Zbuduj mi ten projekt, proszę.

Wprowadzenie do GNU Make

Damian Bulira

damian.bulira@pwr.wroc.pl

IX Sesja Linuksowa

21 kwietnia 2012

Table of contents

Plan wykładu

Software Configuration Management

- Dział inżynierii oprogramowania
- Kontrola wersji
- Zarządzanie procesem rozwoju oprogramowania
- Zarządzanie środowiskiem programistycznym
- Dostarczanie oprogramowania do klienta
- Zarządzanie budowaniem (Build management)

Software Configuration Management

- Dział inżynierii oprogramowania
- Kontrola wersji
- Zarządzanie procesem rozwoju oprogramowania
- Zarządzanie środowiskiem programistycznym
- Dostarczanie oprogramowania do klienta
- Zarządzanie budowaniem (Build management)

Software Configuration Management

- Dział inżynierii oprogramowania
- Kontrola wersji
- Zarządzanie procesem rozwoju oprogramowania
- Zarządzanie środowiskiem programistycznym
- Dostarczanie oprogramowania do klienta
- Zarządzanie budowaniem (Build management)

Software Configuration Management

- Dział inżynierii oprogramowania
- Kontrola wersji
- Zarządzanie procesem rozwoju oprogramowania
- Zarządzanie środowiskiem programistycznym
- Dostarczanie oprogramowania do klienta
- Zarządzanie budowaniem (Build management)

Software Configuration Management

- Dział inżynierii oprogramowania
- Kontrola wersji
- Zarządzanie procesem rozwoju oprogramowania
- Zarządzanie środowiskiem programistycznym
- Dostarczanie oprogramowania do klienta
- Zarządzanie budowaniem (Build management)

Software Configuration Management

- Dział inżynierii oprogramowania
- Kontrola wersji
- Zarządzanie procesem rozwoju oprogramowania
- Zarządzanie środowiskiem programistycznym
- Dostarczanie oprogramowania do klienta
- **Zarządzanie budowaniem (Build management)**

Plan wykładu

Historia

- 1977 - Make - Stuart Feldman -
  W 2004 roku został on uhonorwany ACM Software System Award
- 1988 - GNU Make - Richard Stallman oraz Roland McGrath - pierwszy wpis w changelogu

ACM/Press Release March 24, 2004

Almost every software developer in the world has used Make, or one of its descendants, as a tool for maintaining computer software. [...] Make provides a mechanism for maintaining up-to-date versions of programs that have been altered by many operations on a number of files. It mechanizes many of the activities of program development and maintenance, and has played an integral role in products from virtually every major computer and software vendor.

Historia

- 1977 - Make - Stuart Feldman -
  W 2004 roku został on uhonorwany ACM Software System Award
- 1988 - GNU Make - Richard Stallman oraz Roland McGrath - pierwszy wpis w changelogu

ACM/Press Release March 24, 2004

Almost every software developer in the world has used Make, or one of its descendants, as a tool for maintaining computer software. [...] Make provides a mechanism for maintaining up-to-date versions of programs that have been altered by many operations on a number of files. It mechanizes many of the activities of program development and maintenance, and has played an integral role in products from virtually every major computer and software vendor.

Historia

- 1977 - Make - Stuart Feldman -
  W 2004 roku został on uhonorwany ACM Software System Award
- 1988 - GNU Make - Richard Stallman oraz Roland McGrath -
  pierwszy wpis w changelogu

ACM/Press Release March 24, 2004

Almost every software developer in the world has used Make, or one of its descendants, as a tool for maintaining computer software. [...] Make provides a mechanism for maintaining up-to-date versions of programs that have been altered by many operations on a number of files. It mechanizes many of the activities of program development and maintenance, and has played an integral role in products from virtually every major computer and software vendor.

Plan wykładu

Inne wersje Make'a

- pmake (BSD Make)
- nmake (Windows)
- nmake (Bell Labs / Alcatel-Lucent)
- emake (Electric Cloud)

Inne wersje Make'a

- pmake (BSD Make)
- nmake (Windows)
- nmake (Bell Labs / Alcatel-Lucent)
- emake (Electric Cloud)

Inne wersje Make'a

- pmake (BSD Make)
- nmake (Windows)
- nmake (Bell Labs / Alcatel-Lucent)
- emake (Electric Cloud)

Inne wersje Make'a

- pmake (BSD Make)
- nmake (Windows)
- nmake (Bell Labs / Alcatel-Lucent)
- emake (Electric Cloud)

Inne narzędzia

- scons
- waf
- ant
- cmake
- autotools

Inne narzędzia

- scons
- waf
- ant
- cmake
- autotools

Inne narzędzia

- scons
- waf
- ant
- cmake
- autotools

Inne narzędzia

- scons
- waf
- ant
- cmake
- autotools

Inne narzędzia

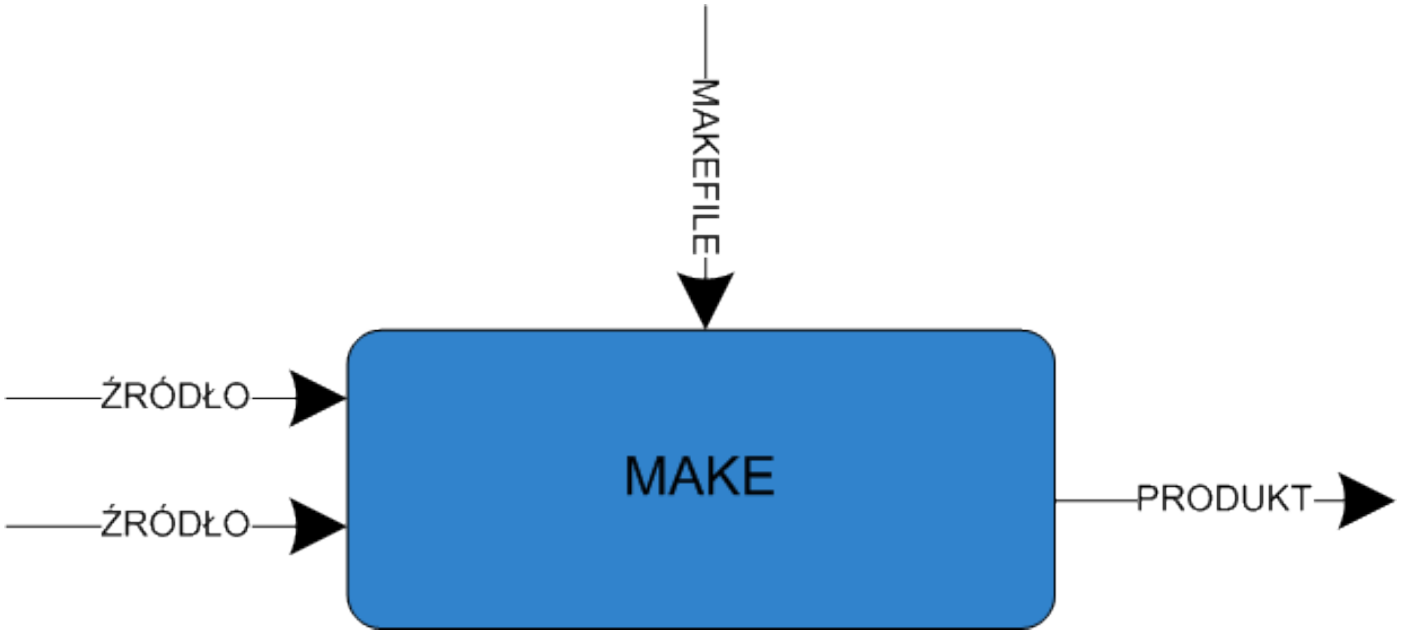
- scons
- waf
- ant
- cmake
- autotools

Inne narzędzia

- scons
- waf
- ant
- cmake
- autotools

Plan wykładu

Czym jest GNU Make?



Make jest narzędziem, które automatyzuje tworzenie plików z plików.

Plan wykładu

Do czego Make jest używany?

- Kompilacja kodu, linkowanie, itd.
- Operacje na plikach
  - Kompresja dźwięku (.wav -> .mp3)
  - Kompresja wideo
  - Kompilacja projektów TEX
  - Przetwarzanie obrazów
  - Tworzenie kopii zapasowych

Do czego Make jest używany?

- Kompilacja kodu, linkowanie, itd.
- Operacje na plikach
  - Kompresja dźwięku (.wav -> .mp3)
  - Kompresja wideo
  - Kompilacja projektów TEX
  - Przetwarzanie obrazów
  - Tworzenie kopii zapasowych

Do czego Make jest używany?

- Kompilacja kodu, linkowanie, itd.
- Operacje na plikach
  - Kompresja dźwięku (.wav -> .mp3)
  - Kompresja wideo
  - Kompilacja projektów TEX
  - Przetwarzanie obrazów
  - Tworzenie kopii zapasowych

Do czego Make jest używany?

- Kompilacja kodu, linkowanie, itd.
- Operacje na plikach
  - Kompresja dźwięku (.wav -> .mp3)
  - Kompresja wideo
  - Kompilacja projektów TEX
  - Przetwarzanie obrazów
  - Tworzenie kopii zapasowych

Do czego Make jest używany?

- Kompilacja kodu, linkowanie, itd.
- Operacje na plikach
  - Kompresja dźwięku (.wav -> .mp3)
  - Kompresja wideo
  - Kompilacja projektów TEX
  - Przetwarzanie obrazów
  - Tworzenie kopii zapasowych

Do czego Make jest używany?

- Kompilacja kodu, linkowanie, itd.
- Operacje na plikach
  - Kompresja dźwięku (.wav -> .mp3)
  - Kompresja wideo
  - Kompilacja projektów TEX
  - Przetwarzanie obrazów
  - Tworzenie kopii zapasowych

Do czego Make jest używany?

- Kompilacja kodu, linkowanie, itd.
- Operacje na plikach
  - Kompresja dźwięku (.wav -> .mp3)
  - Kompresja wideo
  - Kompilacja projektów TEX
  - Przetwarzanie obrazów
  - Tworzenie kopii zapasowych

Plan wykładu

Co nam daje Make?

Zrównoleglenie kompilacji

Niezależne zadania mogą być wykonywane w tym samym czasie na wielu rdzeniach, bez użycia specjalistycznych narzędzi (jobserver).

Śledzenie zależności

Make tworzy drzewo zależności pomiędzy zadaniami (targetami), przez co proces odbywa się w poprawnej kolejności po najkrótszej ścieżce.

Budowanie przyrostowe

Wykonuje jedynie te zadania, w których pliki wejściowe uległy zmianie.

Co nam daje Make?

Zrównoleglenie kompilacji

Niezależne zadania mogą być wykonywane w tym samym czasie na wielu rdzeniach, bez użycia specjalistycznych narzędzi (jobserver).

Śledzenie zależności

Make tworzy drzewo zależności pomiędzy zadaniami (targetami), przez co proces odbywa się w poprawnej kolejności po najkrótszej ścieżce.

Budowanie przyrostowe

Wykonuje jedynie te zadania, w których pliki wejściowe uległy zmianie.

Co nam daje Make?

Zrównoleglenie kompilacji

Niezależne zadania mogą być wykonywane w tym samym czasie na wielu rdzeniach, bez użycia specjalistycznych narzędzi (jobserver).

Śledzenie zależności

Make tworzy drzewo zależności pomiędzy zadaniami (targetami), przez co proces odbywa się w poprawnej kolejności po najkrótszej ścieżce.

Budowanie przyrostowe

Wykonuje jedynie te zadania, w których pliki wejściowe uległy zmianie.

Plan wykładu

Jak działa Make?

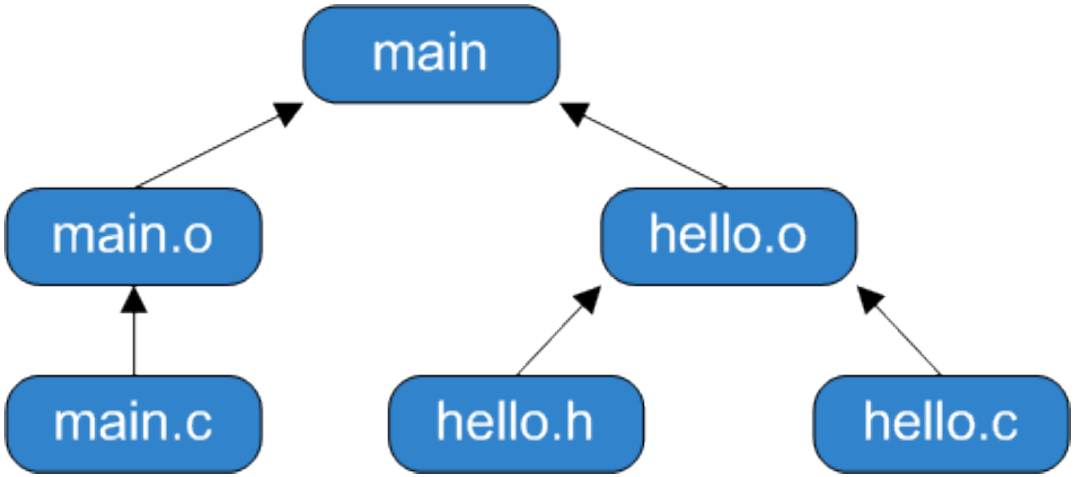
- **zależność** - warunek wstępny, przed wykonaniem procedury (receptury) tworzenia celu
- **receptura** - polecenia powłoki tworzące obiekt(y) docelowy
- **cel** - plik wynikowy (element nieobowiązkowy)

Kiedy przebudować projekt?

Znacznik czasu (timestamp)

```
-rw-r-r- 1 user group 66 Jan 10 13:55 main.c
```

Receptura jest wykonywana, kiedy znacznik czasowy jest nowszy w zależności niż w pliku docelowym, plik docelowy nie istnieje lub target jest zależny od .PHONY.



Plan wykładu

Składnia pliku Makefile

```
cel: zależność1 [zależność2] ...
	receptura
```

Uwaga

Receptura musi zaczynać się od znaku tabulacji.

Przykład 1

```
main: main.c hello.c
	gcc -o main main.c hello.c
```

Przykład 0

```
main: main.c hello.c
```

Składnia pliku Makefile

```
cel: zależność1 [zależność2] ...
	receptura
```

Uwaga

Receptura musi zaczynać się od znaku tabulacji.

Przykład 1

```
main: main.c hello.c
	gcc -o main main.c hello.c
```

Przykład 0

```
main: main.c hello.c
```

Składnia pliku Makefile

```
cel: zależność1 [zależność2] ...
	receptura
```

Uwaga

Receptura musi zaczynać się od znaku tabulacji.

Przykład 1

```
main: main.c hello.c
	gcc -o main main.c hello.c
```

Przykład 0

```
main: main.c hello.c
```

Wywołanie Make'a

```
$ make <target> [options|variables]
```

- Domyślnie Make czyta plik w katalogu z którego jest uruchamiany o nazwie makefile, Makefile lub GNUMakefile
  Dobrą praktyką jest używanie nazwy Makefile
- Jeśli target nie jest podany, to pierwszy niewieloznaczny (non-wildcard) target jest uruchamiany.

Plan wykładu

Buildsystem krok po kroku

Przykład 2

```
main: main.o hello.o
	gcc -o main main.o hello.o
```

- Reguła wprost: linkowanie
- Reguła domyślna: .c -> .o

Plan wykładu

Cele phony

- Wszystkie cele phony muszą być zależne od specjalnego targetu .PHONY
- Targety phony są to etykiety, używane aby wywołać skrypt. Nie muszą tworzyć pliku.
- Targety phony zawsze się wykonają.
- Typowe nazwy targetów phony to `all`, `install`, `clean`, `distclean`, `TAGS`, `info`, `check`

Target clean

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

Cele phony

- Wszystkie cele phony muszą być zależne od specjalnego targetu .PHONY
- Targety phony są to etykiety, używane aby wywołać skrypt. Nie muszą tworzyć pliku.
- Targety phony zawsze się wykonają.
- Typowe nazwy targetów phony to `all`, `install`, `clean`, `distclean`, `TAGS`, `info`, `check`

Target clean

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

Cele phony

- Wszystkie cele phony muszą być zależne od specjalnego targetu .PHONY
- Targety phony są to etykiety, używane aby wywołać skrypt. Nie muszą tworzyć pliku.
- Targety phony zawsze się wykonają.
- Typowe nazwy targetów phony to `all`, `install`, `clean`, `distclean`, `TAGS`, `info`, `check`

Target clean

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

Cele phony

- Wszystkie cele phony muszą być zależne od specjalnego targetu .PHONY
- Targety phony są to etykiety, używane aby wywołać skrypt. Nie muszą tworzyć pliku.
- Targety phony zawsze się wykonają.
- Typowe nazwy targetów phony to `all`, `install`, `clean`, `distclean`, `TAGS`, `info`, `check`

Target clean

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

Cele phony

- Wszystkie cele phony muszą być zależne od specjalnego targetu .PHONY
- Targety phony są to etykiety, używane aby wywołać skrypt. Nie muszą tworzyć pliku.
- Targety phony zawsze się wykonają.
- Typowe nazwy targetów phony to `all`, `install`, `clean`, `distclean`, `TAGS`, `info`, `check`

Target clean

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

Plan wykładu

Zmienne

Zmienne zwykłe

```
CC = gcc
foo.o: foo.c foo.h
	$(CC) -c foo.c
```

Zmienne automatyczne (przykładowe):

- $@ - Target
- $< - Pierwszy element listy zależności
- $^ - Cała lista zależności

Zmienne automatyczne

```
CC = gcc
foo.o: foo.c foo.h
	$(CC) -o $@ $<
```

Zmienne

Zmienne zwykłe

```
CC = gcc
foo.o: foo.c foo.h
	$(CC) -c foo.c
```

Zmienne automatyczne (przykładowe):

- `$@` - Target
- `$<` - Pierwszy element listy zależności
- `$^` - Cała lista zależności

Zmienne automatyczne

```
CC = gcc
foo.o: foo.c foo.h
	$(CC) -o $@ $<
```

Zmienne

Zmienne zwykłe

```
CC = gcc
foo.o: foo.c foo.h
	$(CC) -c foo.c
```

Zmienne automatyczne (przykładowe):

- $@ - Target
- $< - Pierwszy element listy zależności
- $^ - Cała lista zależności

Zmienne automatyczne

```
CC = gcc
foo.o: foo.c foo.h
	$(CC) -o $@ $<
```

Zmienne

Zmienne zwykłe

```
CC = gcc
foo.o: foo.c foo.h
	$(CC) -c foo.c
```

Zmienne automatyczne (przykładowe):

- $@ - Target
- $< - Pierwszy element listy zależności
- $^ - Cała lista zależności

Zmienne automatyczne

```
CC = gcc
foo.o: foo.c foo.h
	$(CC) -o $@ $<
```

Zmienne

Zmienne zwykłe

```
CC = gcc
foo.o: foo.c foo.h
	$(CC) -c foo.c
```

Zmienne automatyczne (przykładowe):

- $@ - Target
- $< - Pierwszy element listy zależności
- $^ - Cała lista zależności

Zmienne automatyczne

```
CC = gcc
foo.o: foo.c foo.h
	$(CC) -o $@ $<
```

Zmienne

Rodzaje przypisań

Zmienne rozwijane rekursywnie

```
x = $(foo)
foo = $(bar)
bar = baz          # bar = $(bar) baz ??
```

Zmienne proste

```
x := foo
y := $(x)
x := bar
```

Konkatenacja

```
x := foo bar
x += baz          # x := $(x) baz
```

Zmienne

Rodzaje przypisań

Zmienne rozwijane rekursywnie

```
x = $(foo)
foo = $(bar)
bar = baz          # bar = $(bar) baz ??
```

Zmienne proste

```
x := foo
y := $(x)
x := bar
```

Konkatenacja

```
x := foo bar
x += baz          # x := $(x) baz
```

Zmienne

Rodzaje przypisań

Zmienne rozwijane rekursywnie

```
x = $(foo)
foo = $(bar)
bar = baz          # bar = $(bar) baz ??
```

Zmienne proste

```
x := foo
y := $(x)
x := bar
```

Konkatenacja

```
x := foo bar
x += baz          # x := $(x) baz
```

Zmienne

- Podczas odwoływania się do zmiennych mogą być używane zarówno nawiasy () jak i {}, lecz według konwencji używany ()
- Podczas odwoływania się do zmiennych jednoliterowych można pominąć nawiasy
- Zmienne mogą zachowywać się jak listy, używane tutaj są makra `$(wordlist`, `$(word`, `$(words`, `$(firstword`, `$(lastword`
- Wartość zmiennej można przesyłać do Make'e podczas wywołania, przykład:
  `make FOO=bar`
- Make posiada bezpośredni dostęp do zmiennych środowiskowych

Zmienne

- Podczas odwoływania się do zmiennych mogą być używane zarówno nawiasy () jak i {}, lecz według konwencji używany ()
- Podczas odwoływania się do zmiennych jednoliterowych można pominąć nawiasy
- Zmienne mogą zachowywać się jak listy, używane tutaj są makra `$(wordlist`, `$(word`, `$(words`, `$(firstword`, `$(lastword`
- Wartość zmiennej można przesyłać do Make'e podczas wywołania, przykład:
  `make FOO=bar`
- Make posiada bezpośredni dostęp do zmiennych środowiskowych

Zmienne

- Podczas odwoływania się do zmiennych mogą być używane zarówno nawiasy () jak i {}, lecz według konwencji używany ()
- Podczas odwoływania się do zmiennych jednoliterowych można pominąć nawiasy
- Zmienne mogą zachowywać się jak listy, używane tutaj są makra `$(wordlist`, `$(word`, `$(words`, `$(firstword`, `$(lastword`
- Wartość zmiennej można przesyłać do Make'e podczas wywołania, przykład:
  `make FOO=bar`
- Make posiada bezpośredni dostęp do zmiennych środowiskowych

Zmienne

- Podczas odwoływania się do zmiennych mogą być używane zarówno nawiasy () jak i {}, lecz według konwencji używany ()
- Podczas odwoływania się do zmiennych jednoliterowych można pominąć nawiasy
- Zmienne mogą zachowywać się jak listy, używane tutaj są makra `$(wordlist`, `$(word`, `$(words`, `$(firstword`, `$(lastword`
- Wartość zmiennej można przesyłać do Make'e podczas wywołania, przykład:
  `make FOO=bar`
- Make posiada bezpośredni dostęp do zmiennych środowiskowych

Zmienne

- Podczas odwoływania się do zmiennych mogą być używane zarówno nawiasy () jak i {}, lecz według konwencji używany ()
- Podczas odwoływania się do zmiennych jednoliterowych można pominąć nawiasy
- Zmienne mogą zachowywać się jak listy, używane tutaj są makra `$(wordlist`, `$(word`, `$(words`, `$(firstword`, `$(lastword`
- Wartość zmiennej można przesyłać do Make'e podczas wywołania, przykład:
  `make FOO=bar`
- Make posiada bezpośredni dostęp do zmiennych środowiskowych

Buildsystem krok po kroku

Przykład 3

```
CC=gcc
.PHONY: clean
main: main.o hello.o hello.h
	gcc -o main main.o hello.o
clean:
	rm -f main main.o hello.o
```

- Zmienna wskazująca na kompilator C
- Target clean
- Zależność od pliku nagłówkowego

Plan wykładu

Buildsystem krok po kroku - porządki w repo

- `VPATH` - ścieżka wyszukiwania zależności (przeważne plików źródłowych)
- `-I` - ścieżka wyszukiwania plików nagłówkowych w gcc

Przykład 4

```
CC=gcc
CFLAGS=-Iheaders
VPATH=src
.PHONY: clean
main: main.o hello.o headers/hello.h
	gcc -o main main.o hello.o $(CFLAGS)
clean:
	rm -f main main.o hello.o
```

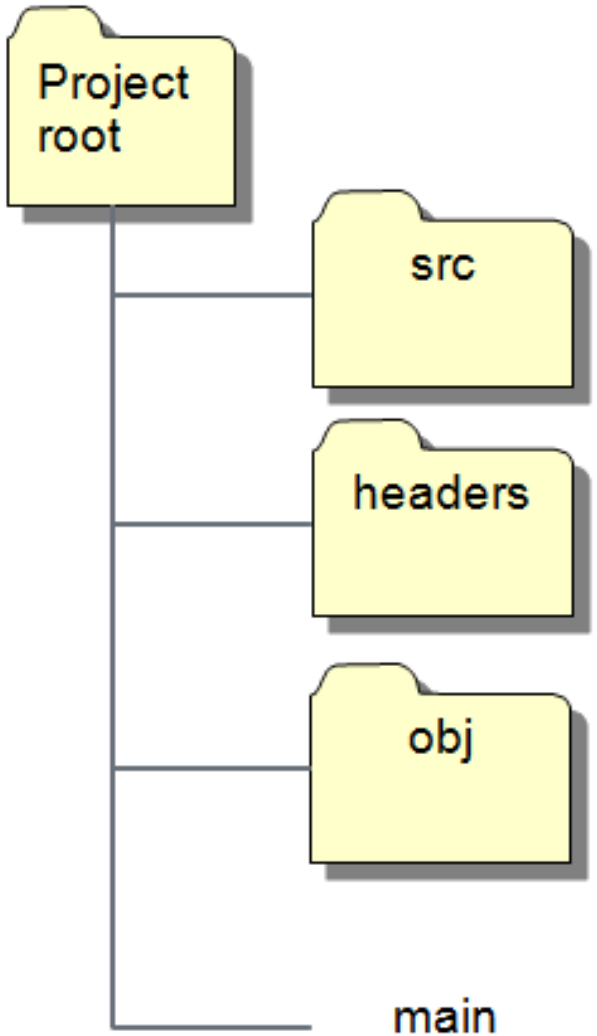
Buildsystem krok po kroku

- Osobny katalog z obiektami
- Reguła kompilacji podana wprost

Przykład 5

```
CC=gcc
CFLAGS=-Iheaders
VPATH=src
OBJDIR=obj
.PHONY: clean
$(OBJDIR)/%.o: %.c headers/hello.h
	$(CC) -c -o $@ $< $(CFLAGS)
main: main.o hello.o
	gcc -o main main.o hello.o
clean:
	rm -f main main.o hello.o
```

Docelowa struktura katalogów



Buildsystem krok po kroku

Wersja ostateczna

```
CC=gcc
VPATH=src
HDIR=headers
_H=hello.h
H=$(addprefix $(HDIR)/,$(_H))
CFLAGS=-I$(HDIR)
OBJDIR=obj
_OBJ=main.o hello.o
OBJ=$(addprefix $(OBJDIR)/,$(_OBJ))
.PHONY: clean
$(OBJDIR)/%.o: %.c $(H)
	$(CC) -c -o $@ $< $(CFLAGS)
main: $(OBJ)
	gcc -o main $(OBJ)
clean:
	rm -f main $(OBJ)
```

Plan wykładu

Wskazówki

- Wyłączajmy reguły domyślne (-r)
- -j pozwala na zrównoleglenie wywołań Make'a (zawsze przekazujmy parametr)
- Przy rekursywnym wywołaniu Make'a używajmy zmiennej $(MAKE)
- Zmienne wewnętrzne piszmy małymi literami, natomiast zmienne przekazywane z wiersza poleceń - wielmiki

Wskazówki

- Wyłączajmy reguły domyślne (-r)
- -j pozwala na zrównoleglenie wywołań Make'a (zawsze przekazujmy parametr)
- Przy rekursywnym wywołaniu Make'a używajmy zmiennej $(MAKE)
- Zmienne wewnętrzne piszmy małymi literami, natomiast zmienne przekazywane z wiersza poleceń - wielmiki

Wskazówki

- Wyłączajmy reguły domyślne (-r)
- -j pozwala na zrównoleglenie wywołań Make'a (zawsze przekazujmy parametr)
- Przy rekursywnym wywołaniu Make'a używajmy zmiennej $(MAKE)
- Zmienne wewnętrzne piszmy małymi literami, natomiast zmienne przekazywane z wiersza poleceń - wielmiki

Wskazówki

- Wyłączajmy reguły domyślne (-r)
- -j pozwala na zrównoleglenie wywołań Make'a (zawsze przekazujmy parametr)
- Przy rekursywnym wywołaniu Make'a używajmy zmiennej $(MAKE)
- Zmienne wewnętrzne piszmy małymi literami, natomiast zmienne przekazywane z wiersza poleceń - wielmiki

Wskazówki

- Używajmy standardowych nazw zmiennych CPP, CC, CXX, AR, AS, LD, CPPFLAGS, CFLAGS, CXXFLAGS, ARFLAGS, ASFLAGS, LDFLAGS, itd.
- Piszmy receptury jako jedno wywołanie powłoki
- Używajmy wbudowanych makr Make'a, a jeśli to niemożliwe używajmy prostych narzędzi cut/tr
- W większych projektach generujmy zależności od plików nagówkiwych za pomocą gcc - rodzina parametrów -M

Wskazówki

- Używajmy standardowych nazw zmiennych CPP, CC, CXX, AR, AS, LD, CPPFLAGS, CFLAGS, CXXFLAGS, ARFLAGS, ASFLAGS, LDFLAGS, itd.
- Piszmy receptury jako jedno wywołanie powłoki
- Używajmy wbudowanych makr Make'a, a jeśli to niemożliwe używajmy prostych narzędzi cut/tr
- W większych projektach generujmy zależności od plików nagówkiwych za pomocą gcc - rodzina parametrów -M

Wskazówki

- Używajmy standardowych nazw zmiennych CPP, CC, CXX, AR, AS, LD, CPPFLAGS, CFLAGS, CXXFLAGS, ARFLAGS, ASFLAGS, LDFLAGS, itd.
- Piszmy receptury jako jedno wywołanie powłoki
- Używajmy wbudowanych makr Make'a, a jeśli to niemożliwe używajmy prostych narzędzi cut/tr
- W większych projektach generujmy zależności od plików nagówkiwych za pomocą gcc - rodzina parametrów -M

Wskazówki

- Używajmy standardowych nazw zmiennych CPP, CC, CXX, AR, AS, LD, CPPFLAGS, CFLAGS, CXXFLAGS, ARFLAGS, ASFLAGS, LDFLAGS, itd.
- Piszmy receptury jako jedno wywołanie powłoki
- Używajmy wbudowanych makr Make'a, a jeśli to niemożliwe używajmy prostych narzędzi cut/tr
- W większych projektach generujmy zależności od plików nagówkiwych za pomocą gcc - rodzina parametrów -M

Plan wykładu

System budowania a CI

- Target `clean` prawie nigdy nie jest używany
- Podstawowe cele: niezawodność, powtarzalność i śledzenie błędów
- Poprawne, nienadmiarowe zależności
- Szybkie buildy przyrostowe
- Pełna automatyzacja
- Prostota utrzymania i rozszerzania

System budowania a CI

- Target `clean` prawie nigdy nie jest używany
- Podstawowe cele: niezawodność, powtarzalność i śledzenie błędów
- Poprawne, nienadmiarowe zależności
- Szybkie buildy przyrostowe
- Pełna automatyzacja
- Prostota utrzymania i rozszerzania

System budowania a CI

- Target `clean` prawie nigdy nie jest używany
- Podstawowe cele: niezawodność, powtarzalność i śledzenie błędów
- Poprawne, nienadmiarowe zależności
- Szybkie buildy przyrostowe
- Pełna automatyzacja
- Prostota utrzymania i rozszerzania

System budowania a CI

- Target `clean` prawie nigdy nie jest używany
- Podstawowe cele: niezawodność, powtarzalność i śledzenie błędów
- Poprawne, nienadmiarowe zależności
- Szybkie buildy przyrostowe
- Pełna automatyzacja
- Prostota utrzymania i rozszerzania

System budowania a CI

- Target `clean` prawie nigdy nie jest używany
- Podstawowe cele: niezawodność, powtarzalność i śledzenie błędów
- Poprawne, nienadmiarowe zależności
- Szybkie buildy przyrostowe
- Pełna automatyzacja
- Prostota utrzymania i rozszerzania

System budowania a CI

- Target `clean` prawie nigdy nie jest używany
- Podstawowe cele: niezawodność, powtarzalność i śledzenie błędów
- Poprawne, nienadmiarowe zależności
- Szybkie buildy przyrostowe
- Pełna automatyzacja
- Prostota utrzymania i rozszerzania

Dziękuję za uwagę

Pytania