Bit – najmniejsza część danych. Przyjmuje wartość 0 lub 1.

Bajt – jest to 8 bitów. Jego maksymalna wartość to 255d.

Word (Słowo) – 2 bajty, czyli 16 bitów. Maksymalna wartość to 0FFFFh (lub 65535d).

Double-Word (Podwójne słowo) – Dwa słowa, czyli 4 bajty (32 bity). Maksymalna wartość to 0FFFFFFFFh (lub 4294967295d).

Quad-Word (Poczwórne słowo) – Cztery słowa, czyli 8 bajtów (64 bity).

Kilobajt – 1024 bajty.

Megabajt – 1024 kilobajty, 1048576 bajtów.

|  |  |  |
| --- | --- | --- |
| Nazwa typu | Rozmiar w bajtach | Zakres wartości |
| **char** | 1 | -128 ÷127 |
| **int** | 2  4 w trybie 32 bit | -32768 ÷ 32767  -2147483648 ÷ 2147483647 |
| **short** | 2 | -32768 ÷ 32767 |
| **long** | 4 | -2147483648 ÷ 2147483647 |
| **unsigned char** | 1 | 0 ÷ 255 |
| **unsigned** | 2  4 w trybie 32 bit | 0 ÷ 65535  0 ÷ 4294967295 |
| **unsigned short** | 2 | 0 ÷ 65535 |
| **unsigned long** | 4 | 0 ÷ 4294967295 |
| **enum** | 2 | -32768 ÷ 32767 |
| **float** | 4 | 1,2E-38 ÷ 3,4E+38  dokładność 7 cyfr |
| **double** | 8 | 2,2E-308 ÷ 1,8E+308  dokładność 15 cyfr |
| **long double** | 10 | 3,4E-4932 ÷ 1,2E+4932  dokładność 19 cyfr |

Wykonując działania arytmetyczne należy pamiętać o ograniczeniach rejestrów

W rejestrach 8-bitowych można zapisać tylko jeden bajt, a więc liczby od 0 do 255,

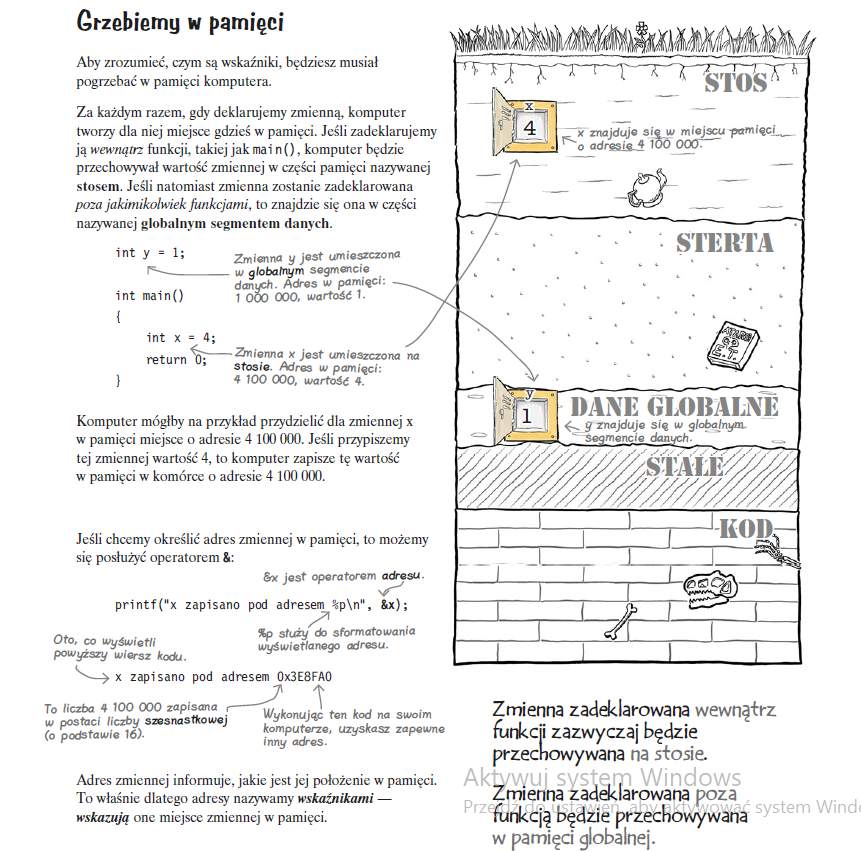
16-bitowych – od 0 do 65 535,

w 32-bitowych – od 0 do 4 294 967 295.

|  |  |  |
| --- | --- | --- |
| Instrukcja | Opis | Inne informacje |
| mov | instrukcja przeniesienia | Kopiuje zawartość jednego fragmentu pamięci do innego: mov [cel], [źródło] |
| push | instrukcja odłożenia na stosie | Kopiuje na stos zawartość wskazanego fragmentu pamięci: push [źródło] |
| pop | instrukcja pobrania ze stosu | Przenosi wartość ze stosu do wskazanego fragmentu pamięci: pop [cel] |
| add | instrukcja dodawania | Dodaje zawartość jednego fragmentu pamięci do innego: add [cel], [źródło] |
| sub | instrukcja odejmowania | Odejmuje zawartość jednego fragmentu pamięci od innego: sub [cel], [źródło] |
| xor | różnica symetryczna | Oblicza różnicę symetryczną wskazanych fragmentów pamięci: xor [cel], [źródło] |
| mul | Mnożenie | Mnoży rejestr przez podaną wartość. Mnoży liczby bez znaku. |
| div | Dzielenie | Instrukcja ta dzieli wartość w rejestrze przez wartość podaną jako dzielnik. |
| jmp | instrukcja skoku | Zmienia wartość rejestru EIP na określony adres: jmp [adres] |
| call | instrukcja wywołania | Działa podobnie jak instrukcja jmp, ale przed zmianą wartości rejestru EIP odkłada na stos adres kolejnej instrukcji: call [adres] |
| lea | instrukcja załadowania adresu | Umieszcza we wskazanym fragmencie pamięci [cel] adres innego fragmentu [źródło]: lea [cel], [źródło] |
| int | przerwanie | Przesyła określony sygnał do jądra systemu, wywołując przerwanie o określonym numerze: int [wartość] |
| INC | inkrementacja | Działa odwrotnie do instrukcji DEC, zwiększa wartość rejestru lub zmiennej o 1. |
| DEC | Dekrementacja | Bardzo prosta instrukcja. Zmniejsza podaną wartość o 1 (zmienna=zmienna-1). |
| SAL/SHL | Przesunięcie arytmetyczne/logiczne w lewo | Przesuwa w lewo bity operandu pierwszego o podaną ilość bitów w operandzie drugim. Starszy przesunięty bit, zostaje zachowany a młodsze bity są wypełniane zerami. |
| SAR/SHR | Przesunięcie arytmetyczne/logiczne w prawo | Instrukcja SAR przesuwa w prawo bity operandu pierwszego o podaną ilość bitów w operandzie drugim. Najmłodszy przesunięty bit, zostaje zachowany, a najstarszy bit zostaje zerowany. |
| cmp | porównanie |  |
| ldr | Odczyt z pamięci | odczyt danej z pamięci i zapisanie do rejestru |
| Orr.w |  |  |
| str |  | zapis danej z rejestru do pamięci |

Instrukcje dzielą się między innymi na instrukcje:

• przeniesienia (mov, push, pop),  
• arytmetyczne (add, sub, inc, neg, mul, div),  
• logiczne (and, or, xor, not),  
• sterujące (jmp, call, int, ret),  
• operujące na bitach, bajtach i łańcuchach znaków (shl, shr, rol, ror),  
• wejścia/wyjścia (in, out),  
• kontroli flag



Pokażę Ci na czym polegają różnice. Będziemy się posługiwać zapisem szesnastkowym liczb, bo ten w najprostszy sposób ukaże odmienności. Do tych odmienności dochodzi przy zapisie conajmniej dwóch bajtów.

Załóżmy, że mamy liczbę FF01. Potraktujemy ją jako liczbę bez znaku, czyli w postaci dziesiętnej będzie to 65281 (FF to 255, 01 to 1, a więc 256\*255+1). Gwoli ścisłości, dla zapisu nie ma znaczenia czy to liczba ze znakiem czy bez - znak zawsze jeśli jest, jest przechowywany w najstarszym bajcie.  
Najmniejszą jednostką informacji powodującą różnice w zapisie jest bajt (8 bitów). Skoro mamy tu dwa bajty, to ta różnica wystąpi między nimi.

|  |  |
| --- | --- |
| **little endian** | **big endian** |
| 01FF | FF01 |

Jak widzisz zapis big endian jest bardziej naturalnym zapisem, bo bajty reprezentujące liczbę zapisywane są w takiej kolejności jak my widzimy, a dokładnie od najbardziej znaczącego bajtu. W przypadku little endian jest zupełnie odwrotnie.

Jeśli liczba będzie 32 bitowa (4 bajty), np. 77AEBB50, to wtedy sytuacja wygląda tak:

|  |  |
| --- | --- |
| **little endian** | **big endian** |
| 50BBAE77 | 77AEBB50 |

czyli analogicznie do dwóch bajtów - w little endian od najmniej znaczącego bajtu.

**Jak się porozumiewać.**

Jeśli chcemy odczytywać lub zapisywać dane, musimy więć zadbać o odpowiednią kolejność bajtów, aby albo uzyskać, albo wstawić do "pamięci" poprawną liczbę. Zajmijmy się najpierw odczytem danych zakładając, że są one w zapisie little endian.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **bajt** | **przesunięcie w lewo (<<)** | **wynik cząstkowy** | **poprzednia wartość** | **wynik (OR)** |
| 50 | 0 | 50 | 0 | 50 |
| BB | 8 | BB00 | 50 | BB50 |
| AE | 16 | AE0000 | BB50 | AEBB50 |
| 77 | 24 | 77000000 | AEBB50 | **77AEBB50** |

Kiedy odczytujemy dane w zapisie big endian "pchamy" kolejny bajty w lewo.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **bajt** | **przesunięcie w prawo (>>)** | **wynik cząstkowy** | **poprzednia wartość** | **wynik (OR)** |
| 77 | 24 | 77000000 | 0 | 77000000 |
| AE | 16 | AE0000 | 77000000 | 77AE0000 |
| BB | 8 | BB00 | 77AE0000 | 77AEBB00 |
| 50 | 0 | 50 | 77AEBB00 | **77AEBB50** |

Przy zapisie danych postępujemy analogicznie odwrotnie. Czyli dla little endian:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **liczba** | **maska** | **wynik cząstkowy** | **przesunięcie w prawo (>>)** | **bajt** |
| 77AEBB50 | 000000FF | 50 | 0 | **50** |
| 77AEBB50 | 0000FF00 | BB00 | 8 | **BB** |
| 77AEBB50 | 00FF0000 | AE0000 | 16 | **AE** |
| 77AEBB50 | FF000000 | 77000000 | 24 | **77** |

Możemy też postąpić trochę inaczej:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **aktualna wartość** | **maska (AND)** | **bajt** | **przesunięcie w prawo (>>)** | **pozostała wartość** |
| 77AEBB50 | 000000FF | **50** | 8 | 0077AEBB |
| 0077AEBB | 000000FF | **BB** | 8 | 000077AE |
| 000077AE | 000000FF | **AE** | 8 | 00000077 |
| 00000077 | 000000FF | **77** | - | - |

Dla big endian trzeba zrobić tak:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **liczba** | **maska (AND)** | **wynik cząstkowy** | **przesunięcie w prawo (>>)** | **bajt** |
| 77AEBB50 | FF000000 | 77000000 | 24 | **77** |
| 77AEBB50 | 00FF0000 | 00AE0000 | 16 | **AE** |
| 77AEBB50 | 0000FF00 | 0000BB00 | 8 | **BB** |
| 77AEBB50 | 000000FF | 00000050 | 0 | **50** |

Te przykłady dotyczą sytuacji kiedy zapisując wynikowe bajty możesz zwiększać indeks tablicy o 1. Jeśli jednak masz możliwość manipulowania indeksem tablicy, to przy zapisie big endian możesz wykorzystać metodę zapisu little endian, ale wstawiając kolejne bajty pod indeksy 3, 2, 1, 0, a nie 0, 1, 2, 3. Albo wykorzystać metodę dla big endian przy zapisie little endian, również zmieniając kolejność wstawiania bajtów.

**Kod pokoju.**

Skoro już wiesz jak przechowywane są dane, to zanim zajmiemy się kodem do konwersji, pokażę Ci jak sprawdzić jaki format zapisu stosuje platforma, na której pracujesz. Wykorzystamy tu typowane tablice (o nich w przyszłym wpisie), a dokładnie Uint16Array oraz Uint8Array. Zapiszemy daną 16-bitową (00 01), a następnie odczytamy bajt spod indeksu 0. Jeśli to najmniej znaczący bajt liczby, to pracujesz na platformie obsługującej little endian (zapis w typowanej tablicy "uwalnia" nas od dbania o sposób zapisu, bo ten jest uzależniony od komputera, na którym jest uruchamiany kod).

const isLittleEndian = new Uint8Array(new Uint16Array([1]).buffer)[0] === 1;

No dobra, kod. Konwersja przy odczycie:

|  |  |
| --- | --- |
|  | /\* z little endian \*/ |
|  | let bytes = [0x50, 0xBB, 0xAE, 0x77]; |
|  | let value = ((((bytes[3] << 8) | bytes[2]) << 8) | bytes[1]) << 8 | bytes[0]; |
|  |  |
|  | /\* z big endian \*/ |
|  | let bytes = [0x77, 0xAE, 0xBB, 0x50]; |
|  | let value = ((((bytes[0] << 8) | bytes[1]) << 8) | bytes[2]) << 8 | bytes[3]; |

Konwersja przy zapisie:

|  |  |
| --- | --- |
|  | /\* na little endian \*/ |
|  | let value = 0x77AEBB50; |
|  | let bytes = []; |
|  | bytes[0] = value & 0xFF; |
|  | value >>= 8; |
|  | bytes[1] = value & 0xFF; |
|  | value >>= 8; |
|  | bytes[2] = value & 0xFF; |
|  | value >>= 8; |
|  | bytes[3] = value & 0xFF; |
|  |  |
|  | /\* na big endian \*/ |
|  | let value = 0x77AEBB50; |
|  | let bytes = []; |
|  | bytes[0] = (value & 0xFF000000) >> 24; |
|  | bytes[1] = (value & 0x00FF0000) >> 16; |
|  | bytes[2] = (value & 0x0000FF00) >> 8; |
|  | bytes[3] = (value & 0x000000FF); |

I odtąd plemiona pozostawały w przyjaźni...  
Tak jak obiecałem w jakimś przyszłym wpisie opowiem o typowanych tablicach oraz pogadamy o... słowach.

.**Makefile**

Programy typu MAKE służą do automatyzacji budowania dużych i małych projektów.

Nazwa zadania zaczyna się w pierwszej kolumnie, kończy

dwukropkiem. Po dwukropku są podane nazwy zadań (lub plików) , od wykonania których zależy wykonanie

tego zadania. W kolejnych wierszach są komendy służące do wykonania danego zadania.

*UWAGA:* komendy NIE MOGA zaczynać się od pierwszej kolumny! Należy je pisać je po jednym

tabulatorze (ale nie wolno zamiast tabulatora stawiać ośmiu spacji).

Aby wykonać dane zadanie, wydajemy komendę make nazwa\_zadania. Jeśli nie podamy nazwy zadania

(co jest często spotykane), wykonywane jest zadanie o nazwie all (wszystko).

all: kompilacja linkowanie

echo "Wszystko zakonczone pomyslnie"

kompilacja:

nasm -O999 -f elf -o plik1.o plik1.asm

nasm -O999 -f elf -o plik2.o plik2.asm

nasm -O999 -f elf -o plik3.o plik3.asm

fasm plik4.asm plik4.o

fasm plik5.asm plik5.o

fasm plik6.asm plik6.o

linkowanie: plik1.o plik2.o plik3.o plik4.o plik5.o plik6.o

ld -s -o wynik plik1.o plik2.o plik3.o plik4.o \

plik5.o plik6.o

help:

echo "Wpisz make bez argumentow"

Ale MAKE jest mądrzejszy, niż może się to wydawać!

Mianowicie: jeśli stwierdzi, ze wynik został stworzony PÓZNIEJ niż pliki .o podane w linii zależności, to nie wykona bloku linkowanie, bo nie ma to sensu skoro program wynikowy i tak jest aktualny. MAKE robi tylko to, co trzeba. Oczywiście, niezależnie od wieku plików .o , dział kompilacja i tak zostanie wykonany (bo nie ma zależności, wiec MAKE nie będzie sprawdzał wieku plików).

Znak odwrotnego ukośnika \ powoduje zrozumienie, ze następna linia jest kontynuacja bieżącej, znak

krzyżyka # powoduje traktowanie reszty linijki jako komentarza.

Jeśli w czasie wykonywanie któregokolwiek z poleceń w bloku wystąpi błąd (ścisłe mówiąc, to gdy błąd zwróci wykonywane polecenie, jak u nas FASM czy NASM), to MAKE *natychmiast przerywa działanie* z informacja o błędzie i nie wykona żadnych dalszych poleceń (pamiętajcie wiec o umieszczeniu w zmiennej środowiskowej PATH ścieżki do kompilatorów).

W powyższym pliku widać jeszcze jedno: zmiana nazwy któregoś z plików lub jakieś opcji sprawi, ze trzeba ja będzie zmieniać wielokrotnie, w wielu miejscach pliku. Bardzo niewygodne w utrzymaniu, prawda? Na szczęście z pomocą przychodzą nam ... zmienne, które możemy deklarować w Makefile i które zrozumie program MAKE.

Składnia deklaracji zmiennej jest wyjątkowo prosta i wygląda tak:

NAZWA\_ZMIENNEJ = wartość

A użycie:

$(NAZWA\_ZMIENNEJ)

Polecam nazwy zmiennych pisać wielkimi literami w celu odróżnienia ich od innych elementów. Pole

wartości zmiennej może zawierać dowolny ciąg znaków.

Jeśli chcemy, aby treść polecenia NIE pojawiała się na ekranie, do nazwy tego polecenia dopisujemy z przodu

znak małpki @, np.

@echo "Wszystko zakończone pomyślnie"

Uzbrojeni w te informacje, przepisujemy nasz wcześniejszy Makefile:

FASM = fasm # ale mozna tu w przyszłosci wpisac pełna sciezke

NASM = nasm

NASM\_OPCJE = -O999 -f elf

LD = ld

LD\_OPCJE = -s

PLIKI\_O = plik1.o plik2.o plik3.o plik4.o plik5.o plik6.o

PROGRAM = wynik

all: kompilacja linkowanie

@echo "Wszystko zakonczone pomyslnie"

kompilacja:

$(NASM) $(NASM\_OPCJE) -o plik1.o plik1.asm

$(NASM) $(NASM\_OPCJE) -o plik2.o plik2.asm

$(NASM) $(NASM\_OPCJE) -o plik3.o plik3.asm

$(FASM) plik4.asm plik4.o

$(FASM) plik5.asm plik5.o

$(FASM) plik6.asm plik6.o

linkowanie: $(PLIKI\_O)

$(LD) $(LD\_OPCJE) -o $(PROGRAM) $(PLIKI\_O)

help:

@echo "Wpisz make bez argumentów"

Oczywiście, w końcowym Makefile należy napisać takie regułki, które pozwolą na ewentualna kompilacje

pojedynczych plików, np.

plik1.o: plik1.asm plik1.inc

$(NASM) $(NASM\_OPCJE) -o plik1.o plik1.asm

Program make jest częścią projektu GNU. Został stworzony na potrzeby automatyzacji procesu kompilacji oraz budowania projektu. Jest on interpreterem plików nazywanych Makefile’ami, w których możemy tworzyć własne reguły kompilacji w taki sposób, aby można było za pomocą dwóch-trzech komend zamienić kod źródłowy w pliki wykonywalne, biblioteki itp.

Spójrzmy na przykładowy plik Makefile:

# Makefile

**main:**  
 g++ main.cpp -o main

Mamy tutaj najprostszą postać Makefile: regułę main, oraz polecenia w niej zawarte. Pozwala to na uruchomienie komendy g++ main.cpp -o main za pomocą polecenia make. Jedyny warunek, który musi być spełniony, to ten, że musimy znajdować się w tym samym katalogu co nasz plik Makefile. Wewnątrz pliku Makefile możemy zawrzeć więcej, niż jedną regułę. Przykład:

Należy pamiętać, że jeżeli nie podamy nazwy reguły która nas interesuje, domyśnie uruchomią się polecenia z pierwszej reguły w pliku. W naszym wypadku jest to reguła main.

Zależności pomiędzy regułami

**Bardzo ważną zaletą**, której nie można pominąć, jest możliwość ustalania **zależności pomiędzy regułami**. W skrócie: jedna regułą może zależeć od drugiej. Lepiej - **jedna reguła może zależeć od wielu reguł**. Daje to spore możliwości - dzięki temu możemy na przykład budować skomplikowane procesy, takie jak np. wyczyść-skompiluj-przetestuj-zainstaluj. Możemy przygotować cztery reguły, które później zgrupujemy w jedną. Poniżej prosty przykład, z jedną zależnością:

# Makefile

main: clean

g++ main.cpp -o main

clean:

rm -f main

W powyższym przykładzie możemy zauważyć, że wykonując polecenie make najpierw zostaną uruchomione się polecenia spod reguły clean, a następnie zostaną wykonane się polecenia zawarte w regule main. Ten sam efekt uzyskamy, wpisując w terminalu komendę make main. Jeżeli chcemy, aby make main odpalało dodatkowo instrukcje spod więcej niż jedną reguły, to robi się to tak:

# Makefile

main: clean debug

g++ main.cpp -o main

clean:

rm -f main

debug:

echo "Compiling project"

Domyślne wartości parametrów

Jako programiści musimy zawsze dbać o to, aby program działał **zawsze** dobrze. Skoro dajemy użytkownikowi możliwość skonfigurowania sobie skryptu instalacyjnego “po swojemu”, to musimy zadbać o to, co się stanie w sytuacji kiedy nie zostaną podane żadne wartości parametrów w momencie uruchomienia programu. Domyślną wartość parametru ustawiamy używając operatora ?=:

# Makefile

CONFIG\_DIR?="/etc/myprogram"

configure:

echo "${CONFIG\_DIR}"

Operator ?= różni się od operatora = tym, że ustawia wartość jeżeli nie została wcześniej ustawiona, bądź została ustawiona wartość pusta. Zatem, jeżeli nie chcemy pozwolić na zewnątrz sterować zmienną, ale chcemy aby była ona dostępna w programie - używamy operatora =.

Ktoś pomyśli teraz: “Ok, ale ja chcę mieć większą kontrolę - przecież chcę wyświetlać ważniejsze instrukcje, aby użytkownik wiedział co się dzieje”. Do pomocy przychodzi operator @. Jeżeli postawimy go przed instrukcją, zostanie ona wytłumiona. Na przykład:

# Makefile

main:

@echo "Compiling..."

g++ main.cpp -o main

# Przykład ignorowania wartości z zewnątrz

P="$(PWD)"

# Przykład ustawiania wartości domyślnej, jeżeli nieustawiono

COMPILE\_FILENAME?="main"

# Domyślna reguła

info:

echo "> Current directory: $P"

echo "> Compile file: ${COMPILE\_FILENAME}"

# Przykład zależności od kilku reguł

production: info clean

echo "> Compiling production code for ${COMPILE\_FILENAME}..."

g++ ${COMPILE\_FILENAME}.cpp -o ${COMPILE\_FILENAME}

echo " Compiled"

# Przykład zależności od kilku reguł

dump: info clean

echo "> Dumping objects and assembly"

g++ ${COMPILE\_FILENAME}.cpp -S

g++ ${COMPILE\_FILENAME}.cpp -c -o ${COMPILE\_FILENAME}.o

# Przykład wytłumienia pojedynczej instrukcji

@touch dump

echo " Dumped"

# Przykład wyciszenia konkretnych instrukcji

clean:

echo "> Cleaning previous compilation"

@if [[ -e "$P/dump" ]]; then \

echo " Cleaning dumped objects"; \

rm -f ${COMPILE\_FILENAME}.s ${COMPILE\_FILENAME}.o dump; \

fi

@if [[ -e "$P/$COMPILE\_FILENAME" ]]; then \

echo " Cleaning executables"; \

rm -f ${COMPILE\_FILENAME}; \

fi

echo " Cleaned"

# Wyciszenie domyślnego wyjścia

.SILENT:  
  
Prawidłowa nomenklatura

Na samym początku należy wyjaśnić kilka drobnych pojęć:

* **Reguła** (*ang. Rule*) - zbiór instrukcji tworzących jedną konkretną operację
* **Warunki wstępne** (*ang. Prerequisites*) - zbiór reguł, od których potrzeby wykonania zależy wykonanie reguły zależnej
* **Instrukcja** (*ang. Command*) - komenda zawarta wewnątrz reguły
* **Cel** (*ang. Target*) - plik wynikowy generowany wewnątrz reguły
* **Cel główny** (*ang. Goal*) - pierwsza reguła w pliku Makefile, uruchamiana za pomocą polecenia make

Równoważnikiem powyższego jest następujący (pseudo) kod:

# Makefile

Goal:

Command;

Command;

Rule: Prerequsites

Command;

g++ main.cpp -o Target;

Rekompilacja dużych projektów - wildcards

Prezentowane przeze mnie przykłady są bardzo proste i jest to zabieg celowy. Przykładowe fragmenty mają przekazać jedynie istotę omawianego tematu, nie rozpraszając uwagi czytelników na niepotrzebnym szumie informacyjnym. Mam nadzieję, że każdej osobie czytającej ten post przyjdzie do głowy pytanie o zastosowanie plików Makefile w większych projektach, składających się z dziesiątek, a niejednokrotnie i setek plików źródłowych. Dla wszystkich tych osób mam dobrą wiadomość: nie ma konieczności modyfikowania plików Makefile przy każdorazowym tworzeniu/usuwaniu plików źródłowych w projekcie. Istnieje bowiem mechanizm wildcards, dzięki któremu wyrażając jedną ścieżkę możemy uchwycić więcej niż jeden plik. O ile ścieżka src/main.cpp prowadzi jedynie do jednego pliku, to ścieżka:

src/\*.cpp

zwróci nam listę wszystkich plików o rozszerzeniu .cpp istniejących wewnątrz katalogu src/ (symbol \* oznacza dowolny ciąg znakowy). W miejsce powyższego odwołania wstawione zostaną poniższe (przykładowe oczywiście) ścieżki:

# Makefile

SOURCES=$(wildcard src/\*.cpp)

OBJECTS=$(patsubst %.cpp, %.o, $(SOURCES))

**main: $(OBJECTS)**

g++ $^ -o $@

$(OBJECTS): src/%.o : src/%.cpp

g++ -c $< -o $@

Tutaj również pojawiło się kilka nowych konstrukcji, które wymagają wyjaśnienia. Otóż, pojawiła się konstrukcja OBJECTS=$(patsubst %.cpp, %.o, $(SOURCES)). Tworzymy zmienną OBJECTS, która wypełniona zostaje wartością zwróconą przez funkcję patsubst. Czym jest funkcja patsubst ? Jest to funkcja zamieniająca wystąpienia określonego fragmentu wewnątrz ciągu znakowego innym fragmentem. Funkcja ta korzysta z mechanizmu *wildcards* (znak %). Powyższe oznacza, że na podstawie wartości zmiennej SOURCES budujemy zmienną OBJECTS, przy czym ciągi .cpp zostają zamienione na .o. Zakładając, że wszystkie pliki wewnątrz katalogu src składają się na target, oraz każdy plik źródłowy ma swój odpowiedni plik obiektowy, możemy utworzyć regułę dla całej dynamicznie wygenerowanej listy plików obiektowych. Aby zrozumieć pozostałą część powyższego kodu, należy przedstawić czym są reguły o wielu targetach.

Reguły o wielu targetach

Specjalnym (bardzo ułatwiającym życie) rodzajem reguł są reguły o wielu targetach. Są to reguły, które potrafią przyjąć listę wielu targetów, na podstawie konkretnych wzorców odczytać target oraz warunki wstępne, po czym wygenerować plik targetu. Przykładowy kod:

# Makefile

**main: src/code.o src/clean.o**

g++ src/code.o src/clean.o -o main

src/code.o src/clean.o: %.o: %.cpp

g++ -c $< -o $@

Tutaj przyjrzeć należy się ostatniej regule. Pierwsza część jej opisu zawiera listę targetów. Druga część to wzorzec dopasowywany do listy targetów w celu wyłuskania każdego kolejnego targetu z listy. Na podstawie dopasowanego wzorca generowany jest ciąg nazywany Stem (znak %), z którego korzystają druga oraz trzecia część reguły. Trzecia część reguły nazywana jest wzorcem warunków wstępnych. Dla powyższego przykładu jako Stem kolejno zostaną odnalezione src/code oraz src/clean, które zostają użyte wewnątrz instrukcji tej reguły. Brzmi to nieco skomplikowanie, ale wystarczy kilka minut testów, aby zrozumieć. Tajemnicze znaczki $< oraz $@ są zmiennymi automatycznymi, o których przeczytać można w dalszej części wpisu. Teraz, bogaci o nową wiedzę, możecie z powrotem wrócić do wcześniejszego przykładu aby móc przeanalizować już całość kodu.

## Zmienne pamiętające kontekst

Pliki Makefile możemy definiować w sposób bardzo zautomatyzowany, o czym przekonaliśmy się w dzisiejszym wpisie. Aby nie zgubić się wewnątrz dynamicznie tworzonych reguł, możemy użyć specjalnego zestawu zmiennych, które pamiętają charakterystyczne informacje na temat aktualnie wykonywanej reguły. Oto ich lista:

* $@ - nazwa pliku targetu w aktualnie uruchomionej regule
* $< - nazwa pierwszego warunku wstępnego
* $^ - lista wszystkich warunków wstępnych (zawiera ewentualne duplikaty)
* $+ - lista wszystkich warunków wstępnych (bez duplikatów)
* $? - lista wszystkich warunków wstępnych, które są nowsze niż target

Instrukcje warunkowe

Programując w C++ można spotkać się z wieloma problemami związanymi z przenośnością kodu. Czasami dla każdego rodzaju systemu operacyjnego trzeba napisać na nowo cały komponent, który następnie trzeba skompilować. Problem, z jakim się spotkamy to potrzeba automatycznej detekcji, które pliki źródłowe powinny zostać skompilowane na bieżącej maszynie? Przecież nie możemy kompilować pod Linuxem plików zawierających #include <windows.h>… Z pomocą przychodzi nam zestaw instrukcji warunkowych.

Wewnątrz plików Makefile możemy używać czterech bardzo zbliżonych do siebie instrukcji warunkowych:

* ifeq - sprawdza, czy dwie wartości są sobie równe
* ifneq - sprawdza, czy dwie wartości są od siebie różne
* ifdef - sprawdza, czy zmienna posiada niepustą wartość
* ifndef - sprawdza, czy zmienna posiada pustą wartość

Schemat budowy każdego z wyżej wymienionych typów instrukcji warunkowych wygląda następująco:

conditional-directive

text-if-true

**else**

text-if-false

**endif**

Ważnym jest, aby mieć świadomość, że warunki które dostarcza nam program make działają na poziomie plików Makefile, a nie instrukcji shellowych zawartych wewnątrz definiowanych reguł. Oznacza to, że jeżeli warunek nie wskoczy, to program make nie widzi tego, co zostaje wewnątrz niego. Ponieważ zmiennych definiowanych wewnątrz plików Makefile nie można modyfikować, powoduje to, że warunki działają na zasadzie preprocesora Makefie. Jeżeli przychodzi potrzeba użyć instrukcji warunkowych zależnych od aktualnego punktu wykonania programu, należy użyć instrukcji warunkowych shellowych.

Składnia pliku makefile:

# komentarz...

cel : zależność1 zależność2 zależność3 ... zależnośćN

komendy ...

komendy ...

...

### **Reguły suffixowe, czyli automatyka v2.0**

Reguły suffixowe (ang. suffix – przyrostek) opisują sposób wytworzenia dowolnego celu mającego w nazwie określony przyrostek. Takim przyrostkiem jest zazwyczaj rozszerzenie pliku, więc w efekcie reguły takie opisują sposób wytwarzania plików o danym rozszerzeniu.

Użycie reguły suffixowej obrazuje poniższy przykład:

program: file1.o file2.o file3.o

gcc -o $@ $^

%.o: %.c

gcc -c -o $@ $^

Zasada działania takiej reguły jest prosta – jeżeli potrzebny jest określony plik z rozszerzeniem “.o”, to jego zależnością staje się taki sam plik z rozszerzeniem “.c”. Do wytworzenia celu zostanie użyta receptura znana z poprzedniego przykładu.

Dzięki regułom suffixowym można łatwo uniknąć wielokrotnego zapisywania reguł różniących się tylko nazwą celu i zależności. Są one w związku z tym bardzo powszechnie używane w plikach Makefile.

### **Cel specjalny .PHONY**

Tak jak pisałem wcześniej, program **make** za każdym razem stara się skojarzyć nazwę celu z nazwą pliku. W związku z tym także podczas budowania celu *all* sprawdzi on, czy w katalogu nie istnieje już plik o takiej nazwie. Na ogół plik o nazwie *all* nie będzie istniał ani też nie będzie tworzony przez recepturę celu *all*, więc będzie on przebudowywany za każdym wywołaniem komendy **make all**. Sytuacja jednak zmieni się, jeśli w którymś momencie w katalogu ze źródłami pojawi się plik o nazwie *all*. Dla wielu niedoświadczonych użytkowników zachowanie to może być zaskakujące, a szukanie źródła problemu może zająć sporo czasu.

Na szczęście istnieje metoda pozwalająca zapobiec takim sytuacjom, poprzez oznaczenie wybranych celów jako “fałszywe”. Służy do tego cel specjalny **.PHONY**. Jest on traktowany przez program **make** w szczególny sposób, ponieważ nie odpowiada on za wytworzenie żadnego produktu, a jego jedyną rolą jest możliwość określenia listy celów, które nie powinny być kojarzone z nazwami plików. Przykład jego użycia możesz zobaczyć poniżej:

all: program

clean:

rm program file1.o file2.o file3.o

.PHONY: all clean

program: file1.o file2.o file3.o

gcc -o $@ $^

%.o: %.c

gcc -c -o $@ $^

Jako zależności celu **.PHONY** można podać listę celów, które powinny być traktowane jako “fałszywe” (czyli niepowiązane z plikami), co raz na zawsze rozwiązuje problem. Cel **.PHONY**, ze względu na powszechność użycia celów *all* i *clean*, można znaleźć w plikach Makefile większości projektów.

Wywołanie funkcji Makefile ma następującą składnię:

$(nazwa\_funkcji argument1, argument2, ... , argumentN)

Zapis ten może nieco przypominać odwołanie do zmiennej. W nawiasach jako pierwsza występuje zawsze nazwa funkcji, a po niej znajduje się lista argumentów oddzielonych przecinkami. Wynik zwracany przez funkcję może być przypisany do zmiennej, przekazany jako argument do innej funkcji, albo użyty jako dowolny element reguły Makefile.

Jako przykład weźmy sobie funkcję **patsubst**, pozwalającą na łatwą zamianę suffixów – jej działanie jest bardzo zbliżone do tego, co można osiągnąć przy użyciu reguły suffixowej. W praktyce funkcja **patsubst** i reguły suffixowe dobrze się uzupełniają i są często używane razem. Przykład wywołania tej funkcji możesz zobaczyć poniżej:

$(patsubst %.c, %.o, file1.c file2.c file3.c)

W efekcie funkcja zwróci listę plików z rozszerzeniem zmienionym z “.c” na “.o”. Wywołanie to można wykorzystać do wygenerowania listy obiektów pośrednich na podstawie listy plików źródłowych. Można więc zmodyfikować nasz przykładowy plik Makefile w następujący sposób:

Funkcja **shell**, bo o niej mowa, pozwala na wywołanie z poziomu pliku Makefile dowolnego polecenia powłoki. Może to być prosta komenda, skrypt lub dowolny program. Nie trudno zauważyć, że otwiera to ogromne możliwości.

W jaki sposób możemy usprawnić nasz przykładowy plik Makefile z użyciem tej funkcji? Zapewne na wiele sposobów, ale jednym z najfajniejszych będzie pozbycie się konieczności podawania listy plików z kodem źródłowym. Dzięki temu będzie można rozbudowywać projekt o kolejne pliki bez konieczności modyfikowania pliku Makefile. Do automatycznego wygenerowania listy plików z kodem użyjemy polecenia **ls \*.c**. Zmodyfikowany plik Makefile możesz zobaczyć poniżej:

### **Problem zależności od nagłówków**

W kwestii minimalizacji pliku Makefile doszliśmy już do poziomu expert, pozostała nam jednak jeszcze jedna nierozwiązana kwestia. Jest to jeden z najczęściej pomijanych podczas pisania pliku Makefile problemów, który jednak niejednemu programiście spędził sen z powiek – obsługa zależności od nagłówków.

Zazwyczaj w plikach Makefile hierarchia zależności jest bardzo prosta – wynikowy plik wykonywalny zależy od plików “.o”, a te z kolei zależą od plików “.c” i na tym sprawa się kończy. Niestety programistyczna rzeczywistość jest nieco bardziej złożona – pliki “.c” załączają zazwyczaj nagłówki “.h“, zawierające różnego rodzaju deklaracje, makrodefinicje i funkcje inline. W trakcie kompilacji o załączenie odpowiednich plików nagłówkowych troszczy się kompilator, dlatego są one często pomijane podczas tworzenia pliku Makefile. Przez długi czas to zaniedbanie może zostać niezauważone, ponieważ wszystko będzie działać jak należy, jednak problem może w końcu pojawić się w najmniej oczekiwanym momencie.

Taki moment ma zazwyczaj miejsce kiedy dokonamy modyfikacji dotykającej wyłącznie plik nagłówkowy – np. zmienimy makrodefinicję lub zmodyfikujemy funkcję inline. W takiej sytuacji wywołanie komendy **make all** zakończy się komunikatem, że nie ma nic do wybudowania, bo cel jest nowszy niż wszystkie jego zależności. Pierwszą reakcją jest wtedy zazwyczaj zaskoczenie – przecież przed chwilą zmieniliśmy kod! Po krótszych lub dłuższych poszukiwaniach prawda wychodzi jednak w końcu na jaw – pliku, który zmodyfikowaliśmy nie ma na żadnej liście zależności.

Naiwnie proste rozwiązanie problemu poprzez dopisanie plików nagłówkowych jako zależności głównego celu bardzo szybko okazuje się nieskuteczne. Wymusi to co prawda ponowne zliknowanie programu wynikowego, jednak nie spowoduje przebudowania obiektów stworzonych z plików z kodem wykorzystujących zmieniony nagłówek. Potrzebujemy więc dodania plików nagłówkowych jako zależności przy kompilacji poszczególnych obiektów.

Rodzi to jednak pewien problem – musimy wiedzieć, które pliki nagłówkowe załączane są przez poszczególne pliki “.c” i musimy tą wiedzę zapisać w postaci reguł w pliku Makefile. Generuje to potrzebę wytworzenia dużej ilości trudnych w utrzymaniu reguł – za każdym razem gdy dodajemy lub usuwamy załączany nagłówek w pliku “.c” musimy także uaktualnić plik Makefile. Dodatkowo sprawa komplikuje się jeszcze bardziej, gdy uwzględnimy fakt, że jedne pliki nagłówkowe mogą także includować inne pliki nagłówkowe. Nasz system budowania przeradza się nagle z krótkiej listy prostych reguł w istne nagłówkowe piekło.

### **Zależności od nagłówków – rozwiązanie**

Na szczęście istnieje prosty sposób na automatyczne wygenerowanie listy zależności od nagłówków, którą można następnie dołączyć do pliku Makefile. Potrafi to dla nas zrobić sam kompilator. Wystarczy do opcji kompilatora dodać flagi **-MMD** oraz **-MP** – pierwsza z nich mówi, że podczas kompilacji oprócz pliku “.o” powinien być także wygenerowany plik “.d” zawierający zależności do plików nagłówkowych (z pominięciem nagłówków systemowych). Druga z podanych opcji powoduje, że dla każdego pliku nagłówkowego zostaną utworzone reguły zapobiegające błędom w przypadku celowego usunięcia jednego z plików – naprawia to znaną przypadłość programu **make**, który domyślnie próbuje w takiej sytuacji znaleźć regułę tworzącą brakujący plik nagłówkowy, co na ogół skutkuje pojawieniem się niespodziewanych błędów.

Po dodaniu powyższych flag kompilacji wystarczy w pliku Makefile załączyć wygenerowane zależności używając polecenia –**include**. Nasz plik Makefile po dokonaniu zmian wyglądać będzie następująco:

PROGRAM = program

SRC = $(shell ls \*.c)

OBJS = $(patsubst %.c, %.o, $(SRC))

DEP = $(patsubst %.c, %.d, $(SRC))

CC = gcc

CFLAGS = -g -Wall -MMD -MP

LDFLAGS = -lm

all: $(PROGRAM)

clean:

rm $(PROGRAM) $(OBJS)

.PHONY: all clean

$(PROGRAM): $(OBJS)

-include $(DEP)

Jak widać nie potrzeba było zbyt wielu zmian. Pojawiła się nowa zmienna **DEP**, przechowująca listę plików zawierających zależności dla poszczególnych obiektów. Wygenerowana została ona w taki sam sposób jak lista obiektów – z użyciem funkcji **patsubst**. Nieco niżej na liście opcji w zmiennej **CFLAGS** pojawiły się dwie wspomniane wcześniej flagi **-MMD** i **-MP**, a na samym końcu, przy pomocy komendy **-include,** do pliku Makefile dołączane są wszystkie wygenerowane pliki z zależnościami.

Warto zwrócić uwagę na znak minusa poprzedzający komendę **include**. Oznacza on tyle, że jeżeli któregoś z plików z zależnościami zabraknie, to kompilacja i tak nie powinna zakończyć się błędem – jest to istotne podczas pierwszej kompilacji projektu, kiedy pliki z zależnościami nie są jeszcze wygenerowane.

W taki oto łatwy sposób udało nam się poradzić sobie z powszechnie występującym problemem zależności od plików nagłówkowych. Liczę na to, że prostota tego rozwiązania skłoni Cię do jego użycia w Twoich projektach – widziałem już zdecydowanie zbyt wiele plików Makefile, w których tego brakowało.

### **Przysłanianie zmiennych: -Wshadow**

Kompilator ostrzeże nas, gdy zmienna zostanie przysłonięta przez zmienną o tej samej nazwie:

**int** **main**() {

**int** i {0};

{

**int** i{1}; // warning: declaration of 'i' shadows a previous local

}

**return** i;

}  
  
**Brak wirtualnego destruktora: -Wnon-virtual-dtor**

Poinformowani zostaniemy, jeśli jakaś klasa posiada metodę wirtualną i nie dostarcza wirtualnego destruktora:

class Foo { // 'class Foo' has virtual functions and accessible non-virtual destructor

virtual void foo() {}

};

int main() {}  
  
**Nieużywane identyfikatory: -Wunused**

Flaga ta ostrzega o wszystkim, co nie jest używane w naszym kodzie. Implikuje ona flagi: -Wunused-function``-Wunused-label -Wunused-value -Wunused-variable

**int** **main**(**void**) {

**int** i{}; // unused variable 'i'

}

Częstym sposobem “pozbycia się” warninga -Wunused-variable jest zdefiniowanie makra:

#**define** UNUSED(x) (void)(x)

**int** **main**(**void**) {

**int** i{};

UNUSED(i);

}  
**Rzutowanie na niepoprawny typ: -Wsign-conversion**

Informuje nas, jeśli będziemy próbowali rzutować zmienną bez znaku na zmienną ze znakiem lub odwrotnie:

int main() {

unsigned int j = -1; // unsigned conversion from 'int' to 'unsigned int' changes value from '-1' to '4294967295'

}  
**Nieprawidłowe wcięcia: -Wmisleading-indentation**

Ostrzega nas, gdy wcięcia mogą powodować złe zrozumienie kodu:

#include <iostream>

int main(int argc, char\* argv[]) {

int i = argc;

if (i == 0) // this 'if' clause does not guard...

i = 0;

std::cout << "cpp-polska rlz\n"; // ...this statement, but the latter is misleadingly indented as if it were guarded by the 'if'

}  
  
**Powielone warunki: -Wduplicated-cond**

Dostaniemy ostrzeżenie, gdy w warunku if/else powtórzy się ten sam warunek:

int main(int argc, char\* argv[]) {

if (argc == 0) {} // previously used here

else if (argc == 0) {} // duplicated 'if' condition

}  
  
**Powielony kod w rozgałęzieniach: -Wduplicated-branches**

Kompilator wypisze warning, gdy w bloku kodu if/else pojawi się zduplikowany kod:

int main(int argc, char\* argv[]) {

int i {};

if (argc == 0) { // this condition has identical branches [-Wduplicated-branches]

i = 2;

}

else {

i = 2;

}

}  
  
**Wykrywanie błędów logicznych: -Wlogical-op**

Dostaniemy ostrzeżenie, gdy zostanie wykryte użycie operatora logicznego zamiast bitowego lub gdy warunki są takie same:

int main(int argc, char\* argv[]) {

if (argc > 0 || argc > 0); // logical 'or' of equal expressions

int i = argc || 0x00031; logical 'or' applied to non-boolean constant

}  
  
**Nieprzydatne rzutowanie: -Wuseless-cast**

Kompilator wypisze ostrzeżenie, gdy będziemy rzutować na ten sam typ:

**int** **main**(**int** argc, **char**\* argv[]) {

**int** i = **static\_cast**(argc); // useless cast to type 'int'

}  
  
**Rzutowanie zmiennoprzecinkowe: -Wdouble-promotion**

Kompilator ostrzeże nas, gdy wystąpi niejawne rzutowanie z typu float na typ double:

constexpr double LOW\_QUALITY\_PI = 3.1;

float arena(float radius) {

return LOW\_QUALITY\_PI \* radius \* radius; // implicit conversion from 'float' to 'double' to match other operand of binary expression

}

int main() {}