e);`
- `struct stack *stack_init(void);`
- When dealing with a reference to a struct C also uses a special syntax to access its members.
-
  ```
  int stack_push(struct stack *stack, int e) {
   // ...
   if (stack->pos == STACK_SIZE) {
      // do something smart.
      // ...
  }
  ```
- Don't worry, will be explained fully later on.

The lab assignment: infix2rpn

- Two tasks: implement a data structure and an algorithm.
- Two roles: library developer and application programmer.
- Roles separated by Application Programming Interface (API)
- API is defined in the header file.
- Abstraction:
  data structure implemented → focus on algorithm.

The lab assignment: infix2rpn (2)

- Read assignment text at least twice!
- Data structure is a stack.
- Declared in `stack.h`
- Defined in `stack.c`
- Tested in `check_stack.c`
- Let's get started!

The lab assignment: infix2rpn (3)

- All assignments come with a framework of code.
- C files contain the comments:

  ```
  // ... SOME CODE MISSING HERE ...
  ```
- You can complete the assigment by only adding code.
- Writing C files from scratch is also OK.
- Don't change the header files!
- We use the framework header files for grading.
- So if your interface does not match ours, the grading tests will fail.

The lab assignment: infix2rpn (4)

- Unpack and build the assignment by running: `make`
- Run it with: `./infix2rpn`
- Examine `infix2rpn.c`
- Examine `stack.h`
- Examine and modify `stack.c`
- Run tests with: `make check`
- Examine test results.