Modules, Makefiles, Editors, Git

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού
Κώστας Χατζηκοκολάκης
Creating large programs

- A large program might contain hundreds of thousands lines of code
- Having such a program in a single `.c` file is not practical
  - Hard to write
  - Hard to read and understand
  - Hard to maintain
  - Slow to compile
- We need to split it in semantically related units
Modules

- A module (ενότητα) is a collection of related data and operations
- They allow achieving abstraction (αφαίρεση), a notion of fundamental importance in programming
- The user of the module only needs to know what the module does
- Only the author of the module needs to know how it is implemented
- This is useful even when the author and the user are the same person
- They will be used to implement Abstract Data Types later in this course
Information Hiding

• A notion closely related to abstraction

• Since the user does not need to know how the module is implemented, anything not necessary for using the module should be hidden
  - internal data, auxiliary functions, data types, etc

• This allows modifying parts of the program independently
  - a function visible only within the module cannot affect other parts of the program
  - think of changing a car's tires, it should not affect its engine!
Modules in C

- A module in C is represented by a header file `module.h`
  - we already know several modules: `stdio.h`, `string.h`, ...

- It simply declares a list of functions
  - also constants and typedefs

- Describes what the module does
  - often with documentation for these functions
Modules in C

• E.g. A `stats.h` module with two functions

```c
#include <limits.h>     // INT_MIN, INT_MAX

// Επιστρέφει το μικρότερο στοιχείο του array (INT_MAX αν size == 0)
int stats_find_min(int array[], int size);

// Επιστρέφει το μεγαλύτερο στοιχείο του array (INT_MIN αν size == 0)
int stats_find_max(int array[], int size);
```

• Prefixing all functions with `stats_` is a good practice (why?)
Using a C module

- `#include "module.h"
- Use the provided functions
- As **users**, we don't need to know how the module is implemented!

```c
// minmax.c - Το βασικό αρχείο του προγράμματος

#include <stdio.h>
#include "stats.h"

int main() {
    int array[] = { 4, 35, -2, 1 };
    printf("min: %d\n", stats_find_min(array, 4));
    printf("max: %d\n", stats_find_max(array, 4));
}
```
Implementing a C module

• The module's implementation is provided in a file module.c
• module.c contains the definitions of all functions declared in module.h

```
// stats.c - Υλοποίηση του stats module

#include "stats.h"

int stats_find_min(int array[], int size) {
    int min = INT_MAX; // "default" τιμή, μεγαλύτερη από όλες
    for(int i = 0; i < size; i++)
        if(array[i] < min)
            min = array[i]; // βρέθηκε νέο ελάχιστο
    return min;
}
```
Compiling a program with modules

• Simply compiling `minmax.c` together with `module.c` works

```
gcc minmax.c stats.c -o minmax
```

• But this compiles both files every time

• What if we change a single file in a program with 1000 `.c` files?
Separate compilation

- We can compile each `.c` file *separately* to create an `.o` file.

- Then link all `.o` files together to create the executable.

```bash
gcc -c minmax.c -o minmax.o
gcc -c stats.c -o stats.o
gcc minmax.o stats.o -o minmax
```

- If we change `minmax.c`, we only need to recompile *that* file and relink.
  - Makefiles make this very easy.
Multiple implementations of a module

- The same `module.h` can be implemented in different ways

```c
// stats_alt.c - Εναλλακτική υλοποίηση του stats module

#include "stats.h"

// Επιστρέφει 1 αν value <= array[i] για κάθε i
int smaller_than_all(int value, int array[], int size) {
    for(int i = 0; i < size; i++)
        if(value > array[i])
            return 0;
    return 1;
}

int stats_find_min(int array[], int size) {
    for(int i = 0; i < size; i++)
        if(smaller_than_all(array[i], array, size))
            return array[i];
    return INT_MAX;  // εδώ φτάνουμε μόνο σε περίπτωση κενού array
}
Compiling with multiple implementations

• `minmax.c` is compiled **without knowing** how `stats.h` is implemented
  - this is **abstraction**!

• We can then link with **any** implementation we want

```
gcc -c minmax.c -o minmax.o

# use the first implementation
gcc -c stats.c -o stats.o
gcc minmax.o stats.o -o minmax

# OR the second
gcc -c stats_alt.c -o stats_alt.o
gcc minmax.o stats_alt.o -o minmax
```
Multiple implementations of a module

• All implementations should provide the same high-level behavior
  - So the program will work with any of them

• But one implementation might be more efficient than some other
  - This often depends on the specific application

• Which implementation of stats.h would you choose?
Makefiles

• Good programmers are lazy
  - they want to spend their time programming, not compiling
• Nobody likes typing the same gcc ... commands 100 times
• We can automate compilation with a Makefile
A simple Makefile

- This means: to create the file `minmax` run these commands

- To compile we run `make minmax`
  - or simply `make` to compile the first target in the Makefile
A simple Makefile - first problem

- We modify `minmax.c`, but make refuses to rebuild `minmax`

```
$ make minmax
make: 'minmax' is up to date.
```

- solution: dependencies

```
minmax: minmax.c stats.c
  gcc -c minmax.c -o minmax.o
  gcc -c stats.c -o stats.o
  gcc minmax.o stats.o -o minmax
```

- this means: `minmax` depends on `minmax.c, stats.c`
  - if any of these files is newer (last modification time) than `minmax` itself, the commands are run again!
A simple Makefile - second problem

- We modify `minmax.c`, but `make` recompiles **everything**

- Solution: **separate rules** for each file we create

```
minmax.o: minmax.c
    gcc -c minmax.c -o minmax.o

stats.o: stats.c
    gcc -c stats.c -o stats.o

minmax: minmax.o stats.o
    gcc minmax.o stats.o -o minmax
```

- To build `minmax` we need to build `minmax.o`, `stats.o`
  - `minmax.o` depends on `minmax.c` which is newer, so `make` recompiles
  - `stats.o` depends on `stats.c` which is older, so no need to recompile
Implicit rules

• *make* knows how to make *foo.o* if a file *foo.c* exists, by running:

```
gcc -c foo.c -o foo.o
```

• This is called an **implicit rule**

• So we don't need rules for *o* files!

```
minmax: minmax.o stats.o
    gcc minmax.o stats.o -o minmax
```
Variables

- We can use variables to further simplify the Makefile.
  - To create a variable: \texttt{VAR = ...}
  - To use a variable we write \texttt{$(VAR)$} anywhere in the Makefile.

- This allows to easily reuse the Makefile.

```makefile
# Άρχεια .o (αλλάζουμε απλά σε stats_alt.o για τη δεύτερη υλοποίηση!)
OBJS = minmax.o stats.o

# Το εκτελέσιμο πρόγραμμα
EXEC = minmax

$(EXEC): $(OBJs)
    gcc $(OBJs) -o $(EXEC)
```
CFLAGS variable

• A **special variable**

• Passed as arguments to the compiler when compiling a `.o` file using an implicit rule

• E.g. enable all warnings, treat them as errors, and allow debugging

  \[
  \text{CFLAGS} = \text{-Wall} \ \text{-Werror} \ \text{-g}
  \]
Auxiliary rules

- Then don't really create files but run useful commands
- E.g. we can use `make clean` to delete all files the compiler built

```bash
clean:
  rm -f $(OBJ) $(EXEC)
```

- And `make run` to compile and execute the program with predefined arguments

```bash
ARGS = arg1 arg2 arg3

run: $(EXEC)
  ./$(EXEC) $(ARGS)
```
Structuring a large project

- As projects grow, having all files in a single directory is not practical.
- E.g. we want the same module to be used by many programs.
- A simple structure:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>include</td>
<td>shared modules, used by multiple programs</td>
</tr>
<tr>
<td>modules</td>
<td>module implementations</td>
</tr>
<tr>
<td>programs</td>
<td>executable programs</td>
</tr>
<tr>
<td>tests</td>
<td>unit tests (we'll talk about these later)</td>
</tr>
<tr>
<td>lib</td>
<td>libraries (we'll talk about these later)</td>
</tr>
</tbody>
</table>
Putting the pieces together

# paths
MODULES = ../../modules
INCLUDE = ../../include

# Compile options. Το -I<dir> χρειάζεται για να βρει το gcc τα αρχεία
CFLAGS = -Wall -Werror -g -I$(INCLUDE)

# Αρχεία .o, εκτελέσιμο πρόγραμμα και παράμετροι
OBJS = minmax.o $(MODULES)/stats.o
EXEC = minmax
ARGS =

$(EXEC): $(OBJS)
    gcc $(OBJS) -o $(EXEC)

clean:
    rm -f $(OBJS) $(EXEC)

run: $(EXEC)
    ./$(EXEC) $(ARGS)
Editor use in programming

- Programs are plain text files
- Any editor can be used
- But using an editor **efficiently** is important
- It can make the difference between boring and creative programming
Editor types

• Old-school editors: *vim, emacs, …*
  - Fast, reliable, very configurable, available everywhere
  - Compiling/debugging is hard, needs tweaking

• IDEs: *Visual Studio, Eclipse, NetBeans, CLion, …*
  - Integrated compiler, debugger and many other tools
  - Too much “magic”, not ideal for learning

• Modern code-editors: *VS Code, Sublime Text, Atom, …*
  - Good balance between the two
  - Many options, a bit of tweaking is needed
VS Code

- Modern, open-source code editor, available for all major systems
- Made by Microsoft, but it's completely different than Visual Studio (an IDE)
- Will be used in lectures
  - lecture code is configured for use in VS Code
  - but you are free to use any other editor you want
- Installation instructions for all tools used in the class
Configuring VS Code

- `.vscode` dir provided in the lecture code
  - you can copy this directory in any of your projects
- You only need to modify `.vscode/settings.json`

```json
{
  "c_project": {
      // Directory στο οποίο βρίσκεται το πρόγραμμα
      "dir": "programs/minmax",
      // Όνομα του εκτελέσιμου προγράμματος
      "program": "minmax",
      // Ορίσματα του προγράμματος.
      "arg1": "-4",
      "arg2": "35",
      ...
  },
}
```
Compiling/Executing in VS Code

- **Menu Terminal / Run Task**

- **Make:** compile executes
  
  ```
  make <program>
  ```

  Errors are nicely displayed

- **Make:** compile and run executes
  
  ```
  make <program>
  ./<program> <arg1> <arg2> ...
  ```

- **Ctrl-Shift-B** executes the default task
Debugging in VS Code

• Set breakpoints (F9)

• F5 to start debugging

• We can examine/modify variables while execution is paused

• We can execute code step by step

• We can see where **segmentation faults** happen
A few useful VS Code features

- **Ctrl-P**: quickly open file
- **Ctrl-Shift-0**: find function
- **Ctrl-/:** toggle comment
- **Ctrl-Shift-F**: search/replace in all files
- **Ctrl-``**: move between code and terminal
- **F8**: go to next compilation error
- **Alt-up, Alt-down**: move line(s)
Git

• A system for tracking changes in source code
  - used by most major projects today

• Very useful when multiple developers collaborate in the same code
  - but also for single-developer projects

• We will use it for
  - lecture code
  - labs
  - projects

• We will store repositories in github.com, a popular Git hosting site
Git, main workflow

1. **clone** a repository, creating a local copy

2. Modify some files

3. **commit** changes to the local repository

4. **push** the changes to the remote repository

For multiple developers/machines:

5. **pull** changes from a different local repository copy
Git, getting started

- Install Git following the instructions
- Configure Git
  ```
  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"
  ```
- Create an account on github.com
- Create an empty (public or private) repository `test-repo` on github.com
  - Check “Initialize this repository with a README”
  - Its URL will be `https://github.com/<username>/test-repo`
Git, cloning a repository

```bash
git clone https://github.com/<username>/test-repo
```

- This will create a directory `test-repo` containing a local repository copy
- Check that `README.md` is present
- Try running `git status` inside `test-repo`
Git, committing changes

• Modify README.md

• Run `git status`
  - README.md appears as **modified**

• To commit the changes:

  ```
  git commit -a -m "Change README"
  ```

- `-a`: commit **all** modified files
- `-m "..."`: assign a message to the commit
Git, adding files

• Create a new file `foo.c`

• Run `git status`
  
  - `foo.c` appears as `untracked`

• To add it

  ```
  git add foo.c
  git commit -m "Add foo.c"
  ```

• Run `git status` again

  ```
  Your branch is ahead of 'origin/master' by 2 commits.
  ```
Git, pushing commits

- Visit (or clone) [https://github.com/<username>/test-repo](https://github.com/<username>/test-repo)
  - the local changes do not appear

- To push your local commits to the remote repository

  ```
git push
```
Git, pulling commits

- From a different local repository copy (e.g. a different machine)
  
git pull

- The remote changes are copied to the local repository

- Local changes should be committed before running this
  - They will be **merged** with the remote ones
.gitignore

- Files listed in the `.gitignore` special file are ignored by Git (blacklist)
- The inverse is often useful
  - save nothing except files in `.gitignore` (whitelist)

```plaintext
# Αγνοούμε όλα τα αρχεία (όχι τα directories)
*
!*/

# Εκτός από τα παρακάτω
!* .c
!* .h
!* .mk
!Makefile
! .gitignore
! README.md
! .vscode/.*.json
```
Readings

• T. A. Standish. Data Structures, Algorithms and Software Principles in C, Chapter 4

• Robert Sedgewick. Αλγόριθμοι σε C, Κεφ. 4

• make manual, Chapter 2

• A beginner's guide to Git

• VS Code introductory videos