

```
vector<intT> f
ntT> res;
tT(2)>=i) {if
= 5 - int(2
es.push_back(nu
raIT; int main
egraIT n; while
umber: "; cin >
= v.begin(); i
()) cout << ",
; } return 0; }
t(int N) { if (
urn 1; long do
; i++) { result
n() { int N; co
ial " << N << "
("pause"); retu
; i++) cout <<
int *num = new
; for(
; i++) num[i] =
% (last-first+1
e(a[i]<mid) i++
while(a
a[i],a[j]); i++
num[j]; j--;
```

پیک

سی بعلاوه

```
std::cout
```

```
<< مراحل کامپایل
```

```
<< پروژه‌های چند فایل
```

```
<< Make
```

```
<< CMake
```

```
<< std::endl;
```

مراحل کامپایل

1

Preprocessor

خروجی این فاز، همان کد ورودی است که روی آن چند پیش‌پردازش انجام شده است. از جمله پیش‌پردازش‌ها، `expand` کردن ماکروها، دیفاین‌ها و اینکلودها و حذف کامنت‌ها است.

برای مثال، اگر یک `file.hpp` داشته باشیم و آن را در `file.cpp` اینکلود کنیم، می‌بینیم که در `file.i` کل محتوای `file.hpp` به جای دستور `#include` آمده است.

فلگ `-E`، خروجی این فاز را می‌دهد:

```
g++ -E file.cpp -o file.i
```

Compiler

خروجی این فاز، فایل اسمبلی است که کد تبدیل شده به دستورات CPU مربوطه است.

فلگ `-S`، خروجی این فاز را می‌دهد:

```
g++ -S file.cpp -o file.s
```

Assembler

خروجی این فاز، فایل باینری است که کد اسمبلی تبدیل شده به دستورات ماشین باینری است.

فلگ `-c`، خروجی این فاز را می‌دهد:

```
g++ -c file.cpp -o file.o
```

3

Linker

خروجی این فاز، برنامه قابل اجرا است.

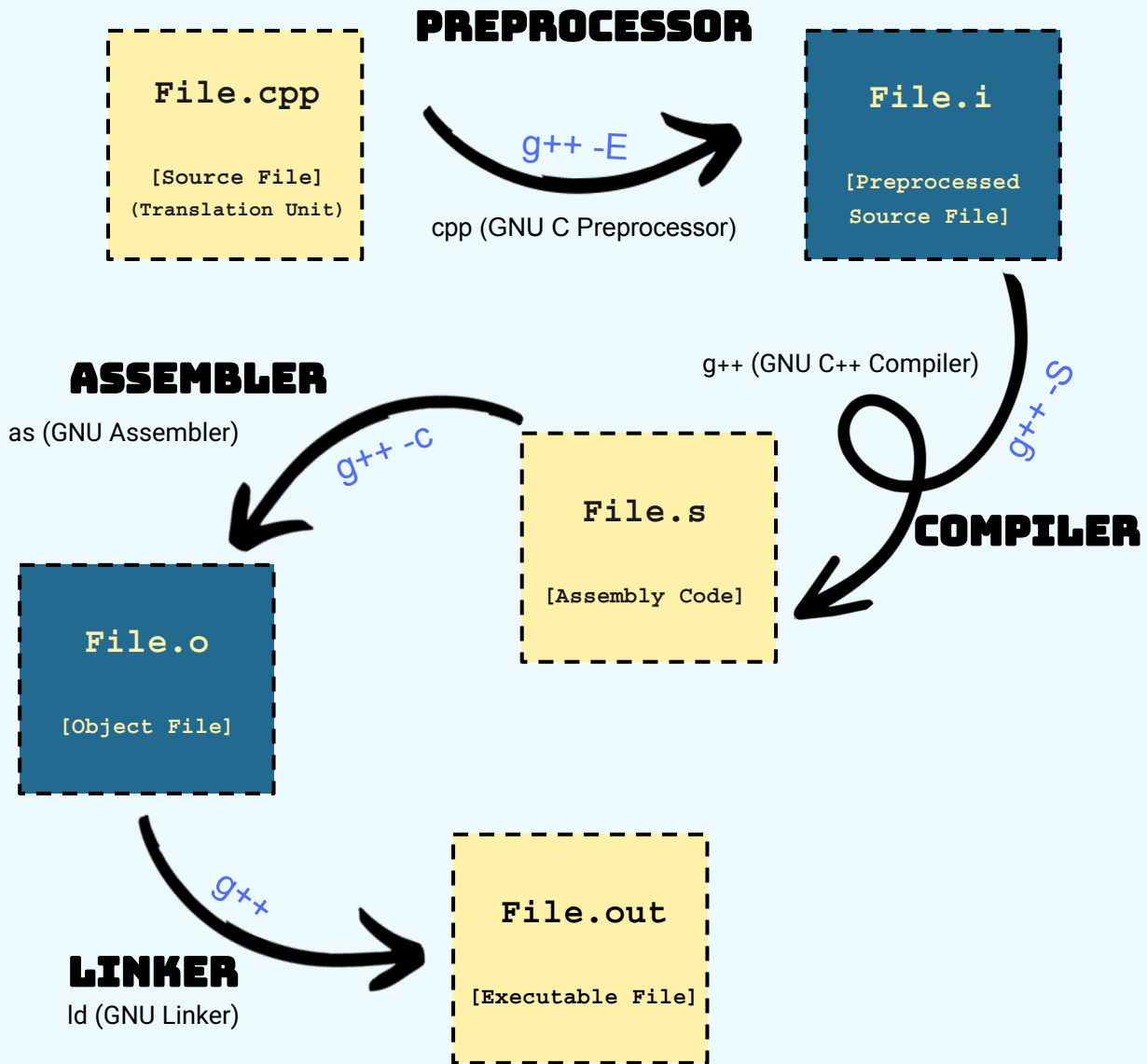
در این فاز، چندین `object file` یعنی فایل `.o` با هم ترکیب می‌شوند و یک فایل خروجی تولید می‌شود. (لایبرری‌ها [فایل `.a`] هم در همین فاز به کد وصل می‌شوند)

یعنی اگر چند فایل `file1.cpp` و `file2.cpp` داشته باشیم، هر کدام از اینها به طور مستقل به فایل `.o` تبدیل شده و در نهایت به هم وصل می‌شوند:

```
g++ file1.o file2.o -o file.out
```

احتمالاً تا الان یک آشنایی اولیه با `Makefile` دارید. در این مطلب می‌خواهیم یک مرور کلی از پروسه کامپایل کردن و پروژه‌های چند فایل با استفاده از `Makefile` داشته باشیم و به چند نکته مهم در آن اشاره کنیم. در نهایت هم ابزار `CMake` را به طور مختصر معرفی می‌کنیم.

`Makefile` یک ابزار برای تولید فایل قابل اجرا برنامه از سورس آن است. این ابزار برای `dependency management` استفاده می‌شود. برای درک بهتر این مطلب، خوب است که مروری بر نحوه کامپایل شدن یک برنامه سی یا سی++ داشته باشیم:



وقتی می‌زنیم: `g++ file.cpp -o file.out`، همه این مراحل به صورت یکجا طی می‌شوند!

وقتی با پروژه‌ای چند فایل کار داریم، به جای اینکه با هر تغییر، کل این مراحل برای همه فایل‌ها طی شوند (که باعث افزایش چشم‌گیر زمان کامپایل می‌شود)، هر کدام از فایل‌های سورس (که به آنها translation unit هم گفته می‌شود) را در ابتدا تبدیل به object file می‌کنیم (فایل .o)، و در نهایت آنها را لینک می‌کنیم.

حال وقتی که یک فایل تغییر می‌کند، نیازی به کامپایل مجدد همه فایل‌ها نیست. فقط فایل‌ها که تغییر کرده دوباره به .o کامپایل می‌شود و با هم از نو لینک می‌کنیم.

Makefile

در عمل چنین قوانینی را داریم:

```
file.o: file.cpp file.hpp file2.hpp
g++ -c file.cpp -o file.o
```

هر **object file** به یک **cpp** و چندین **hpp** وابسته است. اینجا هدف (**target**) ساخت **file.o** است که به 3 فایل جلویش وابسته است. در صورت جدیدتر بودن یکی از آنها از **file.o**، دستور زیرش اجرا می‌شود که فایل را کامپایل می‌کند.

متغیرهای اتوماتیک (Automatic Variables)

برای جلوگیری از تاپ تکراری، **Makefile** چندین **automatic variable** دارد که نوشتن قوانین را راحت می‌کنند. چند تا از آنها را با استفاده از یک مثال نام می‌بریم:

```
file.o: file.cpp file.hpp file2.hpp
```

```
$(@) = file.o
      (the target)
$^ = file.cpp file.hpp file2.hpp
     (all prerequisites)
$< = file.cpp
     (the first prerequisite)
$? = e.g. file.cpp file2.hpp
     (prerequisites newer than target)
```

با استفاده از آنها، قوانین تولید **object file**ها به شکل زیر نوشته می‌شوند:

```
file.o: file.cpp file.hpp file2.hpp
g++ -c $< -o $@
```

وابستگی‌ها (Dependencies)

ما می‌خواهیم فایل‌ها فقط در صورت تغییر مجدد کامپایل شوند. برای انجام این کار بحث **dependencies** قابل اجتناب نیست. وقتی یک فایل تغییر می‌کند، بدیهیست که به کامپایل مجدد نیاز دارد. ولی در حالتی دیگر، که وقتی فایلی که اینکلود شده هم تغییر می‌کند، باید از اول کامپایل کنیم چرا که تغییرات فایل هدر می‌تواند روی فایل ما تاثیر داشته باشد. در **Makefile**، برای هر فایل سورس (**translation unit**) همه **dependency**های آن را می‌نویسیم. برنامه **Make** با توجه به تاریخ تغییر فایل‌ها، اگر یک فایلی نیازمند کامپایل مجدد بود، آن را انجام می‌دهد. برای استفاده، فایل متنی به نام **Makefile** یا **makefile** را در تاپ‌لول پروژه‌مان می‌سازیم و دستور **make** را در ترمینالی که در محل پروژه باز است اجرا می‌کنیم. کامنت کردن در **Makefile** با استفاده از کاراکتر **#** انجام می‌شود.

قانون (Rule)

یک قانون میک‌فایل از 3 بخش تشکیل شده است:

```
target: prerequisites
      recipe
```

target فایلیست که می‌خواهیم تولید کنیم، **prerequisites** وابستگی‌ها (**dependency**) می‌باشند و **recipe** دستوریست که در صورت جدیدتر بودن یکی از **prerequisite**ها از **target** اجرا می‌شود. توجه کنید که **recipe** باید با کاراکتر **تب** و نه اسپیس شروع شود. در غیر این صورت با ارور **missing separator** مواجه می‌شویم.

متغیرها (Variables)

در Makefile برای افزایش خوانایی کد، می‌توان متغیرها را به شکل زیر تعریف کرد:

```
FILES = file.o file2.o
```

در صورت طولانی بودن تعریف متغیر (یا هر خطی از جمله prerequisite)، می‌توان با استفاده از کاراکتر \ ادامه تعریف را به خط بعدی برد:

```
FILES = src/file1.cpp \  
src/file2.cpp
```

برای دسترسی به مقدار این متغیر، از variable expansion استفاده می‌کنیم:

```
$(FILES) or ${FILES}
```

برای متغیرها، علاوه بر =، می‌توان از چند اپراتور دیگر هم استفاده کرد که شامل = و += و ?= است.

فرق = و := در وقتی است که در تعریف متغیر یک expansion داشته باشیم. وقتی متغیر با = مقدار دهی شده باشد، هر بار که به مقدارش دست می‌یابیم متغیرهای داخلش را از نو expand می‌کند ولی := فقط یکبار و در همان خط، expansion را انجام می‌دهد.

با یک مثال نکته بالا را نشان می‌دهیم:

```
foo = abc  
bar = $(foo) bar  
foo = xyz  
# $(bar) is now "xyz bar"
```

در صورتی که در تعریف bar از := استفاده می‌کردیم، با تغییر مقدار foo مقدار آن تغییر نمی‌کرد.

با استفاده از += می‌توان به انتهای مقدار یک متغیر اضافه کرد. اپراتور ?= وقتی متغیر مقداری نداشته باشد، به آن مقدار سمت راست را می‌دهد.

توابع (Functions)

توابع در makefile داخل (\$) نوشته می‌شوند که اولین کلمه آن نام تابع است و در ادامه پارامترهای آن می‌آید.

مثلا از تابع info برای پرینت کردن استفاده می‌شود:

```
$(info Printing a variable: $(var))
```

با تابع wildcard می‌توان فایل‌ها را با * انتخاب کرد:

```
$(wildcard *.txt)
```

با استفاده از تابع shell می‌توان خروجی یک دستور ترمینال را گرفت. مثلا تابع زیر همه فایل‌های .cpp داخل فولدر src را می‌دهد:

```
FILES = $(shell find src/ -name "*.cpp" -type f)
```

توابع firstword و lastword کلمه اول و آخر یک رشته (یا مقدار متغیر) را می‌دهند.

تابع filter برای نگه داشتن کلماتی که با pattern خاصی مچ می‌شوند استفاده می‌شود. تابع filter-out برعکس این کار را انجام می‌دهد و آنهایی که مچ هستند را حذف می‌کند.

مثال زیر main.cpp و همه فایل‌هایی که با .o تمام می‌شوند را از VAR حذف می‌کند:

```
VAR2 = $(filter-out main.cpp %.o, $(VAR))
```

تابع patsubst که مخفف pattern substitution است، به شکل زیر است:

```
$(patsubst pattern, replacement, text)
```

این تابع بر روی آیتم‌های text (که می‌تواند خروجی یک متغیر باشد) که با pattern مچ هستند، replacement را اجرا می‌کند. برای مثال:

```
$(patsubst %.o,%.cpp,file.o file2.s file3.o)
```

خروجی این تابع file.cpp و file3.cpp است.

هدف (Target)

اگر در command-line جلوی دستور make نام target-ای را بیاوریم، آن یک قانون را به طور خاص اجرا می‌کند. در صورت اجرای make خالی، اولین قانون در فایل اجرا می‌شود.

قوانینی که کاری با فایلی به نام target ندارند را در تارگتی به نام PHONY قرار می‌دهیم. مانند قانون print-help که بالاتر نوشته شد و یا تارگت مرسوم all که در آن همه خروجی‌های برنامه را تولید می‌کنیم:

```
.PHONY: all print-help
all: $(EXE)
```

همانطور که می‌بینیم، یک قانون می‌تواند recipe-ای نداشته باشد و فقط چند تارگت را صدا بزند:

```
.PHONY: clean clean-obj clean-exe
clean: clean-obj clean-exe
```

با استفاده از PHONY مطمئن می‌شویم که اگر فایلی هم نام target بود مشکلی ایجاد نمی‌شود.

مثال کامل

از آنجا که قانون ساخت EXE کار linker را انجام می‌دهد، برای ورودی تمامی 0ها به آن داده شده است (\$^).

```
FILES = file.o file2.o
EXE = program.out
CXX = g++ -std=c++11

all: $(EXE)
$(EXE): $(FILES)
    $(CXX) $^ -o $@
file.o: file.cpp file.hpp file2.hpp
    $(CXX) -c $< -o $@
file2.o: file2.cpp file2.hpp
    $(CXX) -c $< -o $@

.PHONY: all clean
clean:
    rm -f $(FILES) $(EXE)
```

یک ساختار میک‌فایل که مشابه patsubst کار می‌کند substitution reference است:

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

```
$(var:%.o=%.cpp)
```

معادل مثال بالا:

دستور (Recipe)

بخش recipe یک قانون می‌تواند از چند دستور در چند خط نیز تشکیل شده باشد. دستورها در یک subshell اجرا می‌شوند و خود دستور اجرا شده نیز در ترمینال پرینت می‌شود.

برای جلوگیری از پرینت خود دستور می‌توان قبل از آن کاراکتر @ را گذاشت:

```
print-help:
    echo This prints echo and the text
    @echo This doesn't print the echo command
```

برای جلوگیری از پرینت شدن خروجی دستور می‌توان آن را به null device ری‌دایرکت کرد:

```
command arg >/dev/null
    # redirect stdout to null
command arg >/dev/null 2>&1
    # redirect errors to stdout as well
```

در صورتی که target هیچ prerequisite-ای نداشته باشد، می‌توان خط اول recipe را به شکل زیر نوشت:

```
target: ; recipe
```

وقتی که در حین اجرای یک recipe دستوری ارور دهد، اجرای make تمام می‌شود. می‌توان با گذاشتن - قبل از دستور

```
-g++ file.cpp
```

ارورها را ایگنور کرد:

CMake، یک جایگزین؟

Build System

بیلد سیستم‌ها برنامه‌هایی اند که برای تولید خودکار خروجی قابل اجرای کد ما استفاده می‌شوند.

Makefile یک بیلد سیستم محسوب می‌شود و پس از توصیف پروژه‌مان به آن، می‌تواند به صورت خودکار پروژه را کامپایل کند و وابستگی‌ها را در نظر می‌گیرد. هر زبان برنامه نویسی بیلد سیستم‌های خودش را دارد. برای مثال در زبان جاوا Maven و Gradle مشهورند.

اکوسیستم سی++

سی++ تنها یک استاندارد است و کامپایلرها، پیاده‌سازی‌ای از آن استاندارد هستند. از آنجا که در استاندارد گفته نشده، ابزار استاندارد برای مدیریت بیلد و لایبرری و... وجود ندارد. از این رو ابزارهای متعددی در این راستا توسط جامعه توسعه‌دهندگان سی++ ساخته شده است.

چند مورد از این ابزارها، Makefile، Bazel و هر IDE از جمله Visual Studio که روش خودشان را برای هندل کردن پروژه‌ها دارند. یا مثلا Ninja که با هدف سریع بودن بیلدها ساخته شده است.

ولی برخی ابزار، خودشان هدف بیلد کردن را ندارند. بلکه چیزی high-levelتر هستند که با نوشتن یک توصیف پروژه، می‌توانند آن را به توصیف مورد نیاز برای بیلد سیستم‌های مختلف تبدیل کنند.

از جمله این برنامه‌ها Premake و CMake است.

مشکل محیط توسعه

همانطور که معلوم است، Build Systemها الزاما cross-platform نیستند.

در یک پروژه که افراد زیادی روی آن کار می‌کنند، معمولا نمی‌شود همه را محدود به یک سری ابزار خاص برای کد زدنشان کرد. مثلا یکی روی VS کار می‌کند (که از file solution و vcxproj استفاده می‌کند) و یکی در لینوکس کد می‌زند و از Makefile استفاده می‌کند.

با استفاده از ابزاری مثل CMake، می‌توان برای اکثر Build Systemها و IDEها فایل توصیف پروژه را تولید کرد و فقط یک توصیف اصلی در فرمت خود CMake (که در فایل CMakeLists.txt نوشته می‌شود) داشت.

CMake

همانطور که گفته شد، CMake یک برنامه برای تولید فایل پروژه‌های مختلف از یک شرح پروژه مشترک است. یعنی CMake خود کار بیلد شدن را مستقیم انجام نمی‌دهد بلکه به بیلد سیستم‌های دیگر می‌سپارد.

CMake از زبان توصیفی خود استفاده می‌کند و پس از نوشتن CMakeLists.txt، با اجرای برنامه می‌توانیم فایل‌های بیلد سیستم مورد نظرم را دریافت کنیم.

این ابزار dependencyها را هم به طور خودکار هندل می‌کند. این یعنی نیازی به نوشتن دستی تمامی prerequisiteهای object fileها نیست.

یک مثال ساده

فرض کنید سه فایل زیر را در پروژه داریم:

```
├── foo.cpp
├── foo.hpp
└── main.cpp
```

داخل foo.cpp یک تابع تعریف شده که declaration آن در foo.hpp قرار دارد. در نهایت این تابع توسط main صدا زده می‌شود. در ادامه برای بیلد کردن این پروژه کوچک، از CMake استفاده می‌کنیم.

قدم اول

در اولین مرحله ابتدا باید ورژنی از CMake را که از آن استفاده می‌کنیم مشخص کنیم؛ این موضوع باعث می‌شود توابع CMake-ای که استفاده می‌کنیم توسط ورژن مناسبی از CMake که آن‌ها را پشتیبانی می‌کند اجرا شوند. برای این کار از تابع زیر استفاده می‌کنیم:

```
cmake_minimum_required(VERSION <min>)
```

بلافاصله بعد از تعیین ورژن CMake باید نام پروژه را تعیین کنیم. این کار از طریق تابع زیر قابل انجام است:

```
project(<PROJECT-NAME>
  [VERSION <major>[.<minor>[.<patch>]]]
  [DESCRIPTION <project-description-string>]
  [HOMEPAGE_URL <url-string>]
  [LANGUAGES <language-name>...])
```

بنابراین در انتها برای پروژه خودمان دو خط زیر را به فایل CMakeLists.txt اضافه می‌کنیم:

```
cmake_minimum_required(VERSION 3.12)
project(foo_proj
  VERSION 1.0
  DESCRIPTION "A project to introduce CMake"
  LANGUAGES CXX)
```

قدم دوم

پس از تعیین ورژن CMake و نام پروژه، باید فایل مد نظر خروجی به همراه فایل کدهای آن را تعیین کنیم. فایل خروجی می‌تواند مثلاً فایلی قابل اجرا یا یک لایبرری باشد. به خروجی‌ها target می‌گویند.

```
add_executable(<name>)
```

متغیر <name> باید در یک پروژه یکتا باشد. حال می‌توان برای target ساخته‌شده فایل‌های کد را تعیین کرد:

```
target_sources(<name> PRIVATE
  <source1> <source2>...)
```

علاوه بر آن واضح است که تنها یک تابع main باید در فایل‌های لیست‌شده وجود داشته باشد.

```
add_executable(foo)
target_sources(foo PRIVATE
  main.cpp foo.cpp)
```

اکنون می‌شود برای target ساخته‌شده برخی مشخصات از جمله نسخه سی++ را تعیین کرد:

```
target_compile_features(foo PRIVATE cxx_std_11)
```

قدم آخر

اکنون برای بیلد کردن پروژه، ابتدا یک فولدر build می‌سازیم و داخل آن دستور زیر را وارد می‌کنیم:

```
cmake ..
```

با این کار، فایل‌های build system پلتفرم فعلی در فولدر کنونی (build) با استفاده از CMakeLists.txt فولدر قبلی (..) ساخته می‌شود.

پس از ساخته شدن بیلد سیستم مورد نظر (مثلاً یک Makefile)، با دستور مورد نظر آن سیستم، فایل قابل اجرا را می‌سازیم (مثلاً دستور make را می‌زنیم). اکنون پروژه بیلد شده و آماده اجرا می‌باشد.


```
vector<intT> f
ntT> res;
tT(2) >= j) {if
= j; j = intT(2
es.push_back(nu
raIT; int main
egraIT n; while
umber: "; cin >
= v.begin(); i
()) cout << ",
; } return 0; }
t(int N) { if (
urn 1; long do
; i++) { result
n() { int N; co
ial " << N << "
("pause"); retu
; i++) cout <<
int *num = new
; for(
; i++) num[i] =
% (last-first+1
e(a[i]<mid) i++
while(a
a[i],a[j]); i++
num[j]; j--;
```

تاریخ انتشار: بهار ۱۴۰۲

نویسندگان: سامان اسلامی نظری، میثاق محقق

طراحان: الهه خداوردی، شهریار عطار

دستیار آموزشی ارشد: طاها فخاریان

