

# Build tools

# Compilers

# Historical compilers

- Proprietary
  - Intel C++ Compiler (ICC, 1970's?)
  - Microsoft Visual C++ (MSVC, 1993)
  - ARM Compiler (ARMCC, 2005)
  - AMD Optimizing C/C++ Compiler (AOCC, 2017)
- Open source
  - GNU Compiler Collection (GCC, 1987)
  - LLVM (2003–)

# Evolution of compilers

- 2014: **ARM** Compiler rebased on **LLVM**
- 2017: **AMD** Compiler was always based on **LLVM**
- 2021: **Intel** C++ Compiler rebased on **LLVM**

# Current major compilers

- **Microsoft Visual C++**
  - default on MS Windows (in MS Visual Studio)
- **GCC**
  - default on most open source OSs
- **LLVM** (for C/C++: **Clang**)
  - base for hardware vendor compilers (**Intel**, **ARM**, **AMD**, **nVidia**)
  - default on MacOS, iOS (in Apple X Code)
  - default for native applications on Android

# Components of a compiler

- Front-end (parses and analyses code – language-specific)
- Intermediate representation (IR) (most code optimization happens here)
- Back-end (writes assembly or machine code – ISA-specific)
  
- LLVM frontends:
  - C and C++ (Clang), Fortran (Flang), Rust, Zig, Swift
- LLVM backends:
  - Intel/AMD/ARM compilers, nVidia CUDA compiler, AMD ROCm

# LLVM IR

```
define dso_local noundef i32 @square(int)(i32 noundef %num) #0 !dbg !10 {
entry:
  %num.addr = alloca i32, align 4
  store i32 %num, ptr %num.addr, align 4
  call void @llvm.dbg.declare(metadata ptr %num.addr, metadata !16, metadata !DIExpression()), !dbg !17
  %0 = load i32, ptr %num.addr, align 4, !dbg !18
  %1 = load i32, ptr %num.addr, align 4, !dbg !19
  %mul = mul nsw i32 %0, %1, !dbg !20
  ret i32 %mul, !dbg !21
}

declare void @llvm.dbg.declare(metadata, metadata, metadata) #1

attributes #0 = { mustprogress noline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true"
  "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { nocallback noreturn nosync nounwind speculatable willreturn memory(none) }
```

# Compiling

# Compiler invocation (1)

- Use `man gcc` / `man clang` for help.
- Compile and link:

```
gcc -o executable source_code.c
```

- Compile only:

```
gcc -c -o file.o file.c
```

- Link only

```
gcc -o executable file0.o file1.o file2.o file3.o
```

- Write assembly

```
gcc -S assembly.S source_code.c
```

- Internally, `gcc` runs other tools (assembler: `as`, linker: `ld`)

## Compiler invocation (2)

- Enable warnings:

```
gcc -Wall -c -o file.o file.c
```

- Enable optimization:

```
gcc -Wall -O3 -c -o file.o file.c
```

# Note for MacOS

Install binutils:

- from Homebrew <https://brew.sh/>

```
brew install binutils
```

- or from MacPorts <https://www.macports.org>

```
port install binutils
```

Utilities may be prefixed by a g:

objdump → gobjdump

# Tools

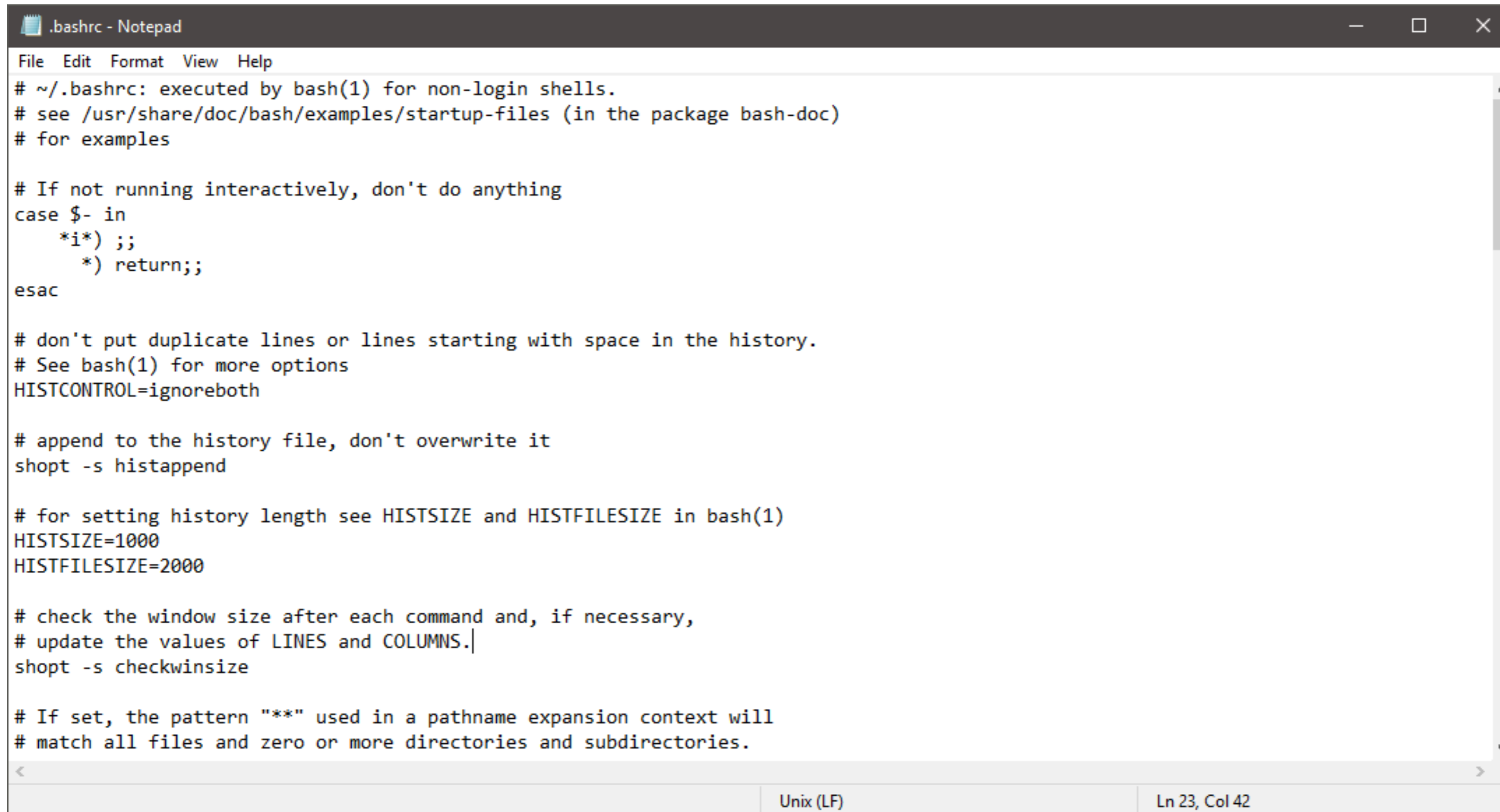
- `hexdump` dump hexadecimal representation of any file
  - `hexdump -C` also print ASCII for valid ASCII bytes
  - `hexdump -C | less` “pipe” output to pager
  - `hexdump -C > file.hex` write output to a file
- `objdump` dump contents of object file
  - `objdump -p` print object information (dynamically-linked libraries, symbols)
  - `objdump -M intel -d` disassembles object file, prints assembly code
- or online: <http://godbolt.org>

# Editing code

# Applications for writing code

- Text editors
- Code editors
- Integrated development environment (IDE)

# Text editor: Notepad



```
.bashrc - Notepad
File Edit Format View Help
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
  *i*) ;;
  *) return;;
esac

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
shopt -s checkwinsize

# If set, the pattern "*" used in a pathname expansion context will
# match all files and zero or more directories and subdirectories.
```

Unix (LF) Ln 23, Col 42

# Code editor: emacs

```
File Edit Options Buffers Tools Operate Mark Regexp Immediate Subdir Help
57 (global-set-key (kbd "C-c a") 'screenwriter-action-block)
58 (global-set-key (kbd "C-c d") 'screenwriter-dialog-block)
59 (global-set-key (kbd "C-c t") 'screenwriter-transition)
60 (setq auto-mode-alist (cons ('("\\.scp" . screenplay-mode) auto-mode-alist)
61 (setq auto-mode-alist (cons ('("\\.md" . markdown-mode) auto-mode-alist)
62
63 ;; w3m setup
64 (setq browse-url-browser-function 'w3m-browse-url)
65 (autoload 'w3m-browse-url "w3m" "Ask a WWW browser to show a URL." t)
66 (global-set-key "\C-xm" 'browse-url-at-point)
67 (setq w3m-use-cookies t)
68
69 ;; auto-complete
70 ;; install by running emacs and doing an m-x load-file.el
71 ;; load ~/.emacs.d/auto-complete/etc/install.el
--:--- .emacs 21% L68 (Emacs-Lisp AC Abbrev)
8 ** <2021-09-18 1300-1600>
9 * Grocery
10 :CATEGORY: Food
11 ** TODO Artichokes
12 ** TODO Bagels
13 - Flour
14 - Baking soda
15 - Rock salt
16 ** Pretzels
17
18
--:**- List.org Bot L12 (Org U:%%- a-compat32 2% L5
```

# Code editor: vi / vim / neovim

The screenshot shows a Neovim code editor interface. The top status bar displays system information: 13/7 11.45 PM, 41% battery, 7% CPU usage, 9.5 GB memory, 1.0 kB download, and 21 kB upload. The editor is open to a file named `h/s/main.rs`. The left sidebar shows a file tree with the following structure:

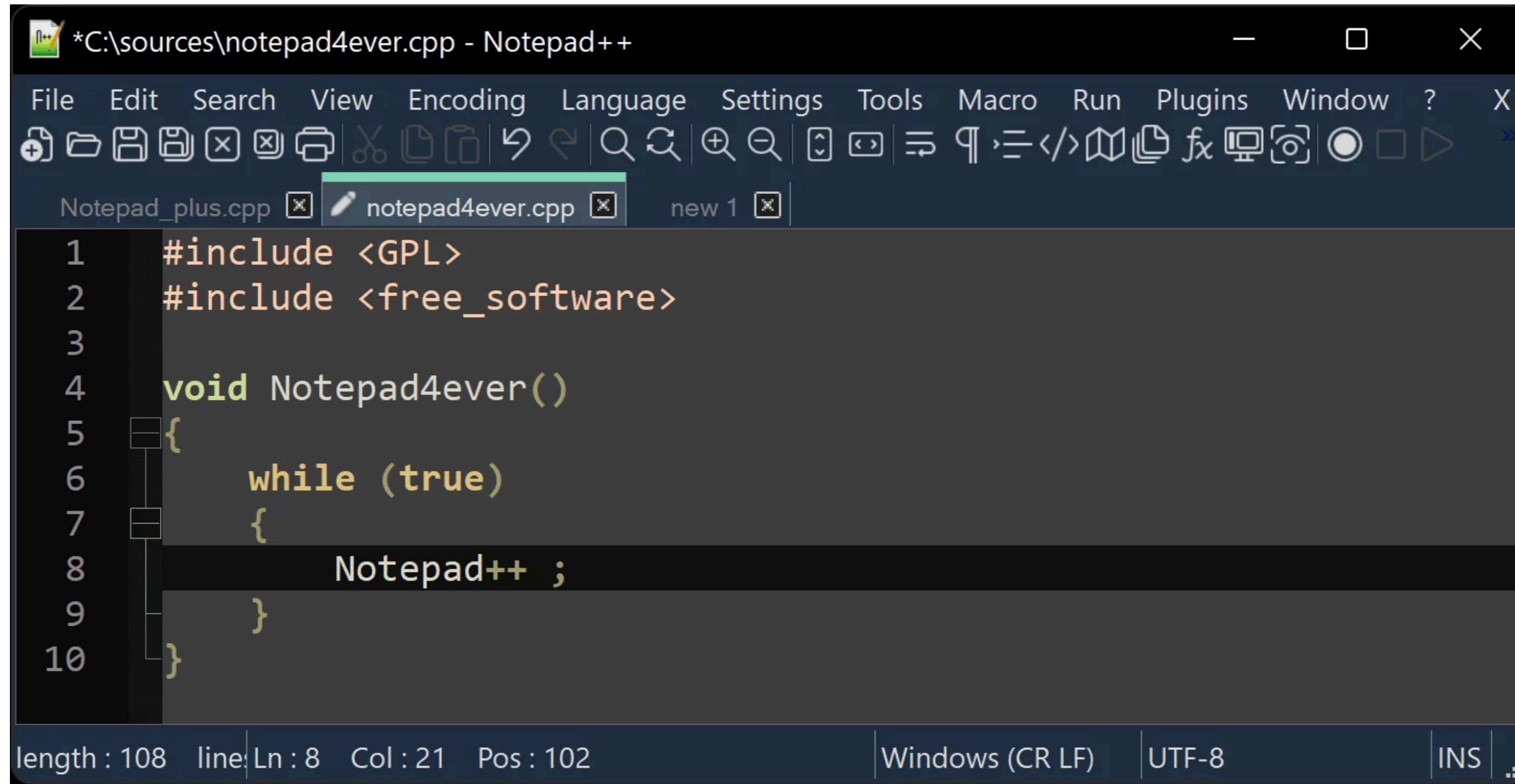
- h/s/main.rs+ (selected)
- h/Cargo.toml
- h/.gitignore

The main editor area displays the following Rust code:

```
9 use rand::Rng;
8 use std::cmp::Ordering;
7 use std::io;
6
5 fn main() {
4     let num = rand::thread_rng().gen_range(1, 101);
3     loop {
2         let mut guess = String::new();
1         io::stdin().read_line(&mut guess).expect("error");
x 10        let example = std::io::std rustc: cannot find value `std` in module `std::io` not found in `st
1
2         let guess: usize = mat stderr Function [LC] pub fn stderr() -> Stderr
3             Ok(num) => num, stdout Function [LC] pub fn stdout() -> Stdout
4             Err(_) => continue,
5         };
6
7         match guess.cmp(&num) {
8             Ordering::Less => println!("Too small!"),
9             Ordering::Greater => println!("Too big!"),
10            Ordering::Equal => {
11                println!("You win!");
12                break;
13            }
14        }
15    }
16 }
```

The error message on line 10 indicates a compilation error: `rustc: cannot find value `std` in module `std::io` not found in `st``. The status bar at the bottom shows the editor is in `INSERT` mode, with a completion popup for `std` showing options like `stdin`, `stderr`, and `stdout`. The status bar also displays the current file path, mode, and error information: `E:2(L10)E:2(L9)`.

# Code editor: Notepad++

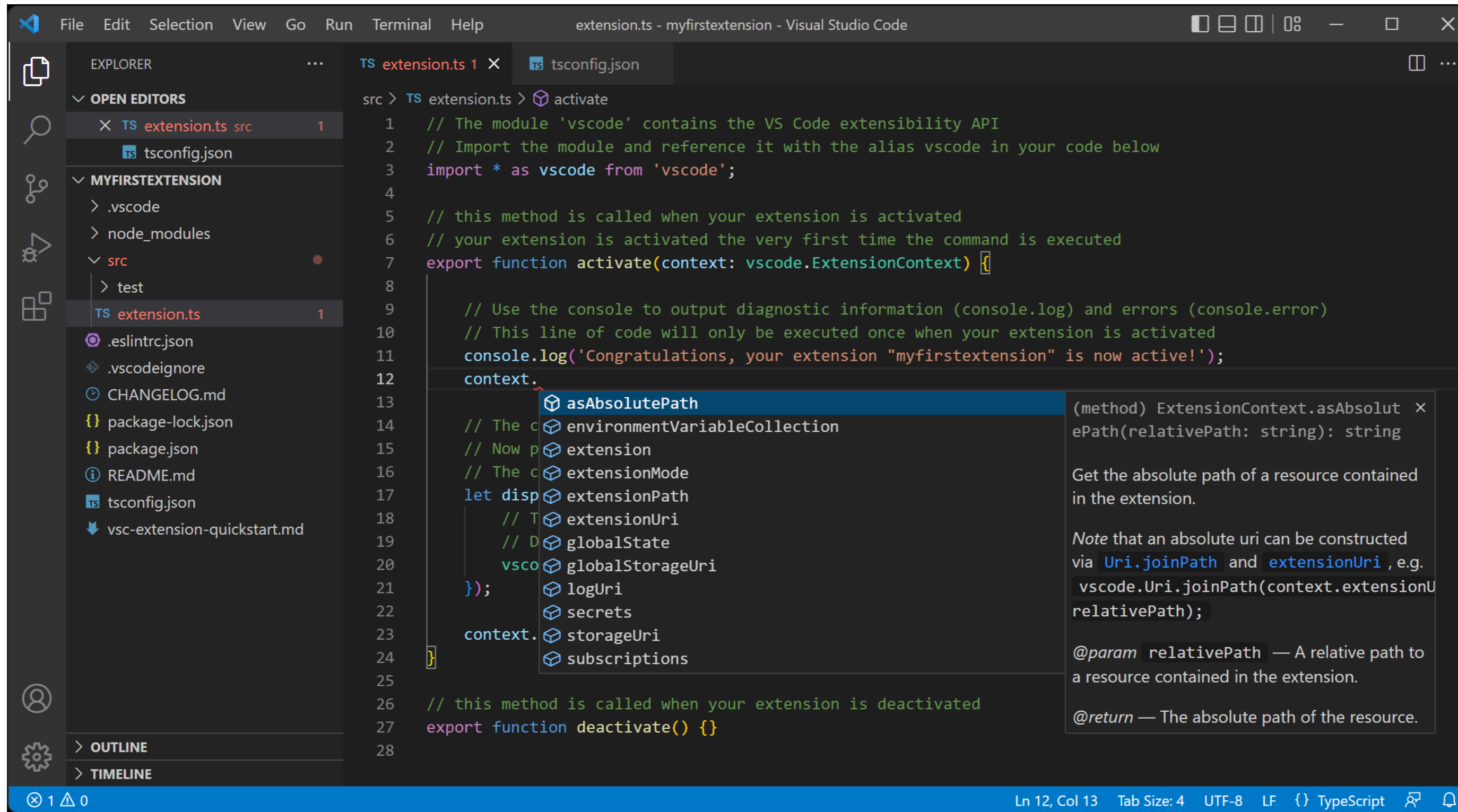


The image shows a screenshot of the Notepad++ code editor. The window title is "\*C:\sources\notepad4ever.cpp - Notepad++". The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, Plugins, Window, and Help. The toolbar contains various icons for file operations, editing, and development. The editor has three tabs: "Notepad\_plus.cpp", "notepad4ever.cpp" (which is active), and "new 1". The code in the editor is as follows:

```
1  #include <GPL>
2  #include <free_software>
3
4  void Notepad4ever()
5  {
6      while (true)
7      {
8          Notepad++ ;
9      }
10 }
```

The status bar at the bottom shows "length : 108", "lines: Ln : 8", "Col : 21", "Pos : 102", "Windows (CR LF)", "UTF-8", and "INS".

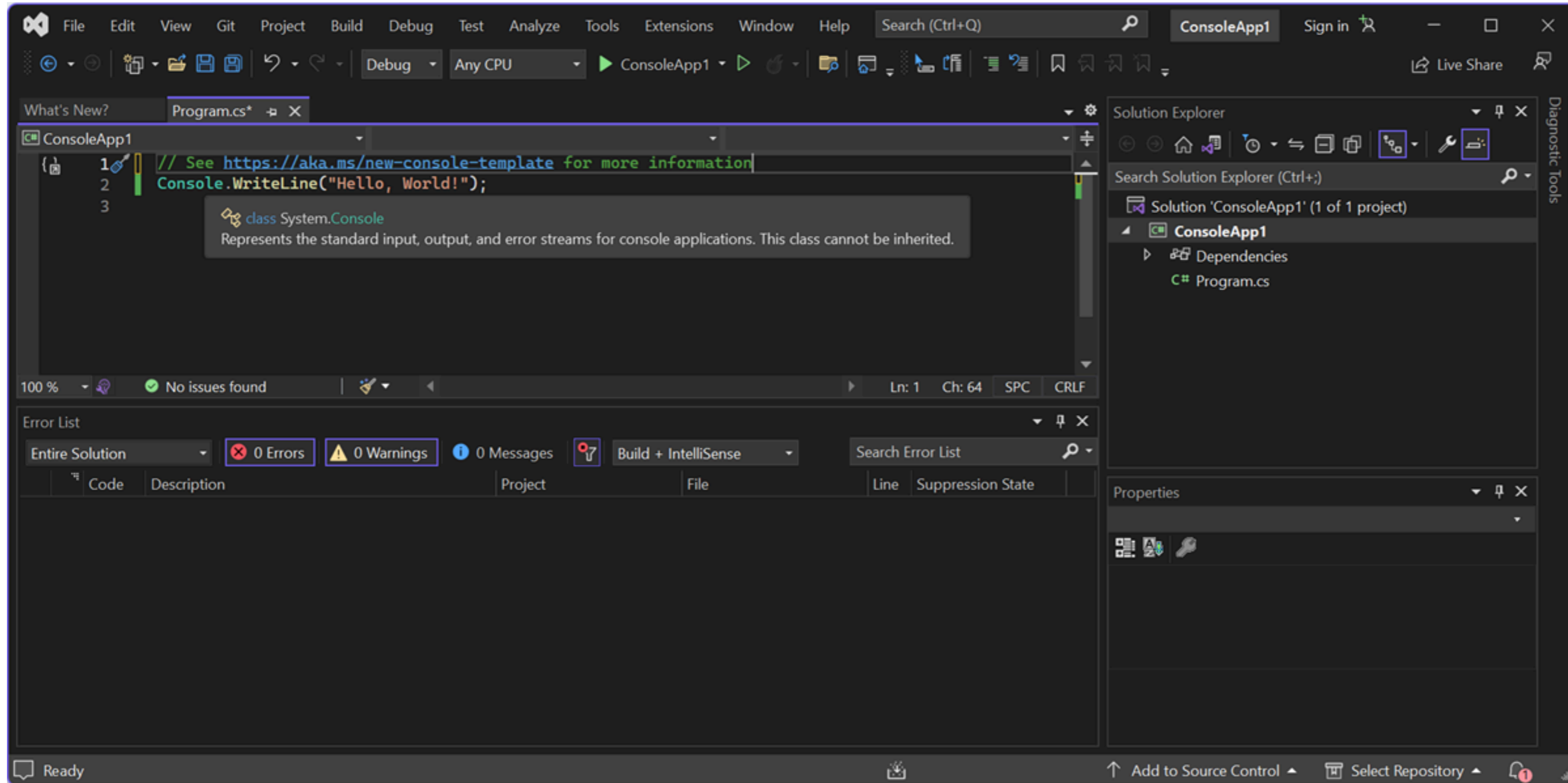
# Code editor: Visual Studio Code



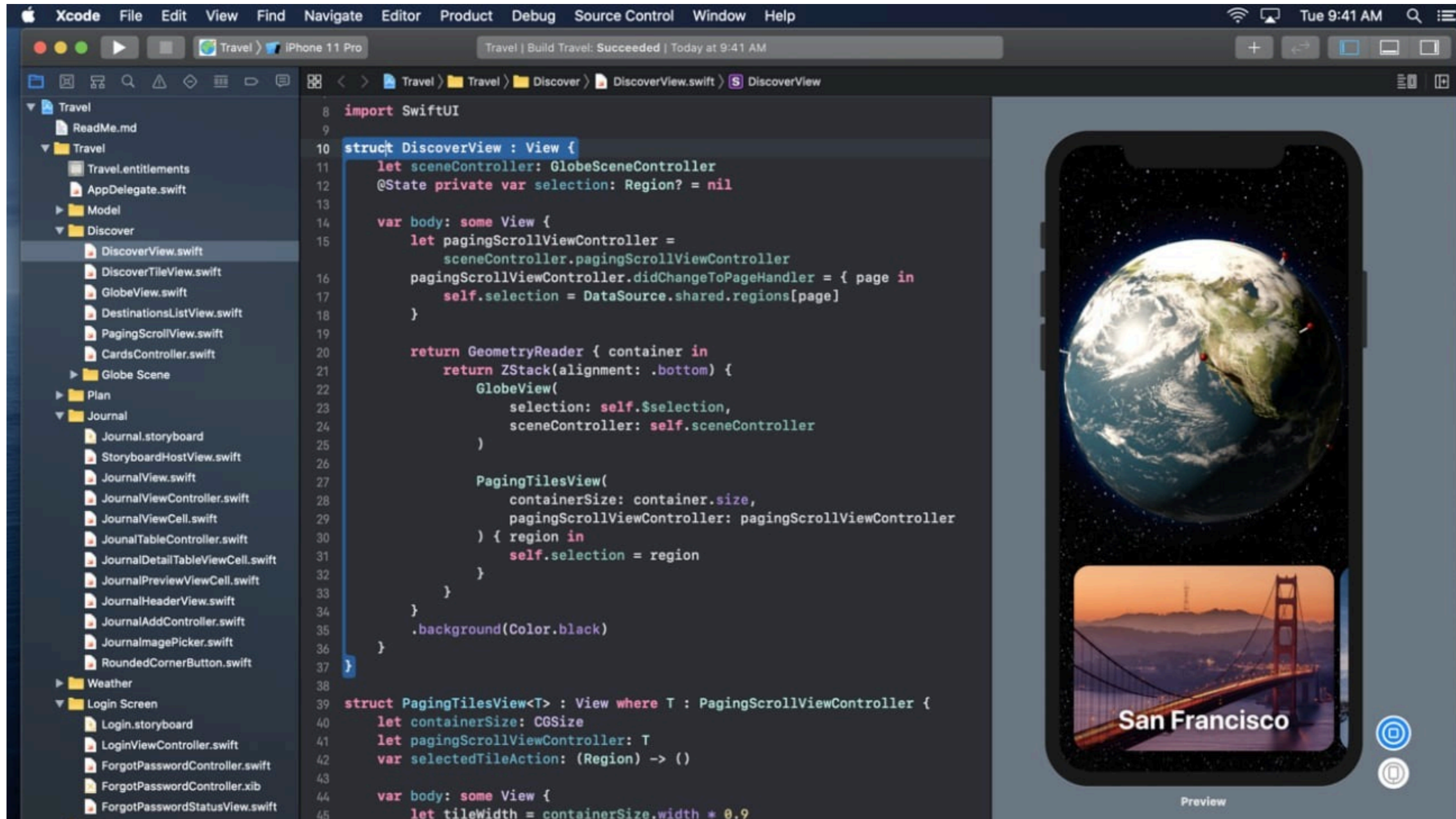
## More code editors

- gedit
- Kate
- Sublime Text (paid)
- many more...

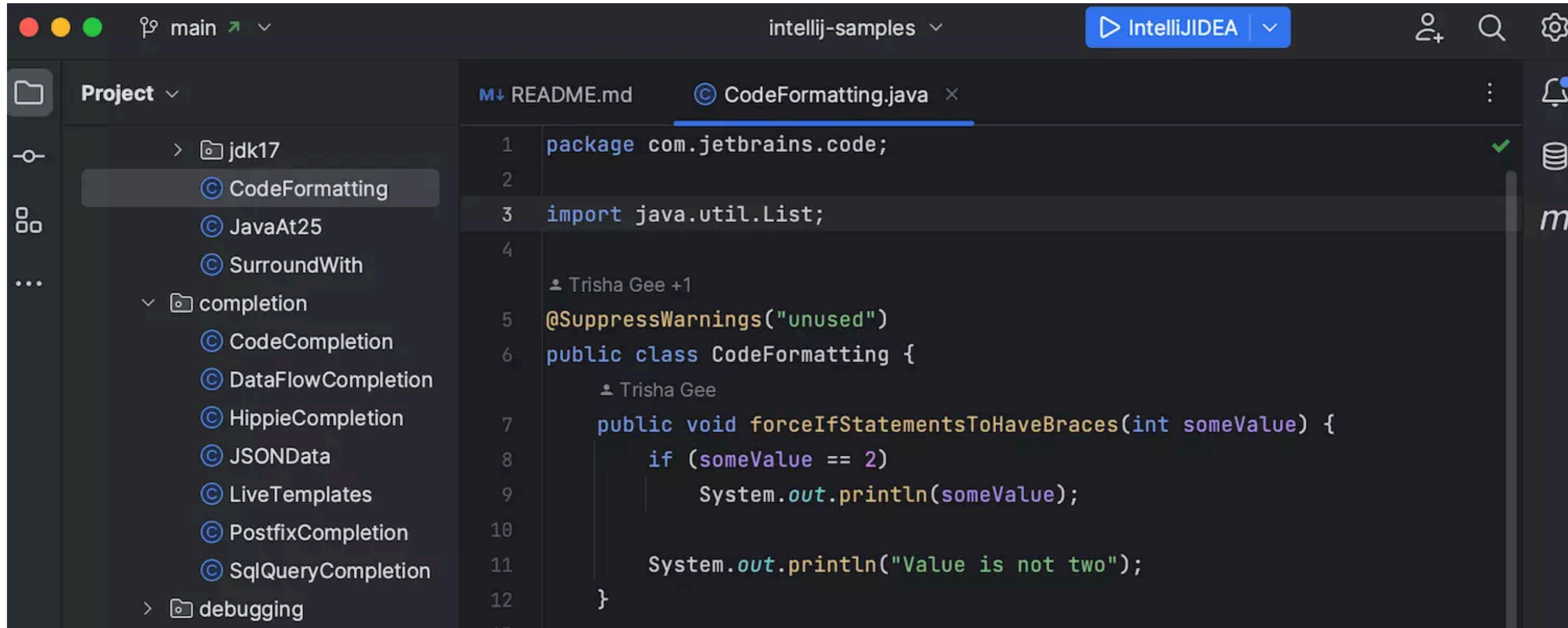
# IDE: Microsoft Visual Studio (paid)



# IDE: Apple Xcode



# IDE: IntelliJ IDEA (paid)



## More IDEs

- PyCharm (Python, paid)
- Android Studio
- KDevelop
- QtCreator
- Dev-C++
- Spyder (Python)
- ...

# Code editor vs. IDE

## IDE pros:

- one-click compile
- IDE aware of whole project
  - can suggest code completions from different files
- integrated tools (e.g. debugger)

## IDE cons:

- Project setup takes time and effort
- “Walled garden” problem
  - By default, anyone who wants to compile your project needs the same IDE.

# Build systems

# How do we compile a complex project?

- Option 1:

```
gcc -Wall -O3 -c -o ggml.o ggml.c
gcc -Wall -O3 -c -o ggml-alloc.o ggml-alloc.c
g++ -Wall -O3 -c -o llama.o llama.cpp
g++ -Wall -O3 -c -o common.o common/common.c
g++ -Wall -O3 -c -o console.o common/console.c
g++ -Wall -O3 -c -o grammar-parser.o common/grammar-parser.c
g++ -Wall -O3 -shared -fPIC -o libllama.so ggml.o ggml-alloc.o llama.o \
    common.o console.o grammar-parser.o
```

- Option 2

- Put above commands in a “shell script” file, e.g. `compile.sh`
- Run:

```
./compile.sh
```

- Problems:

- Difficult to modify (e.g. change compiler options)
- We recompile everything everytime

# Build automation

- IDE integrated:
  - Visual Studio
  - Xcode
- Stand-alone:
  - make
  - Bazel (based on Google's internal tool Blaze) / Buck (Facebook)
  - Ninja (Google, for Chrome)
  - CMake (uses make, Ninja,...), qmake (uses make), Meson (uses Ninja, ...)

# Make

- Create a file named Makefile:

```
ggml.o: ggml.c ggml.h ggml-cuda.h
      gcc -Wall -O3 -c -o ggml.o ggml.c

ggml-alloc.o: ggml-alloc.c ggml.h ggml-alloc.h
      gcc -Wall -O3 -c -o ggml-alloc.o ggml-alloc.c

llama.o: llama.cpp ggml.h ggml-alloc.h ggml-cuda.h ggml-metal.h llama.h
      g++ -Wall -O3 -c -o llama.o llama.cpp

common.o: common/common.cpp common/common.h build-info.h common/log.h
      g++ -Wall -O3 -c -o common.o common/common.cpp

console.o: common/console.cpp common/console.h
      g++ -Wall -O3 -c -o console.o common/console.cpp

grammar-parser.o: common/grammar-parser.cpp common/grammar-parser.h
      g++ -Wall -O3 -c -o grammar-parser.o common/grammar-parser.cpp

libllama.so: ggml.o ggml-alloc.o llama.o common.o console.o grammar-parser.o
      g++ -Wall -O3 -shared -fPIC -o libllama.so ggml.o ggml-alloc.o llama.o \
      common.o console.o grammar-parser.o
```

- Run

```
make libllama.so
```

# Make rule syntax

```
target: source0 source1 source2 ...  
      recipe
```

Whenever one of the sources was modified after the target, run the recipe (to rebuild the target).

Otherwise, consider target up-to-date and do nothing.

**Beware:** recipe must be offset to the right using a TAB character!

# Make variables

We can create variables in a Makefile with

```
VARIABLE_NAME := string content of the variable
```

Then, `$(VARIABLE_NAME)` can be used,  
and will be expanded into `string content of the variable`.

Example:

```
contents_of_my_variable.txt:  
    echo $(VARIABLE_NAME) > contents_of_my_variable.txt
```

```
ggml.o: ggml.c ggml.h ggml-cuda.h
gcc -Wall -O3 -c -o ggml.o ggml.c

ggml-alloc.o: ggml-alloc.c ggml.h ggml-alloc.h
gcc -Wall -O3 -c -o ggml-alloc.o ggml-alloc.c

llama.o: llama.cpp ggml.h ggml-alloc.h ggml-cuda.h ggml-metal.h llama.h
g++ -Wall -O3 -c -o llama.o llama.cpp

common.o: common/common.cpp common/common.h build-info.h common/log.h
g++ -Wall -O3 -c -o common.o common/common.cpp

console.o: common/console.cpp common/console.h
g++ -Wall -O3 -c -o console.o common/console.cpp

grammar-parser.o: common/grammar-parser.cpp common/grammar-parser.h
g++ -Wall -O3 -c -o grammar-parser.o common/grammar-parser.cpp

libllama.so: ggml.o ggml-alloc.o llama.o common.o console.o grammar-parser.o
g++ -Wall -O3 -shared -fPIC -o libllama.so ggml.o ggml-alloc.o llama.o \
common.o console.o grammar-parser.o
```

```
CC := gcc
CXX := g++
CFLAGSS := -Wall -O3
CXXFLAGS := -Wall -O3

ggml.o: ggml.c ggml.h ggml-cuda.h
$(CC) $(CFLAGS) -c -o ggml.o ggml.c

ggml-alloc.o: ggml-alloc.c ggml.h ggml-alloc.h
$(CC) $(CFLAGS) -c -o ggml-alloc.o ggml-alloc.c

llama.o: llama.cpp ggml.h ggml-alloc.h ggml-cuda.h ggml-metal.h llama.h
$(CXX) $(CXXFLAGS) -c -o llama.o llama.cpp

common.o: common/common.cpp common/common.h build-info.h common/log.h
$(CXX) $(CXXFLAGS) -c -o common.o common/common.cpp

console.o: common/console.cpp common/console.h
$(CXX) $(CXXFLAGS) -c -o console.o common/console.cpp

grammar-parser.o: common/grammar-parser.cpp common/grammar-parser.h
$(CXX) $(CXXFLAGS) -c -o grammar-parser.o common/grammar-parser.cpp

libllama.so: ggml.o ggml-alloc.o llama.o common.o console.o grammar-parser.o
$(CXX) $(CXXFLAGS) -shared -fPIC -o libllama.so ggml.o ggml-alloc.o llama.o \
common.o console.o grammar-parser.o
```

## Special make variables

- $\$(@)$  the target of the current rule
- $\$( < )$  the first source of the current rule
- $\$( ^ )$  all the sources of the current rule

```
CC := gcc
CXX := g++
CFLAGSS := -Wall -O3
CXXFLAGS := -Wall -O3

ggml.o: ggml.c ggml.h ggml-cuda.h
$(CC) $(CFLAGS) -c -o ggml.o ggml.c

ggml-alloc.o: ggml-alloc.c ggml.h ggml-alloc.h
$(CC) $(CFLAGS) -c -o ggml-alloc.o ggml-alloc.c

llama.o: llama.cpp ggml.h ggml-alloc.h ggml-cuda.h ggml-metal.h llama.h
$(CXX) $(CXXFLAGS) -c -o llama.o llama.cpp

common.o: common/common.cpp common/common.h build-info.h common/log.h
$(CXX) $(CXXFLAGS) -c -o common.o common/common.cpp

console.o: common/console.cpp common/console.h
$(CXX) $(CXXFLAGS) -c -o console.o common/console.cpp

grammar-parser.o: common/grammar-parser.cpp common/grammar-parser.h
$(CXX) $(CXXFLAGS) -c -o grammar-parser.o common/grammar-parser.cpp

libllama.so: ggml.o ggml-alloc.o llama.o common.o console.o grammar-parser.o
$(CXX) $(CXXFLAGS) -shared -fPIC -o libllama.so ggml.o ggml-alloc.o llama.o \
common.o console.o grammar-parser.o
```

```
CC := gcc
CXX := g++
CFLAGSS := -Wall -O3
CXXFLAGS := -Wall -O3

ggml.o: ggml.c ggml.h ggml-cuda.h
$(CC) $(CFLAGS) -c -o $@ $(<)

ggml-alloc.o: ggml-alloc.c ggml.h ggml-alloc.h
$(CC) $(CFLAGS) -c -o $@ $(<)

llama.o: llama.cpp ggml.h ggml-alloc.h ggml-cuda.h ggml-metal.h llama.h
$(CXX) $(CXXFLAGS) -c -o $@ $(<)

common.o: common/common.cpp common/common.h build-info.h common/log.h
$(CXX) $(CXXFLAGS) -c -o $@ $(<)

console.o: common/console.cpp common/console.h
$(CXX) $(CXXFLAGS) -c -o $@ $(<)

grammar-parser.o: common/grammar-parser.cpp common/grammar-parser.h
$(CXX) $(CXXFLAGS) -c -o $@ $(<)

libllama.so: ggml.o ggml-alloc.o llama.o common.o console.o grammar-parser.o
$(CXX) $(CXXFLAGS) -shared -fPIC -o $@ $(^)
```

# Static pattern rules

- Static pattern syntax:

```
target0 target1 target2 ... : target-pattern : source-pattern  
recipe
```

- Target pattern contains %, which will match anything
- Source pattern also contains %, which is replaced by the match in target
- Example:

```
some_file.o other_file.o third_file.o : %.o : %.c  
recipe
```

is equivalent to:

```
some_file.o: some_file.c  
recipe  
  
other_file.o: other_file.c  
recipe  
  
third_file.o: third_file.c  
recipe
```

```
ggml.o: ggml.c ggml.h ggml-cuda.h  
$(CC) $(CFLAGS) -c -o $(@) $(<)  
  
ggml-alloc.o: ggml-alloc.c ggml.h ggml-alloc.h  
$(CC) $(CFLAGS) -c -o $(@) $(<)
```

becomes

```
ggml.o ggml-alloc.o: %.o: %.c %.h  
$(CC) $(CFLAGS) -c -o $(@) $(<)  
  
ggml.o: ggml-cuda.h # Additional sources  
ggml-alloc.o: ggml.h # Additional sources
```

```
CC := gcc
CXX := g++
CFLAGSS := -Wall -O3
CXXFLAGS := -Wall -O3

ggml.o: ggml.c ggml.h ggml-cuda.h
$(CC) $(CFLAGS) -c -o $@ $(<)

ggml-alloc.o: ggml-alloc.c ggml.h ggml-alloc.h
$(CC) $(CFLAGS) -c -o $@ $(<)

llama.o: llama.cpp ggml.h ggml-alloc.h ggml-cuda.h ggml-metal.h llama.h
$(CXX) $(CXXFLAGS) -c -o $@ $(<)

common.o: common/common.cpp common/common.h build-info.h common/log.h
$(CXX) $(CXXFLAGS) -c -o $@ $(<)

console.o: common/console.cpp common/console.h
$(CXX) $(CXXFLAGS) -c -o $@ $(<)

grammar-parser.o: common/grammar-parser.cpp common/grammar-parser.h
$(CXX) $(CXXFLAGS) -c -o $@ $(<)

libllama.so: ggml.o ggml-alloc.o llama.o common.o console.o grammar-parser.o
$(CXX) $(CXXFLAGS) -shared -fPIC -o $@ $(^)
```

```
CC := gcc
CXX := g++
CFLAGSS := -Wall -O3
CXXFLAGS := -Wall -O3

ggml.o ggml-alloc.o: %.o: %.c %.h
    $(CC) $(CFLAGS) -c -o $(@) $(<)

ggml.o: ggml-cuda.h # Additional sources
ggml-alloc.o: ggml.h # Additional sources

llama.o: llama.cpp ggml.h ggml-alloc.h ggml-cuda.h ggml-metal.h llama.h
    $(CXX) $(CXXFLAGS) -c -o $(@) $(<)

common.o console.o grammar-parser.o: %.o: common/%.cpp common/%.h
    $(CXX) $(CXXFLAGS) -c -o $(@) $(<)

common.o: build-info.h common/log.h # Additional sources

libllama.so: ggml.o ggml-alloc.o llama.o common.o console.o grammar-parser.o
    $(CXX) $(CXXFLAGS) -shared -fPIC -o $(@) $(^)
```

```
CC := gcc
CXX := g++
CFLAGSS := -Wall -O3
CXXFLAGS := -Wall -O3

COBJS := ggml.o ggml-alloc.o
CXXOBS_LLAMA := llama.o
CXXOBS_COMMON := common.o console.o grammar-parser.o
CXXOBS := $(CXXOBS_LLAMA) $(CXXOBS_COMMON)

# Build rules
$(COBJS): %.o: %.c %.h
    $(CC) $(CFLAGS) -c -o $@ $(<)

$(CXXOBS_LLAMA): %.o: %.cpp %.h
    $(CXX) $(CXXFLAGS) -c -o $@ $(<)

$(CXXOBS_COMMON): %.o: common/%.cpp common/%.h
    $(CXX) $(CXXFLAGS) -c -o $@ $(<)

libllama.so: $(COBJS) $(CXXOBS)
    $(CXX) $(CXXFLAGS) -shared -fPIC -o $@ $(^)

# Additional sources
ggml.o: ggml-cuda.h
ggml-alloc.o: ggml.h
llama.o: llama.cpp ggml.h ggml-alloc.h ggml-cuda.h ggml-metal.h
common.o: build-info.h common/log.h
```

# Phony and default targets

- A “phony” target does not necessarily correspond to a file name:

```
.PHONY: clean  
clean:  
    rm libllama.so
```

- If no target is provided to the make command, the default target is the first one. A common pattern is:

```
.PHONY: default  
default: libllama.so
```

```
CC := gcc
CXX := g++
CFLAGSS := -Wall -O3
CXXFLAGS := -Wall -O3

COBJS := ggml.o ggml-alloc.o
CXXOBS_LLAMA := llama.o
CXXOBS_COMMON := common.o console.o grammar-parser.o
CXXOBS := $(CXXOBS_LLAMA) $(CXXOBS_COMMON)
LIBTARGET := libllama.so

.PHONY: default clean

# Build rules
default: $(LIBTARGET)

clean:
    rm -f $(COBJS) $(CXXOBS) $(LIBTARGET)

$(COBJS): %.o: %.c %.h
    $(CC) $(CFLAGS) -c -o $(@) $(<)

$(CXXOBS_LLAMA): %.o: %.cpp %.h
    $(CXX) $(CXXFLAGS) -c -o $(@) $(<)

$(CXXOBS_COMMON): %.o: common/%.cpp common/%.h
    $(CXX) $(CXXFLAGS) -c -o $(@) $(<)

$(LIBTARGET): $(COBJS) $(CXXOBS)
    $(CXX) $(CXXFLAGS) -shared -fPIC -o $(@) $(^)
```

```
# Additional sources
ggml.o: ggml-cuda.h
ggml-alloc.o: ggml.h
llama.o: llama.cpp ggml.h ggml-alloc.h ggml-cuda.h ggml-metal.h
common.o: build-info.h common/log.h
```



# Using shell commands

- The syntax is:

```
$(shell any-shell-command)
```

- For example:

```
TODAY := $(shell date)  
C_FILES := $(shell ls *.c)
```

# String replacement in variables

- The syntax is:

```
$(variable:pattern=replacement)
```

- The pattern contains %, which will match any substring
- The replacement may contain %, which will be replaced by the matched substring
- For example:

```
C_FILES := $(shell ls *.c)  
O_FILES := $(C_FILES:%.c=%.o)
```

# For more about make

```
# Using make  
man make
```

```
# Writing Makefiles  
info make
```