

CS100 Recitation 5

Who? GKxx

When? March 21, 2022

Warmup

```
● luna@sappho:~$ bat -p c-js.c
#include <stdio.h>

int main() {
    puts("-0.5" + 1);
}
● luna@sappho:~$ gcc c-js.c && ./a.out
0.5
● luna@sappho:~$
```

Warmup

```
● luna@sappho:~$ bat -p c-js.c  
#include <stdio.h>
```

```
int main() {  
    printf("%d\n", 50 ** "2");  
}
```

```
● luna@sappho:~$ gcc c-js.c && ./a.out  
2500
```

```
● luna@sappho:~$ █
```

Contents

IO

Streams

Redirect to Files

File Input and Output

String as a Stream

Structures

Define a Structure

Operations on `struct`

Coding

Good Coding-style

Examples

Streams

Definition (Stream)

A **stream** is a sequence of characters read from or written to an IO device.

The term **stream** is intended to suggest that the characters are **generated**, or **consumed**, sequentially over time.

Streams

Definition (Stream)

A **stream** is a sequence of characters read from or written to an IO device.

The term **stream** is intended to suggest that the characters are **generated**, or **consumed**, sequentially over time.

Standard input and output streams: **stdin** and **stdout**.

- `scanf`, `gets`, `getchar`: read from `stdin`.
- `printf`, `puts`, `putchar`: write to `stdout`.

By default, `stdin` and `stdout` are directed to the console.

Contents

IO

Streams

Redirect to Files

File Input and Output

String as a Stream

Structures

Define a Structure

Operations on `struct`

Coding

Good Coding-style

Examples

Redirection

We can redirect the standard streams to files:

- Use `< filename` to redirect `stdin` to a file.
- Use `> filename` to redirect `stdout` to a file.

Redirection

We can redirect the standard streams to files:

- Use `< filename` to redirect `stdin` to a file.
- Use `> filename` to redirect `stdout` to a file.
- Example: `./program < test.in > test.out`
- The online grader redirects your program and compares the output file with the answer file.

Redirection

We can redirect the standard streams to files:

- Use `< filename` to redirect `stdin` to a file.
- Use `> filename` to redirect `stdout` to a file.
- Example: `./program < test.in > test.out`
- The online grader redirects your program and compares the output file with the answer file.
- Input from any file terminates with **EOF**!
- **EOF** is a special character `#defined` as `-1`.
- It is suggested to use `int` to store the return-value of `getchar`, why?

Redirection

Use freopen to redirect:

```
int main() {  
    freopen("in_file.txt", "r", stdin);  
    freopen("out_file.txt", "w", stdout);  
    // ...  
}
```

Redirection

Use freopen to redirect:

```
int main() {  
    freopen("in_file.txt", "r", stdin);  
    freopen("out_file.txt", "w", stdout);  
    // ...  
}
```

- `stdin` and `stdout` are redirected to "input_file.txt" and "output_file.txt" respectively.
- "r": read; "w": write;
- There are also some other open modes.

Contents

IO

Streams

Redirect to Files

File Input and Output

String as a Stream

Structures

Define a Structure

Operations on `struct`

Coding

Good Coding-style

Examples

File IO Functions

```
int main() {  
    FILE *in = fopen("in_file.txt", "r");  
    FILE *out = fopen("out_file.txt", "w");  
    int a, b;  
    fscanf(in, "%d%d", &a, &b);  
    fprintf(out, "%d\n", a + b);  
    printf("%d\n", a + b);  
    fclose(in);  
    fclose(out);  
    return 0;  
}
```

- FILE: a special type storing the information of a file.
- fscanf, fprintf, fgets, fputs, fgetc, fputc.
- Use fopen and fclose.

Contents

IO

Streams

Redirect to Files

File Input and Output

String as a Stream

Structures

Define a Structure

Operations on `struct`

Coding

Good Coding-style

Examples

String IO Functions

- sscanf: read data in an “scanf-way” from a string.
- sprintf: write data in a “printf-way” to a string.

```
// roundabout way, just for demonstration
int main() {
    char str[100];
    gets(str);
    int a, b;
    sscanf(str, "%d%d", &a, &b);
    char result[100];
    sprintf(result, "%d", a + b);
    puts(result);
    return 0;
}
```


Contents

IO

Streams
Redirect to Files
File Input and Output
String as a Stream

Structures

Define a Structure
Operations on `struct`

Coding

Good Coding-style
Examples

Define a struct

```
struct Tile {  
    int num;  
    char kind;  
};
```

Define a struct

```
struct Tile {  
    int num;  
    char kind;  
};
```

- A structure is a user-defined data type: `struct Tile`.
- We can define a variable of such type:

```
struct Tile t;
```

Define a struct

```
struct Tile {  
    int num;  
    char kind;  
};
```

- A structure is a user-defined data type: `struct Tile`.
- We can define a variable of such type:

```
struct Tile t;
```

- Use `member-access operator`:

```
t.num = 1;  
t.kind = 's';  
printf("%d\n", t.num);
```

Define a struct

An unnamed structure (which cannot be used after definition):

```
struct {  
    int num;  
    char kind;  
};
```

Define a struct

An unnamed structure (which cannot be used after definition):

```
struct {  
    int num;  
    char kind;  
};
```

Defining both a structure and a variable (**not suggested coding-style**):

```
struct Tile {  
    int num;  
    char kind;  
} t;
```

Use typedef

```
typedef long long LL;
```

Use `typedef`, so that we don't need the `struct` keyword everytime we use it.

```
typedef struct {  
    int num;  
    char type;  
} Tile;
```

Use typedef

Within the `typedef` declaration, you cannot refer to the type alia.

```
typedef struct {  
    int value;  
    Node *next;    // Error  
} Node;
```


Use typedef

Within the `typedef` declaration, you cannot refer to the type alia.

```
typedef struct {  
    int value;  
    Node *next;    // Error  
} Node;
```

Correct way: Give it a name first.

```
typedef struct _node_ {  
    int value;  
    struct _node_ *next;  
} Node;
```

Incomplete Type

You cannot define a member of the type itself:

```
struct Widget {  
    struct Widget w;  
    int x;  
};
```

- In syntax: during the definition, the type '`struct Widget`' is an **incomplete type**. It is not allowed to define a variable of an incomplete type.
- In semantics: What's the size of a '`struct Widget`'?

Memory Alignment

```
typedef struct {  
    int num;  
    char kind;  
} Tile;
```

```
sizeof(Tile) != sizeof(int) + sizeof(char)
```

- In most implementations, the structure above takes 8 bytes. The storage will be aligned to multiple of 4.

Contents

IO

Streams
Redirect to Files
File Input and Output
String as a Stream

Structures

Define a Structure
Operations on `struct`

Coding

Good Coding-style
Examples

Initialization

- **Default initialization** of a structure initializes every member by default (with an undefined value).
- **Value initialization** of a structure initializes every member by value-initialization (with all types of '0').

Initialization

- **Default initialization** of a structure initializes every member by default (with an undefined value).
- **Value initialization** of a structure initializes every member by value-initialization (with all types of '0').
- **Copy initialization**: `Title a = b;` copies the value of each member of `b` to `a`.
- `b` must be of type `Title`.

Copy-assignment

```
Tile a, b;  
a.num = 1; a.kind = 's';  
b = a;
```

The [assignment operator](#) is generated by the compiler, which copies the value of each member of RHS to LHS.

A Unique Type

Every structure is a unique type.

A Unique Type

Every structure is a unique type.

```
typedef struct {  
    int num;  
    char kind;  
} Fake_tile;  
Fake_tile ft;  
ft = a; // Error  
Fake_tile ft2 = a; // Error
```

Fake_tile and Tile are **different types**, even though their definitions look the same.

Parameter and Return-value

```
Tile next_tile(Tile t) {  
    Tile next;  
    next.num = t.num + 1;  
    next.kind = t.kind;  
    return next;  
}
```

Parameter and Return-value

```
Tile next_tile(Tile t) {  
    Tile next;  
    next.num = t.num + 1;  
    next.kind = t.kind;  
    return next;  
}
```

- When passing as an argument, it is in fact **copy-initializing** the parameter `Tile t`.
- When returning from a function, it is in fact **copy-initializing** the temporary object generated by the calling expression. (In C, and before C++11)

Parameter and Return-value

How many copies are there?

```
Tile next_tile(Tile t) {  
    Tile next = t;  
    ++next.num;  
    return next;  
}  
  
int main() {  
    Tile tile, tile2;  
    tile.num = 1;  
    tile.kind = 's';  
    tile2 = next_tile(tile);  
    return 0;  
}
```

Parameter and Return-value

```
Tile next_tile(Tile t) {  
    Tile next = t;  
    ++next.num;  
    return next;  
}  
  
// in main  
Tile tile, tile2;  
tile.num = 1; tile.kind = 's';  
tile2 = next_tile(tile);
```

- copy-initialization of parameter t.
- copy-initialization of next;
- copy-initialization of a temporary object generated by next_tile(tile), with the value returned.
- copy-assignment to tile2.

Minor Optimization

Since `t` is a copy of the argument, we don't need another copy.

```
Tile next_tile(Tile t) {  
    ++t.num;  
    return t;  
}
```

Minor Optimization

Since `t` is a copy of the argument, we don't need another copy.

```
Tile next_tile(Tile t) {  
    ++t.num;  
    return t;  
}
```

- `t` is destroyed immediately the function `returns`.

Minor Optimization

Since `t` is a copy of the argument, we don't need another copy.

```
Tile next_tile(Tile t) {  
    ++t.num;  
    return t;  
}
```

- `t` is destroyed immediately the function `returns`.
- Why do we have to copy something **almost dead**, instead of simply **extending** its lifetime?

Minor Optimization

Since `t` is a copy of the argument, we don't need another copy.

```
Tile next_tile(Tile t) {  
    ++t.num;  
    return t;  
}
```

- `t` is destroyed immediately the function `returns`.
- Why do we have to copy something **almost dead**, instead of simply **extending** its lifetime?
- **C++11** solves this problem.

Dynamic Allocation

malloc and free as usual.

```
Tile *thetile  
    = (Tile *)malloc(sizeof(Tile));  
Tile *manytiles  
    = (Tile *)malloc(sizeof(Tile) * n);  
free(thetile); free(manytiles);
```

Dynamic Allocation

malloc and free as usual.

```
Tile *thetile
    = (Tile *)malloc(sizeof(Tile));
Tile *manytiles
    = (Tile *)malloc(sizeof(Tile) * n);
free(thetile); free(manytiles);
```

Access through pointers: dereference, and then access.

```
*thetile.num = 1;    // Error!
(*thetile).num = 1;  // Correct
thetile->num = 1;    // Preferred
```

Remark

The **member-access operator** has **higher** precedence than the **dereference operator**.

Contents

IO

Streams

Redirect to Files

File Input and Output

String as a Stream

Structures

Define a Structure

Operations on `struct`

Coding

Good Coding-style

Examples

Good Coding-style

- The simpler, the better.
- Code in a modern way.
- Strive to compile warning-free at the maximum warning level.
- At least understand every warning completely.
- Read masterpieces, and imitate.

Contents

IO

Streams
Redirect to Files
File Input and Output
String as a Stream

Structures

Define a Structure
Operations on `struct`

Coding

Good Coding-style
Examples