Kompilacja i scalanie programów w linii poleceń *gcc i make*

Małgorzata Stankiewicz

kwiecień 2012

GCC - GNU Compiler Collection

GCC

Zestaw kompilatorów do różnych języków programowania rozwijany w ramach projektu GNU i udostępniany na licencji GPL oraz LGPL.
GCC jest podstawowym kompilatorem w systemach uniksopodobnych, przy czym szczególnie ważną rolę odgrywa w procesie budowy jądra Linux.

Zawartość GCC

Zainstalowane programy: c++, cc (link to gcc), cpp, g++, gcc, gccbug, gcov
Zainstalowane katalogi: /usr/include/c++, /usr/lib/gcc, /usr/share/gcc-4.6.2
Zainstalowane biblioteki: libgcc.a, libgcc_eh.a, libgcc_s.so, libgcov.a, libgomp.{a,so}, liblto_plugin.so, libmudflap.{a,so}, libmudflapth.{a,so}, libquadmath.{a,so}, libssp.{a,so}, libssp_nonshared.a, libstdc++.{a,so}, libsupc++.a

Opis zawartości

c++ Kompilator C++

Opis zawartości

c++ Kompilator C++

cc Kompilator C

Opis zawartości

c++ Kompilator C++

cc Kompilator C

cpp Preprocesor C; wykorzystywany przez kompilator do rozwinięcia #include, #define i im podobnych w plikach źródłowych

Opis zawartości

c++	Kompilator C++
cc	Kompilator C
cpp	Preprocesor C; wykorzystywany przez kompilator do rozwinięcia #include, #define i im podobnych w plikach źródłowych
g++	Kompilator C++

Opis zawartości

c++	Kompilator C++
cc	Kompilator C
cpp	Preprocesor C; wykorzystywany przez kompilator do rozwinięcia #include, #define i im podobnych w plikach źródłowych
g++	Kompilator C++
gcc	Kompilator C

Opis zawartości

c++	Kompilator C++
cc	Kompilator C
cpp	Preprocesor C; wykorzystywany przez kompilator do rozwinięcia #include, #define i im podobnych w plikach źródłowych
g++	Kompilator C++
gcc	Kompilator C
gccbug	Skrypt powłoki wykorzystywany do ułatwionego tworzenia raportów błędów

Opis zawartości

c++	Kompilator C++
cc	Kompilator C
cpp	Preprocesor C; wykorzystywany przez kompilator do rozwinięcia #include, #define i im podobnych w plikach źródłowych
g++	Kompilator C++
gcc	Kompilator C
gccbug	Skrypt powłoki wykorzystywany do ułatwionego tworzenia raportów błędów
libgcc	Run-time support dla gcc

Opis zawartości

c++	Kompilator C++
cc	Kompilator C
cpp	Preprocesor C; wykorzystywany przez kompilator do rozwinięcia #include, #define i im podobnych w plikach źródłowych
g++	Kompilator C++
gcc	Kompilator C
gccbug	Skrypt powłoki wykorzystywany do ułatwionego tworzenia raportów błędów
libgcc	Run-time support dla gcc
libssp	Zawiera procedury wspierające funkcjonalności chroniące przed przepełnieniem stosu

Opis zawartości

c++	Kompilator C++
cc	Kompilator C
cpp	Preprocesor C; wykorzystywany przez kompilator do rozwinięcia #include, #define i im podobnych w plikach źródłowych
g++	Kompilator C++
gcc	Kompilator C
gccbug	Skrypt powłoki wykorzystywany do ułatwionego tworzenia raportów błędów
libgcc	Run-time support dla gcc
libssp	Zawiera procedury wspierające funkcjonalności chroniące przed przepełnieniem stosu
libstdc++	Standardowa biblioteka C++

Opis zawartości

c++	Kompilator C++
cc	Kompilator C
cpp	Preprocesor C; wykorzystywany przez kompilator do rozwinięcia #include, #define i im podobnych w plikach źródłowych
g++	Kompilator C++
gcc	Kompilator C
gccbug	Skrypt powłoki wykorzystywany do ułatwionego tworzenia raportów błędów
libgcc	Run-time support dla gcc
libssp	Zawiera procedury wspierające funkcjonalności chroniące przed przepełnieniem stosu
libstdc++	Standardowa biblioteka C++
libsupc++	Dostarcza procedury wsparcia dla języka programowania C++

Główne opcje gcc

-x language	Określa rodzaj wchodzących plików (dla wszystkich plików za opcją). Mozliwe wartości: c, objective-c, c-header, c++, cpp-output, assembler, assembler-with-cpp.
-x	Wyłącza określanie wchodzących plików (patrz wiersz zwyżej).
-E	Tylko preprocesing, na wyjściu dostajemy pliki zródłowe przerobione przez preprocesor. Standardowe końcówki plików (.c, .cpp, .i, etc...) są zamieniane na .i, Opcją -o możemy podać inną nazwę (końcówkę).
-S	Zatrzymuj po poziomie kompilacji, nie assembluj, na wyjściu mamy plik źródłowym z kodem assemblera. Standardowe końcówki plików (.c, .cpp, .i, etc...) są zamieniane na .s, Opcją -o możemy podać inną nazwę (końcówkę).
-c	Zatrzymuj po poziomie kompilacji lub asemblacji, bez linkowania. Na wyjściu dostajemy pliki typu obiekt ("object file") dla każdego pliku źródłowego. Standardowe końcówki plików (.c, .cpp, .i, etc...) są zamieniane na .o. Opcją -o możemy podać inną nazwę (końcówkę).
-o *nazwa*	Wskazanie nazwy pliku wynikowego dla danej operacji (przy kompilacji standardowo dostajemy **a.out**).
-v	Wypisywanie (na standardowym wyjściu dla błędów), komend wywoływanych podczas kolejnych etapów kompilacji. Wypisuje również numer wersji: programu sterującego kompilatorem, preprocesora oraz właściwego kompilatora.

Optymalizacja i linker

Opcje linkera.

-l*library*	Użyj biblioteki *library* kiedy linkujesz. Uwaga! Gcc automatycznie dodaje przedrostel **lib** i końcówkę **.a**, np. -l*FOX* w celu załadowania **libFOX.a**. Patrz też `-**L**'.
-**nostdlib**	Nie używaj standardowych bibliotek systemowych i startowych plików kiedy linkujesz. Używaj tylko wskazane.
-**lg++**	Programy w C++ czesto mogą wymagać tej opcji dla pełnego, poprawnego linkowania.

Opcje optymalizacji.

-**O** -**O**level	Optymalizacja (dużo czasu, dużo pamięci :). Poziom optymalizacji: 0,1,2,3, jesli 0, to brak optymalizacji.

Opcje preprocesora

-include *file*	Najpierw przetwarzaj plik *file* (np. kompiluj najpierw file).
-imacros *file*	Najpierw przetwarzaj plik *file*, wypluj wyniki na wyjście, zanim zaczniesz przetwarzać resztę plików.
-idirafter *dir*	Dodaj katalog dir jako drugi katalog do przeszukiwania dla plików nagłówkowych. Jeśli plik nagłówkowy nie został znaleziony w żadnym ze wskazanych wcześniej katalogów, przeszukuj ten katalog.
-nostdinc	Nie szukaj w katalogu ze standardowymi plikami nagółwkowymi, szukaj tylko w katalogach wskazanych przez `-**I**' i w katalogu bieżącym.
-nostdinc++	Nie przeszukuj katalogów z plikami nagłówkowymi specyficznymi dla c++ (np. /usr/include/g++), ale szukaj w innych standardowych.
-nostdinc	Nie szukaj w katalogu ze standardowymi plikami nagółwkowymi, szukaj tylko w katalogach wskazanych przez `-**I**' i w katalogu bieżącym.
-H	Wypisz nazwę każdego używanego pliku nagłówkowego.
-**D**macro	Ustaw macro na `1'.
-**D**macro=defn	Zdefinjuj makro macro jako defn.
-**U**macro	Skasuj definicje makra macro.

W pracy gcc wyróżnić można pięć poziomów:

- Preprocesing - preprocesorowanie, wywołanie preprocesora
- Compilation - kompilacja, właściwa kompilacja
- Optymalization - optymalizacja kodu (opcjonalnie)
- Assembling - asemblacja, kompilacja z użyciem asemblera
- Linking - linkowanie (konsolidacja), dołącznie bibliotek

Prekompilacja (preprocesing)

Etap ten odpowiada za stworzenie ostatecznego kodu źródłowego aplikacji. Zostają dołączone pliki wskazane dyrektywą preprocesora #include, zostają wykonane podstawienia makrodefinicji stworzonych przez #define, usunięte komentarze oraz puste linie. Tworzony jest plik z rozszerzeniem zgodnym ze specyfikacją.
Aby wykonać samą prekompilacje pliku źródłowego należy uruchomić gcc z przełącznikiem -E

Kompilacja właściwa (compilation)

Etap ten odpowiada za kompilacje prekompilowanego kodu źródłowego do kodu asemblera. To na tym etapie kompilator wychwytuje błędy w kodzie źródłowym i zgłasza je użytkownikowi. Tworzony jest plik asemblera z rozszerzeniem .s.
Aby przerwać proces kompilacji na tym poziomie(asemblacja i linkowanie się nie odbędzie), należy uruchomić gcc z przełącznikiem -S

Optymalizacja kodu asemblera (optymalization)

Etap ten odpowiada ze wprowadzenie zmian w kodzie asemblera, które docelowo mają zwiększyć jego efektywność(zwiększyć wydajność aplikacji oraz zmniejszyć jej rozmiar). Aby tego dokonać, kompilator między innymi eliminuje nieużywane fragmenty kodu, optymalizuje przydział rejestrów, usprawnia sposób obliczania adresów względnych.

Optymalizacja

- Poziom 0 - kompletnie wyłącza optymalizacje.

Optymalizacja

- Poziom 0 - kompletnie wyłącza optymalizacje.
- Poziom 1 - prowadzi do stworzenia mniejszego i szybszego kodu bez straty dużej ilości czasu podczas kompilacji. Bardzo podstawowy poziom, często nie przynoszący widocznych efektów.

Optymalizacja

- Poziom 0 - kompletnie wyłącza optymalizacje.
- Poziom 1 - prowadzi do stworzenia mniejszego i szybszego kodu bez straty dużej ilości czasu podczas kompilacji. Bardzo podstawowy poziom, często nie przynoszący widocznych efektów.
- Poziom 2 - zalecany poziom optymalizacji - kompilator będzie próbował zwiększyć wydajność kodu za cenę jego rozmiaru oraz czasu kompilacji.

Optymalizacja

- Poziom 0 - kompletnie wyłącza optymalizacje.
- Poziom 1 - prowadzi do stworzenia mniejszego i szybszego kodu bez straty dużej ilości czasu podczas kompilacji. Bardzo podstawowy poziom, często nie przynoszący widocznych efektów.
- Poziom 2 - zalecany poziom optymalizacji - kompilator będzie próbował zwiększyć wydajność kodu za cenę jego rozmiaru oraz czasu kompilacji.
- Poziom 3 - nie zalecany dla GCC 4.x - kod będzie miał duży rozmiar, wykonanie będzie mogło wymagać więcej pamięci, a czas kompilacji mocno się wydłuży. Twórcy GCC nie zalecają używania tego poziomu optymalizacji.

Optymalizacja

- Poziom 0 - kompletnie wyłącza optymalizacje.
- Poziom 1 - prowadzi do stworzenia mniejszego i szybszego kodu bez straty dużej ilości czasu podczas kompilacji. Bardzo podstawowy poziom, często nie przynoszący widocznych efektów.
- Poziom 2 - zalecany poziom optymalizacji - kompilator będzie próbował zwiększyć wydajność kodu za cenę jego rozmiaru oraz czasu kompilacji.
- Poziom 3 - nie zalecany dla GCC 4.x - kod będzie miał duży rozmiar, wykonanie będzie mogło wymagać więcej pamięci, a czas kompilacji mocno się wydłuży. Twórcy GCC nie zalecają używania tego poziomu optymalizacji.
- -Os - nacisk na zmniejszenie rozmiaru kodu + flagi optymalizacyjne jak w poziomie 2

Asemblacja (assembling)

Na tym poziomie następuje przetworzenie kodu asemblera na kod maszynowy. Kod maszynowy(binarny) umieszczany jest w pliku obiektowym (Object file) z rozszerzeniem .o.

Inline Assembler w C/C++

GCC umożliwia użycie asemblera w kodzie. Nie są to jednak pojedyncze instrukcje, tylko całe bloki razem ze zdefiniowanymi specjalnym systemem interfejsem między asemblerem a C/C++.
Dzięki temu GCC może o wiele lepiej optymalizować kod.

Konsolidacja, linkowanie (linking)

Na tym poziomie, GCC wykonuje trzy czynności które dopełniają proces kompilacji. Linker szuka w bibliotekach systemowych lub wskazanych przez użytkownika kodu który nie został zdefiniowany w plikach źródłowych. Następnie, przypisuje kod maszynowy do ustalonych adresów. Na końcu zaś, tworzy wykonywalny plik binarny ELF(Executable and Linking File)

MAKE

make

Program powłoki systemowej automatyzujący proces kompilacji programów, na które składa się wiele zależnych od siebie plików. Program przetwarza plik reguł Makefile i na tej podstawie stwierdza, które pliki źródłowe wymagają kompilacji. Zaoszczędza to wiele czasu przy tworzeniu programu, ponieważ w wyniku zmiany pliku źródłowego kompilowane są tylko te pliki, które są zależne od tego pliku. Dzięki temu nie ma potrzeby kompilacji całego projektu. Make nadaje się również do innych prac, które wymagają przetwarzania wielu plików zależnych od siebie.

Parametry wywołania make

Wywołanie programu **make** ma następującą postać:

make [opcje] [makrodefinicje] [-f plik_sterujący] [cel]

W najprostszym przypadku linia polecenia zawiera tylko słowo make. Program próbuje wtedy odczytać polecenia z pliku sterującego o standardowej nazwie: **m a k e f i l e**, Makefile lub MakeFile. Jeśli w bieżącym katalogu nie ma pliku o takiej nazwie to zgłaszany jest błąd. Jeśli plik sterujący nosi inną nazwę to należy użyć wywołania:

make -f nazwa_pliku_sterującego

W linii wywołania można zdefiniować nowe lub zmienić istniejące już makrodefinicje, np.: **make „CC=gcc”** Można też polecić osiągnięcie innego celu niż domyślny, np.:
make all, make clean lub make install

Makefile

Makefile

Plik reguł dla programu make. Zawiera opis zależności pomiędzy plikami źródłowymi programu. Umożliwia to przetwarzanie tylko tych plików, które się zmieniły od ostatniej kompilacji i plików od nich zależnych. Skraca to znacznie czas generowania pliku wynikowego. Format pliku różni się w zależności od implementacji programu make, ale podstawowe reguły są takie same dla wszystkich odmian make.

CEL: SKŁADNIKI
KOMENDA

CEL to nazwa pliku docelowego, który jest tworzony z plików wymienionych jako SKŁADNIKI, zaś KOMENDA podaje komendę, która tworzy plik docelowy CEL z plików składowych SKŁADNIKI

CEL: SKŁADNIKI KOMENDA

CEL to nazwa pliku docelowego, który jest tworzony z plików wymienionych jako SKŁADNIKI, zaś KOMENDA podaje komendę, która tworzy plik docelowy CEL z plików składowych SKŁADNIKI

```
helloworld.o: helloworld.c
	gcc helloworld.c -o helloworld.o
```

Przykładowo

Program składający się z:

- dwóch plików źródłowych program.cpp, lib.cpp
- jednego nagłówkowego lib.h
- lib.h jest dołączany do obu plików źródłowych

Przykładowo

Program składający się z:

- dwóch plików źródłowych program.cpp, lib.cpp
- jednego nagłówkowego lib.h
- lib.h jest dołączany do obu plików źródłowych

Ręczna kompilacja i konsolidacja tego programu wygląda tak:

g++ -c -o lib.o lib.cpp - *kompilacja lib.cpp*
g++ -c -o program.o program.cpp - *kompilacja program.cpp*
g++ -o program program.o lib.o - *konsolidacja*

Przykładowo

Program składający się z:

- dwóch plików źródłowych program.cpp, lib.cpp
- jednego nagłówkowego lib.h
- lib.h jest dołączany do obu plików źródłowych

Ręczna kompilacja i konsolidacja tego programu wygląda tak:

g++ -c -o lib.o lib.cpp - *kompilacja lib.cpp*
g++ -c -o program.o program.cpp - *kompilacja program.cpp*
g++ -o program program.o lib.o - *konsolidacja*

Zawartość Makefile:

```
program: program.o lib.o
        g++ -o program program.o lib.o
program.o: program.cpp lib.h
        g++ -c -o program.o program.cpp
lib.o: lib.cpp lib.h
        g++ -c -o lib.o lib.cpp
```

W pliku znajdują się 3 reguły:

1. **Target program** – reguła dotyczy konsolidacji. Plikiem wynikowym tej reguły jest plik wykonywalny. Do utworzenia tego pliku kompilator potrzebuje plików program.o i lib.o.

   program: program.o lib.o
   g++ -o program program.o lib.o

2. **Target program.o** – Plik program.o powstaje z pliku program.cpp z dołączonym lib.h.

   program.o: program.cpp lib.h
   g++ -c -o program.o program.cpp

3. **Target lib.o** – Powstaje z pliku lib.cpp z dołączonym lib.h. lib.o:

   lib.cpp lib.h
   g++ -c -o lib.o lib.cpp

Sprzątanie

W pliku Makefile istnieje możliwość tworzenia targetów, które nie są nazwami plików wynikowych, a jedynie nazwami akcji do wykonania. Najpopularniejszą z takich akcji jest clean, którego zadaniem jest *usunięcie wszystkich kompilatów* pośrednich.

Realizuje się to poprzez wpisanie w pliku Makefile następującej reguły:

```
.PHONY: clean
clean:
	rm -f *.o
```

a następnie wpisywanie w terminalu komendy:

```
make clean
```

Wiele kompilatów

Zdarza się w jednym projekcie występuje kilka plików wykonywalnych. Aby zbudować kilka binarek za pomocą jednego pliku Makefile należy **zdefiniować target all** i uzależnić go od wszystkich plików wykonywalnych.

```
all: program1 program2
```

po wpisaniu w terminal komendy: **make** zbudują się wszystkie pliki.

Zmienne w pliku Makefile

W pliku Makefile można definiować zmienne:

```
OBJS=hello.o aux.o

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

hello.o: hello.c
	gcc -c hello.c -o hello.o

aux.o: aux.c
	gcc -c aux.c -o aux.o
```

Zmienne standardowe

W Makefile można używać wielu zmiennych zdefiniowanych standardowo. Najczęściej używane zmienne standardowe:

- CC - nazwa kompilatora języka C
- CXX - nazwa kompilatora języka C++
- CFLAGS - opcje kompilatora języka C
- CXXLAGS - opcje kompilatora języka C
- LFLAGS - opcje dla linkera

Zmienne standardowe mają pewną predefiniowaną wartość (np. zmienna CC ma predefiniowaną wartość 'cc'), którą jednak można zmieniać.

```
CC=gcc
CFLAGS=-g
OBJS=hello.o aux.o

hello: $(OBJS)
	$(CC) $(LFLAGS) $(OBJS) -o hello

hello.o: hello.c
```

Najczęściej używane opcje:

- d włącza tryb szczegółowego śledzenia

Najczęściej używane opcje:

- d włącza tryb szczegółowego śledzenia
- f plik_sterujący umożliwia stosowanie innych niż standardowe nazw plików sterujących

Najczęściej używane opcje:

- d włącza tryb szczegółowego śledzenia
- f plik_sterujący umożliwia stosowanie innych niż standardowe nazw plików sterujących
- i powoduje ignorowanie błędów kompilacji

Najczęściej używane opcje:

- d włącza tryb szczegółowego śledzenia
- f plik_sterujący umożliwia stosowanie innych niż standardowe nazw plików sterujących
- i powoduje ignorowanie błędów kompilacji
- n powoduje wypisywanie poleceń na ekran zamiast ich wykonywania

Najczęściej używane opcje:

- d włącza tryb szczegółowego śledzenia
- f plik_sterujący umożliwia stosowanie innych niż standardowe nazw plików sterujących
- i powoduje ignorowanie błędów kompilacji
- n powoduje wypisywanie poleceń na ekran zamiast ich wykonywania
- p powoduje wypisywanie makrodefinicji (zmiennych) i reguł transformacji

Najczęściej używane opcje:

- d włącza tryb szczegółowego śledzenia
- f plik_sterujący umożliwia stosowanie innych niż standardowe nazw plików sterujących
- i powoduje ignorowanie błędów kompilacji
- n powoduje wypisywanie poleceń na ekran zamiast ich wykonywania
- p powoduje wypisywanie makrodefinicji (zmiennych) i reguł transformacji
- s wyłącza wypisywanie treści polecenia przed jego wykonaniem