Imperative programming

Static program structure

Tamás Kozsik et al

Eötvös Loránd University

October 25, 2022

Table of contents

Static program structure

- Expressions
- Statements
- Sub-programs
- Modules

Module

- Bigger unit
- Great internal cohesion
- Tight interface
  - Weak connection between modules
  - Typically one direction

Build process

- **Preprocessing**
- **Compilation**
- **Linking**
- Optimization
- Code generation
- stb.

Preprocessing

- Preprocessing
- Textual replacements
- Its output is the translation unit
- `gcc -E main.c`
- Lines starting with #

Preprocessor statements

#define

- Definition of a constant value
- Can be defined during compilation
- No need to have a value
- Can be like a function

Preprocessor statements

#define

```
#include <stdio.h>

#define SIZE 10

int main()
{
  int arr[SIZE];
  int sum = 0;

  for (int i = 0; i < SIZE; ++i)
    sum += arr[i];

  printf("%d\n", sum);
}
```

Preprocessor statements

#define

```c
#include <stdio.h>

int main()
{
  int arr[SIZE];
  int sum = 0;

  for (int i = 0; i < SIZE; ++i)
    sum += arr[i];

  printf("%d\n", sum);
}
```

Fordítás

```
gcc -DSIZE=10 main.c
```

Preprocessor statements

#define

```
#define SQR(x) x * x
printf("%d\n", SQR(5));                  /* 25      */
```

Preprocessor statements

#define

```c
#define SQR(x) x * x
printf("%d\n", SQR(5));              /* 25     */
printf("%d\n", SQR(2 + 3));          /* Not 25 */
printf("%d\n", 2 + 3 * 2 + 3);       /* 11     */
```

Preprocessor statements

#define

```
#define SQR(x) x * x
printf("%d\n", SQR(5));                  /* 25     */

printf("%d\n", SQR(2 + 3));              /* Not 25 */
printf("%d\n", 2 + 3 * 2 + 3);           /* 11     */

#define SQR(x) (x) * (x)
printf("%d\n", SQR(2 + 3));
printf("%d\n", (2 + 3) * (2 + 3));       /* 25     */
```

Preprocessor statements

#define

```
#define SQR(x) x * x
printf("%d\n", SQR(5));              /* 25     */
printf("%d\n", SQR(2 + 3));          /* Not 25 */
printf("%d\n", 2 + 3 * 2 + 3);       /* 11     */
#define SQR(x) (x) * (x)
printf("%d\n", SQR(2 + 3));
printf("%d\n", (2 + 3) * (2 + 3));   /* 25     */
printf("%d\n", 100 / SQR(5));        /* Not 4  */
printf("%d\n", 100 / (5) * (5));     /* 100    */
```

Preprocessor statements

#define

```
#define SQR(x) x * x
printf("%d\n", SQR(5));              /* 25     */
printf("%d\n", SQR(2 + 3));          /* Not 25 */
printf("%d\n", 2 + 3 * 2 + 3);       /* 11     */
#define SQR(x) (x) * (x)
printf("%d\n", SQR(2 + 3));
printf("%d\n", (2 + 3) * (2 + 3));   /* 25     */
printf("%d\n", 100 / SQR(5));        /* Not 4  */
printf("%d\n", 100 / (5) * (5));     /* 100    */
#define SQR(x) ((x) * (x))
printf("%d\n", 100 / SQR(5));
printf("%d\n", 100 / ((5) * (5)));   /* 4      */
```

Preprocessor statements

#if, #ifdef, #ifndef, #else, #elif, #endif

```c
#include <stdio.h>

#ifdef __linux__
void f() {
  printf("This is Linux\n");
}
#elif _WIN32
void f() {
  printf("This is Windows\n");
}
#endif

int main() {
  f();
}
```

Preprocessor statements

#include

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

int main()
{
  srand(time(0));
  int* ptr = (int*)malloc(sizeof(int));
  *ptr = rand() % 100 + 1;
  printf("%d\n", *ptr);
  free(ptr);
}
```

Compilation

- Its input is the translation unit
  Source code (in a source file)
  - `factorial.c`
- Compiler
  - `gcc -c factorial.c`
- Its output is the object code (target code)
  - `factorial.o`

Compiler errors

- Violating language rules
- Compiler detects them

factorial.c

```
int factorial(int n)
{
  int result = 1;
  for (i = 2; i <= n; ++i)
    result *= i;
  return result;
}
```

gcc -c factorial.c

```
main.c: In function ‘factorial’:
main.c:4:8: error: ‘i’ undeclared (first use in this function)
    4 |   for (i = 2; i <= n; ++i)
      |        ^
```

Linking

- Its input are the object codes
  - `factorial.o`
- Linker
  - `gcc factorial.o`
- Executable code
  - `a.out` (default name)

Multiple translation units

factorial.c

```c
int factorial(int n)
{
  int result = 1;
  for (int i = 2; i <= n; ++i)
    result *= i;
  return result;
}
```

main.c

```c
#include <stdio.h>

int factorial(int n);

int main() {
  printf("%d\n", factorial(5));
}
```

Compilation, linking, execution

```
$ gcc -c factorial.c -o factorial.o
$ gcc -c main.c -o main.o
$ gcc factorial.o main.o -o a.out
$ ./a.out
$ gcc main.c factorial.c
```

Linking error

factorial.c

```c
int factorial(int n)
{
  int result = 1;
  for (int i = 2; i <= n; ++i)
    result *= i;
  return result;
}
```

main.c

```c
#include <stdio.h>

int faktorial(int n);

int main() {
  printf("%d\n", faktorial(5));
}
```

Compilation, linking, execution

```
$ gcc -c factorial.c main.c
$ gcc factorial.o main.o
main.o: In function `main':
main.c:(.text+0xa): undefined reference to `faktorial'
collect2: error: ld returned 1 exit status
```

Compile-time and run-time linking

Static linking

- Before execution
- Immediately after creating object code
- Advantage: linking errors in compile-time

Dynamic linking

- During execution
- Dynamically linked object code
  - Linux *shared object*: `.so`
  - Windows *dynamic-link library*: `.dll`
- Advantages
  - Smaller executable
  - Less memory usage

Headers

- Header file: `.h`
- Interface between modules
  - `extern`
  - nem `static`
- In the module and its client `#include`
  - Type matching
  - Preventing linking errors
- Preprocessor

Headers

Motivation

calc.h

```c
int factorial(int n);
int square(int n);
```

main.c

```c
#include <stdio.h>
#include "calc.h"

int main()
{
  printf("%d\n", factorial(5));
  printf("%d\n", square(5));
}
```

calc.c

```c
int factorial(int n)
{
  int result = 1;

  for (int i = 2; i <= n; ++i)
    result *= i;

  return result;
}

int square(int n)
{
  return n * n;
}
```

Include guard

Avoiding multiple inclusions

low_level_module.h

```
#ifndef LOW_LEVEL_MODULE
#define LOW_LEVEL_MODULE
// Some definition...
#endif
```

middle_module.h

```
#ifndef MIDDLE_MODULE
#define MIDDLE_MODULE
#include "low_level_module.h"
...
#endif
```

main.c

```
#include "low_level_module.h"
#include "middle_module.h"
```

Include guard

```
#ifndef VECTOR_H
#define VECTOR_H

#define VEC_EOK 0
#define VEC_ENOMEM 1

struct VECTOR_S { ... };
typedef struct VECTOR_S* vector_t;

extern int vectorErrno;

void* vectorAt(vector_t v, size_t idx);
void  vectorPushBack(vector_t v, void* src);

#endif
```

static, extern

positive.c

```c
int positive = 1;
static int negative = -1;
extern int increment;

static void compensate() {
  negative -= increment;
}
void signal() {
  positive += increment;
  compensate();
}
```

main.c

```c
#include <stdio.h>

int increment = 3;
extern int positive;
extern void signal();

int main() {
  signal();
  printf("%d\n", positive);
  return 0;
}
```