9 Kompilacja* – rozszerzenie

Proces kompilacji

GCC to open-sourceowy kompilator, mający swoje początki w latach osiemdziesiątych XX wieku. Początkowo zwany "GNU C Compiler" skupiał się na języku C, potrafiąc generować kod na wiele różnych typów maszyn i systemów operacyjnych. Obecnie jest to "GNU Compiler Collection" potrafiący obsłużyć wiele języków, takich jak C, C++, Objective-C, Java, Fortran, Ada... Kompilator ten zdominował środowisko linuxowe na długie lata, chociaż i w systemie Windows też może zostać zaadoptowany poprzez środowisko MinGW albo Cygwin.

Clang to nowsze rozwiązanie powstałe już na początku XXI wieku. Oparte jest o kompilator LLVM (Low Level Virtual Machine) które w założeniu pozwala na przetwarzanie wielu języków. LLVM, a zarazem i clang, zyskał bardzo na popularności, ze względu na stosunkową prostotę rozbudowy.

Równolegle, na systemach Windows z oczywistych powodów dominuje Visual C/C++.

Różnice w możliwościach tych kompilatorów maleją z czasem. Dużym postępem unifikującym działanie kompilatorów było wprowadzenie standardów C++11 i C++14. Ponadto, wiele parametrów przekazywanych do kompilatorów `gcc` i `clang` są jednakowe.

Ostrzeżenia i błędy

Istnieje duża grupa błędów które można wykryć podczas kompilacji, analizując statycznie kod. Niektóre sytuacje mogą też wyglądać podejrzanie, chociaż wcale błędem nie są. Każdy rodzaj analizy można włączyć odpowiednim parametrem zaczynającym się na `-W`, na przykład:

`-Waddress`
wykrywa podejrzane operacje na adresach, jak np:

```
bool foo() { ... }
int main() {
  if (foo)      //czy adres funkcji jest różny od 0? (zawsze prawda)
    ...
```

.

`-Wparentheses`
wymusza użycie nawiasowania w podejrzanych wyrażeniach. Na przykład:

```
if (a=b)       //przypisz b do a i sprawdź czy wartość jest różny od 0?
    ...
```

w tej sytuacji programista zwykle ma na myśli porównanie `a` do `b`, i.e. `if (a==b)...`. Jednakże, czasami własnie o przypisanie chodzi. Wtedy należy umieścić wyrażenie w

podwójnym nawiasie by ukryć ostrzeżenie:

```
if ((a=b))      //tak, naprawdę chcę przypisać b do a
    ...
```

`-Wreturn-type`
Ostrzega gdy istnieje ścieżka we funkcji nie-`void`, która nie zwraca wartości. Sprawdza jeszcze kilka innych podobnych błędów związanych z powrotami z funkcji.

```
int foo(int x) {
    if (x<5)
        return 3;
    //a jeśli x>=5 to co?
}
```

Lista wszystkich możliwych ostrzeżeń jest bardzo długa. Na szczeście można wiele z nich włączyć jednym parametrem `-Wall`. Ponadto:

`-Wpedantic`	ostrzega według ścisłych reguł ISO C/C++. Wszędzie tam gdzie standard wymusza ostrzeżenia, tam one się pojawią. Istnieją konstrucje nie odpowiadające ISO ale które nie wymuszają ostrzeżenia.
`-Werror`	traktuje wszystkie ostrzeżenia jako błędy
`-Werror=<x>`	traktuje ostrzeżenia z kategorii `x` jako błędy. Np. `-Werror=return-type`
`-Wfatal-errors`	wykrycie pierwszego błędu natychmiast przerywa kompilację
`-w`	zapobiega wyświetlaniu błędów, nawet tych które domyślnie się pojawiają
`-Wextra`	jeszcze więcej ostrzeżeń

Inne przydatne parametry

`-D<makro>`	definiuje nowe makro, przypisując jej wartość 1
`-D<makro>=<wartosc>`	definiuje nowe makro z określoną wartością
`-D<makro>(<pars>)=<wart>`	definiuje nowe makro z parametrami
`-U<makro>`	usuwa definicję makra
`-I<dir>`	dodaj dany katalog do standardowej listy katalogów gdzie poszukiwane są pliki nagłówków.
`-x <lang>`	określa język programowania źródła. Domyślnie wybierany jest na podstawie rozszeżenia pliku.
`-std=<standard>`	określa standard języka. Najpopularniejsze wartości obecnie to: `c++11`, `c++14`, `c++17`.
`-H`	wypisuje nazwy wszystkich includowanych plików. Wcięcia pokazują zagnieżdżenie.

Preprocesor

Preprocesor można traktować jako osobny język programowania transformujący tekst kodu C/C++ w nową werjsę. Jedynie linie rozpoczynające się znakiem # reprezentują operację która ma być wykonana przez preprocesor. Pozostały tekst pozostaje bez zmian, chyba że w kodzie pojawi się nazwa jakiegoś wczesniej zdefiniowanego *makra*.

Makra bez wartości

Makro do nierekurenyjna funkcja operująca na tekście. Najprostrza definicja to: `#define X` Od tego momentu `X` jest zdefiniowana, ale oznacza pusty ciąg znaków. Każde pojawienie się (kompletnego) słowa `X` zostanie zastąpione przez pusty ciąg znaków.

Takie puste makra przeważnie przydają się w preprocesorowych warunkach, umieszczanych w `#ifdef ... #endif`. Jednym z bardziej popularnych flag tego typu jest `NDEBUG`, które oznacza że kod nie powinien być debugowany.

```
int power(int base, int exp) {
#ifndef NDEBUG
  if (exp<0)
    throw "power dziala tylko dla dodatniego wykladnika!"
#endif
  ...
}
```

Teraz jeśli kompilujemy program w wersji release, możemy ustalić marko `NDEBUG` by automatycznie usunąć wszystkie takie kontrolne linie i ogólnie przyspieszyć działanie programu. Moglibysmy np. w `Makefileu` napisać:

```
release : CFLAGS += -DNDEBUG
```

Flaga `NDEBUG` nie jest ustalana nigdzie w standardzie, ale jest ona wykorzystywana, np. w standardowej bibliotece `assert.h`. Jeśli `NDEBUG` nie jest zdefiniowane, to biblioteka definiuje funkcję `assert()` która natychmiast przerywa program jeśli podany argument jest fałszywy. Natomiast jeśli `NDEBUG` jest zdefiniowane, to `assert()` nie robi nic.

```
#include <assert.h>
int power(int base, int exp) {
  assert(exp>=0);
  ...
}
```

Innym bardzo częstym użyciem flagi makrowej jest tak zwany *include guard*, który zapobiega wielokrotnemu załączaniu tego samego plika nagłówka. Moglibyśmy zapisać power.h jako:

```
#ifndef POWER_HEADER_47836846283468
#define POWER_HEADER_47836846283468
int power(int base, int exp);
int sqr(int x);
#endif
```

Taka konstrukcja składa się z trzech częsci:

* Na początku sprawdzamy czy dane makro już jest zdefiniowane. Jeśli tak, to przyjmujemy że dany plik już został zainkludowany.
* W drugiej linii, wewnątrz warunku, definiujemy marko identyfikujące dany plik.
* Na samym końcu pliku kończymy warunek.

Ważne by makro identyfikujace byłu unikalne w projekcie, a najlepiej na świecie. Jeśli ktoś w przyszłości użyje Waszego nagłówka, to dobrze by przez przypadek nazwa makra się nie powtórzyła w innym pliku.

Makra z wartością

Makro może mieć wartość – albo liczbową albo jako dowolny inny tekst, definiowany za pomocą:
`#define <makro> <wartosc>`

przy wartościach numerycznych, częstym błędem jest postawienie średnika na końcu, co może powodować nieoczekiwane błędy:
```
#define SIZE 100;
int arr[SIZE];          //rozwinie sie do: int arr[100;];
```

Generalnie makra powinny być używane tylko gdy ich użycie jest potrzebne podczas preprocessingu. Gdy potrzebna jest stała podczas kompilacji można to uzyskać w inny sposób, np.
```
static const int size=100; //C++ only
enum { size=100 };
constexpr int size=100; //C++11 only (g++ -std=c++11)
```

Makra numeryczne można porównywać w preprocesorze za pomocą `#if`, na przykład
```
#if __GNUC__ > 3
printf("Skompilowane za pomoca gcc wersji > 3")
#endif
```

Jak w przypadku pokazanym powyżej, kompilatory często automatycznie definiują serię makr które można wykorzystać celem rozpoznania wersji systemu, kompilatora czy docelowej maszyny. Kompilatory często też definiują magiczne makra które mają jakieś wbudowane działanie.

`__GNUC__` `__GNUC_MINOR__`	gcc, clang	wersja gcc (zdefiniowana także w clang!)
`__clang__`	clang	flaga rozpoznająca clang
`__clang_major__` `__clang_minor__`	clang	wersja clang
`_MSC_VER`	VC++	wersja Visual C++, wyrażona jako `major*100+minor`
`__cplusplus`	standard	istneje jeśli źródło jest kompilowane językiem C++

`__CUDACC__`	nvcc	flaga rozpoznająca nvcc sterujące kompilacją
`__CUDA_ARCH__`	nvcc	wersja karty graficznej dla której następuje kompilacja w CUDA
`__INTELLISENSE__`	VC++	Zdefiniowane tylko podczas działania modułu Intellisense w Visual C++
`__COUNTER__`	*	każde użycie jest kolejną liczbą całkowitą
`__FILE__`	*	nazwa aktualnego pliku źródłowego
`__LINE__`	*	aktualna linia w pliku źródłowym
`__func__`	*	nazwa funkcji wewnątrz której makro się znajduje
`__FUNCSIG__`	VC++	nazwa funkcji wraz z sygnaturą w której makro się znajduje
`__DATE__`	*	dzień w którym następuje kompilacja, np. `Jan 09 2016`
`__TIME__`	*	moment kompilacji pliku jako ciąg znaków `hh:mm:ss`

Makra funkcyjne

Makra funkcyjne, zwane czasami po prostu makrami, to nierekurenyjne funkcje preprocesora które można wywołać gdzieś w kodzie. Makra pobierają dowolną liczbe argumentów i na ich podstawie składają jakiś tekst. Na przykład:

```
#define SQR(x) ((x)*(x))
```

wszędzie tam gdzie pojawi się wywołanie `SQR`, zostanie ono zastąpione ciągiem znaków wskazanym w definicji, zaś argumenty `x` i `y` zostaną zastąpione tekstem z kodu źródłowego, np:

```
SQR(5)      ⟶    ((5)*(5))
```

Może się wydawać, że powyższa definicja zawiera zbytecznie dużą liczbę nawiasów. Trzeba jednak pamiętac, że argumenty są wstawiane verbatim do treści makra. Brak takich nawiasów może prowadzić do nieoczekiwanych rezultatów, np:

```
#define SQR(x) x*x
SQR(1+1)    ⟶      1+1*1+1
~SQR(5)     ⟶      ~5*5
```

Więcej na temat użycia makr przedstawione jest na "Podstawach programowania".

Pragmy

Instrukcja `#pragma ...` to zestaw operacji z założenia specyficznych dla kompilatora i nie zdefiniowanych w standardzie. Najprostrzym przykładem jest

```
#pragma once
```

które działa w taki sam sposób jak include guard, ale jest mniej elastyczne: wyłącza cały jeden plik. Nie można wyłaczyć części, bądź wykorzystać w kilku plikach. Z drugiej

strony, taka pragma często działa szybciej i nie trzeba wymyślać unikalnej nazwy makra identyfikującego.

Inne przykłady:

`#pragma GCC optimize (...)`	gcc	Włącza specyficzne strategie optymalizacyjne dla pozostałej części kodu. Na przykład argument `("unroll-loops")` spowoduje rozwinięcie pętli podczas kompilacji
`#pragma unroll`	nvcc	Rozwija pętlę bezpośrednio następującą po tym wywołaniu
`#pragma GCC diagnostic ...`	gcc,clang	określa nowe reguły wyświetlania ostrzeżeń i błędów. Na przykład `ignored "-Wall"` wyłacza wszystkie warningi które wcześniej były objęte parametrem `-Wall`
`#pragma warning (<spec>:<list>)`	VC++	Ustala poziom raportowania ostrzeżeń dla błędów o wybranych numerach z listy `<list>`. Na przykład `#pragma warning(disable : 4033)` wyłącza ostrzeganie o wywołaniu `return` bez argumentu z funkcji nie-`void`.
`#pragma pack(n)`	VC++,gcc	Gęsto upakowuje pola struktury czy klasy, nawet jeśli miałoby to spowolnić działanie programu
`#pragma omp parallel for`	OpenMP	Kompilator wspierający rozszeżenie OpenMP generuje wielowątkowy kod do iteracji pętli. GCC i clang wymaga flagi `-fopenmp`. Wsparcie dla OpenMP w clang jest stosunkowo nowe i bywają problemy z jego adopcją. Visual C++ wymaga flagi `/fopenmp`. W cechach projektu można to ustalić pod `C/C++ -> Language -> Open MP Support`.

Rózne kompilatory definiują jeszcze własne, inne komendy procesora. Najpopularniejsze to:

`#line <n> <name>`	*	Przyjmuje, że następująca linia ma numer `<n>` i pochodzi z pliku `<name>`. Przydatne gdy kod źródłowy jest generowany przez inne narzędzie.
`#error <str>`	*	generuje błąd podając `<str>` jako jego opis
`#warning <str>`	gcc, clang	generuje ostrzeżenie
`#include <str>`	*	Załącza plik o podanej nazwie `<str>`. Nazwa może być umieszczona w nawiasach ostrych `<>` lub cudzysłowiu `""`. W pierwszym przypadku, standardowe katalogi są przeszukiwane (oraz te przekazane przez `-I`). W drugim przypadku, przeszukiwany jest wpierw bierzący katalog.

`#include MACRO`	*	Rozwija `MACRO` do nazwy i załącza dany plik.

Biblioteki dynamiczne

Czasami zdarza się, że proces dynamicznego linkowania jest skomplikowany i prowadzi do trudno wykrywalnych błędów. Na szczęście można prześledzić ten proces ustawiając zmienną środowiskową `LD_DEBUG`. Zmienną tą można ustawić na kilka opcji:

`libs`	wypisuje proces wyszukiwania plików `.so` w różnych katalogach
`reloc`	wypisuje komunikaty relokacji kodu – operacji doklejania kodu biblioteki do głównego programu w pamięci operacyjnej
`bindings`	wypisuje proces łączenia każdego użytego symbolu z jego definicją
`versions`	wypisuje proces zależności wersji, gdy dana biblioteka dynamiczna jest wydana w wielu wersjach
`all`	szczegółowy debug zawierający w sobie wszystkie powyższe (i nie tylko)
`statistics`	informacja o liczbie dokonanych operacji, łącznie z czasem działania

Zmienne środowiskowe `LD_LIBRARY_PATH` i `LD_DEBUG` powinny być używane jedynie do własnych celów, np. gdy piszemy nowy program. Gdy chcemy opublikować naszą aplikację, nie możemy zakładać że końcowy użytkownik będzie miał te zmienne ustawione (albo co gorsza – samemu mu je ustawiać, np. w jakimś skrypcie instalacyjnym). Działanie tych zmiennych jest globalne – wpływa na wszystkie uruchamiane programy. Zmiana wartości tej zmiennej może spowodować, że nagle inny program – który dotychczas działał poprawnie – nagle zaczyna się linkować podczas uruchamiania ze złą biblioteką (np. starszą jej wersją), prowadząc do trudno wykrywalnych błędów.

Dlatego `-rpath` jest zdecydowanie zalecaną metodą.

Dekoracja nazw

Podczas linkowania funkcje identyfikowane są za pomocą nazwy. Jeśli w dwóch linkowanych plikach pojawią się dwa symbole o tej samej nazwie, pojawi się błąd, np:

```
libpower.a(power.o): In function ‘sqr’:
power.c:(.text+0x0): multiple definition of ‘sqr’
main.o:main.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Najczęstrzym powodem takiego błędu jest, gdy dana funkcja została zdefiniowana w pliku nagłówka, który jest inkludowana w wielu plikach źródłowych `.c`.

Jednakże, w języku C++ dopuszczalne jest by wiele funkcji miało tą samą nazwę, jeśli tylko typy parametrów są różne. Np. dopuszczalne jest by max.cpp wyglądał następująco:

```
int max(int x, int y) { return x>y?x:y; }
float max(float x, float y) { return x>y?x:y; }
```

Jeśli skompilujemy takie źródło do pliku obiektowego, to będzie on zawierał obie definicje i będą one poprawnie linkowane w innych źródłach C++. Jak to możliwe? W plikach obiektowych nie przechowujemy informacji o typach.

Kompilator C++, w przeciwieństwie do C, koduje informację o typach w nazwie funkcji. Jeśli spróbujemy podejrzeć te nazwy, zobaczymy:

```
$ g++ max.cpp -c -o max.o
$ nm max.o
000000000000001c T _Z3maxff
0000000000000000 T _Z3maxii
```

Powstała nazwa jest „udekorowana" (*name mangling*, albo *name decoration*). Przechowuje ona w swej nazwie informacje o typach, ale także o przestrzeni nazw – jeśli funkcja została zadeklarowana wewnątrz jakiegoś `namespace`, ale także wewnątrz klasy. Sytuacja się komplikuje jeszcze bardziej gdy argumenty są typami templatowymi. Różne kompilatory dekorują na różny sposób. Czasami nawet inne wersje tego samego kompilatora różnią się sposobem dekoracji.

Ogólnie wszystkie nazwy dekorowane zaczynają się on pojedynczego podkreślnika. Przyjęte jest w standardzie, że użytkownik nie powinien tak nazywać swoich zmiennych i funkcji – właśnie by uniknąć niespodziewanych konfliktów z dekorowaniem.

Jeśli chcielibyśmy odzyskać informacje z tak zakodowanej nazwy, możemy użyć komendy `c++filt`:

```
$ c++filt _Z3maxff
max(float, float)
```

Zauważmy, że nie dowiedzieliśmy się jaki typ jest zwracany. Faktycznie – ta informacja nie jest tam zawarta. Nie jest ona potrzebna, ponieważ w C++ funkcje nie mogą różnić się tylko typem zwracanej wartości.

W większości przypadków dekorowanie i od-dekorowanie jest realizowane przez kompilator i linker bez naszej wiedzy. Musimy o tym jednak pamiętać w dwóch przypadkach:

* Funkcję napisaną w C++ chcemy wywołać z języka C lub innego.
* Chcemy wczytać bibliotekę „ręcznie" w kodzie programu (o czym za chwilę)

Aby móc te rzeczy zrobić, trzeba funkcję napisaną w C++ odpowiednio przygotować, poprzez wyłączenie dekorowania. Robimy to konstrukcją `extern "C"`: albo umieszczając to przed każdą funkcją którą chcemy zmienić, albo tworząc cały blok tak oznaczony:

```
extern "C" {
int power(int base, int exp);
int sqr(int x);
}
```

Wewnątrz konstrukcji `extern "C"` nie możemy korzystać z tych udogodnień C++ które wymagają dekorowania. W szczególności, funkcje tam zadeklarowane nie mogą być przeciążone.

Załóżmy, że nasz projekt implementuje funkcje `power` i `sqr` w języku C++, ale funkcja `main` jest w C. Przypomnijmy, że wewnątrz `main` znajduje się wywołanie `power`. Będziemy chcieli skompilować projekt następującymi operacjami:

```
$ g++ power.cpp -fPIC -c -o power.o
$ g++ sqr.cpp -fPIC -c -o sqr.o
$ g++ -shared power.o sqr.o -o libpow.so
$ gcc main.c -c -o main.o
$ gcc main.o -L. -lpow -Wl,-rpath=. -o main
```

Zmienimy wtedy plik `power.h` tak by zawierał konstrukcję `extern "C"` tak jak w przykładzie powyżej. Szybko pojawi się problem: język C nie rozumie tej konstrukcji. Tak napisany plik `power.h` nie może zostać zainkludowany w `main.c`. Rozwiązać to możemy szybko używając preprocesora:

```
#ifdef __cplusplus
extern "C" {
#endif
int power(int base, int exp);
int sqr(int x);
#ifdef __cplusplus
}
#endif
```

Nie jest to piękne, ale jest to dość powszechne. W ten sposób ten sam plik nagłówka może być stosowany zarówno dla plików C jak i C++ i w obu przypadkach obowiązywać będą nazwy nieudekorowane. Tak skorygowany projekt skompiluje się poprawnie i z sukcesem wywołamy funkcje C++ z poziomu czystego C.

Symbole publiczne i prywatne

Jak pisaliśmy już wcześniej, jeśli w dwóch linkowanych bibliotekach pojawi się ta sama nazwa – spowoduje to błąd. Gdy jest to ta sama funkcja, to problem często można rozwiązać lepiej przeprojektowując projekt. Natomiast zdarza się, że kolizje są zupełnie przypadkowe. Na przykład – być może dwie biblioteki tworzą swoją wewnętrzną, prywatną funkcję `debug` albo `print` albo `dump` etc... Każda z nich działa zupełnie inaczej i ma sens tylko w kontekście danej biblioteki.

Najprostszym rozwiązaniem dostępnym w standardzie języka C, to zadeklarować daną funkcję jako `static`. Taka funkcja ma `statyczne linkowanie`, co oznacza – jest dostępna wyłącznie w obrębie pliku obiektowego w którym jest ona kompilowana.

```
static void debug();
```

Uwaga: słowo kluczowe `static` jest "przeciążone". Przeważnie używa się go jako:

* Statyczna zmienna lokalna wewnątrz funkcji, ma żywotność zmiennej globalnej, ale dostępna jest tylko z danej funkcji.
* Statyczny element klasy w C++ jest zmienną/funkcją globalną, ukrytą w przestrzeni nazw danej klasy

Natomiast my tutaj mówimy o zwykłej funkcji lub zmiennej globalnej zadeklarowanej jako `static`.

Takie funkcje nadal mają swój wpis w tabeli symboli, ale linker wie, że je należy ignorować gdy dany plik łączony jest z innym:

```
$ cat static.c
static int foo = 1;
static const int bar = 2;
static void debug() {}
$ gcc static.c -c -o static.o
$ nm static.o
0000000000000000 r bar
0000000000000000 t debug
0000000000000000 d foo
```

Takie statyczne wartości reprezentowane są literą małą; publiczne są duże.

Częściej zdarza się, że dany symbol jest używany w obrębie całej biblioteki (a nie tylko pojedynczego pliku obiektowego) ale nie powinien być widoczny przez potencjalnego użytkownika takiej biblioteki. Niestety standard języka C/C++ nie daje jednoznacznego rozwiązania. W gcc/g++ można to ustawić za pomocą *atrybutów*. Atrybuty do dodatkowa informacja załączona do elementów języka, takich jak funkcje i zmienne. W gcc atrybuty tworzy się pisząc `__attribute__(( ... ))` przed albo po elemencie nas interesującym. W naszym przypadku chcemy napisać:

```
int sqr(int x) __attribute__((visibility("hidden")))
```

Tak skompilowany projekt spowoduje, że nasza biblioteka `libpow.so` nadal będzie zawierała funkcję `sqr` ale nie będzie ona dostępna na zewnątrz biblioteki:

```
$ nm libpower.so | grep "\( sqr$\)\|\( power$\)"
0000000000000680 T power
00000000000006e0 t sqr
$ nm main | grep "\( sqr$\)\|\( power$\)"
                 U power
```

Gdybyśmy teraz spróbowali użyć `sqr` wewnątrz main-a, dostalibyśmy taki komunikat linkera:

```
main.o: In function ‘main’:
main.c:(.text+0x61): undefined reference to ‘sqr’
```

Standard C++11 dostarcza też składnie `[[ ... ]]`, ale brak jest odpowiedniego standardowego atrybutu i trzeba używać niestandardowych z przestrzeni nazw gnu.

```
[[gnu::visibility("hidden")]] int sqr(int x);
```

Jeśli wszystkie funkcje mają być domyślnie ukryte, poza tymi które explicite określimy jako publiczne, możemy kompilować z flagą `-fvisibility=hidden`. Wówczas trzeba skorygować plik nagłówka jako:

```
int power(int base, int exp) __attribute__((visibility("default")));
int sqr(int x);
```

Symbole słabe

Symbole słabe to takie funkcje biblioteczne, które mogą zostać nadpisane przez inne biblioteki. Jeśli dla przykładu w pliku `power.h` funkcję `sqr` jako słabą:

```
int power(int base, int exp);
int sqr(int x)  __attribute__((weak));
```

to w pliku `main` – tam gdzie używamy naszej biblioteki dynamicznej – jesteśmy wstanie ją przedefiniować. Jeśli na końcu pliku dopiszemy

```
int sqr(int x) { return x*x*x; }
```

program skompilujemy i uruchomimy, to okaże się, że `sqr` została nadpisana. Nowa funkcja jest wywoływana nawet z wewnątrz biblioteki!

```
$ ./main 2 4
512
```

Takie słabe linkowanie stosuje się przeważnie gdy użytkownik redefiniuje standardowe funkcje c/c++. Przydatnym może okazać się na przykład:

* Definiowanie własnych reguł obsługi błędów
* Definiowanie własnych globalnych operatorów obsługujących pamięć (`new`,`delete`)

Symbole słabe mogą pojawić się też w programie głównym. Tak zadeklarowane funkcje nie muszą posiadać definicji – nawet jeśli są one użyte! Niezdefiniowana funkcja ma adres `null` i może zostać uzupełniona poprzez załadowanie kolejnej biblioteki (np. manualnie – o czym za chwilę)

Manualne otwieranie bibliotek dynamicznych

Jak dotąd w powyższych przykładach biblioteki dynamiczne były otwierane na początku działania programu. Wszystkie informacje: jaką bibliotekę otworzyć i które funkcje wczytać (zaimportować) zapisane są „na twardo" w kodzie programu. Tak być nie musi! Możemy ręcznie w kodzie otworzyć nową bibliotekę i załadować z niej dowolny symbol, nawet jeśli podczas kompilacji nie wiemy co będziemy łączyć. Taki sposób linkowania przydatny jest np. przy definiowaniu *pluginów* do aplikacji: dodatkowej funkcjonalności programu bez potrzeby rekompilacji źródła.

Funkcjami do operacji na bibliotekach dynamicznych są dostępne w standardowej bibliotece o nagłówku `dlfcn.h`, i linkowanej przez `-ldl`:

`dlopen`		Otwiera nową bibliotekę dynamiczną
	`filename`	nazwa pliku do otwarcia. Jeśli jest `null` to otwierany jest aktualnie załadowany program główny
	`flags`	Parametry jak biblioteka ma być ładowana. Niektóre z nich to:

	`RTLD_LAZY`	tworzy połączenie z poszczególnymi symbolami w chwili ich pierwszego użycia
	`RTLD_NOW`	wszystkie połączenia są rozwinięte natychmiast. Jedna z flag: `RTLD_LAZY` albo `RTLD_NOW` musi być ustawiona
	`RTLD_GLOBAL`	wszystkie symbole są potencjalnie łączone z kolejnymi bibliotekami otwieranymi dynamicznie.
	`RTLD_LOCAL`	symbole łączone są jedynie z niezdefiniowanymi, już istniejącymi pozycjami.
	`RTLD_NODELETE`	nie wyładowuje biblioteki przy użyciu `dlclose`. W szczególności, symbole statyczne nie będą reinicjalizowane przy wielokrotnym otwarciu biblioteki w ten sposób.
	zwracana wartość	handler do biblioteki – do wykorzystania przez pozostałe funkcje. Może być `null` jeśli operacja się nie powiodła.
`dlsym`		Ładuje wskazany symbol
	`handle`	Handler do biblioteki w której chcemy wyszukać symbol
	`symbol`	Nazwa symbolu (jako standardowy string w C)
	zwracana wartość	wskaźnik do symbolu. Może być `null` jeśli biblioteka jedynie deklaruje ów symbol, nie podając definicji, lub jeśli operacja się nie powiodła.
`dlclose`		Zamyka wskazaną bibliotekę
	`handle`	Handler do biblioteki którą zamykami

Jak to wygląda w praktyce? Załóżmy, że nasza biblioteka `libpow.so` oprócz `power` definiuje jeszcze funkcje `sum` i `max` które dla dwóch argumentów całkowitych wykonują wskazaną operację. Wszystkie te funkcje mają taką samą sygnaturę: `int (int, int)`.

W programie `main.c` nie będziemy wywoływać wprost żadnej z tych funkcji, lecz zapytamy użytkownika którą chce uruchomić:

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
int main(int argc, char** argv) {
  const char* funcname = argv[1];
  int a = atoi(argv[2]);
  int b = atoi(argv[3]);
```

Następnie spróbujemy dynamicznie otworzyć naszą bibliotekę:

```
  void* handle = dlopen("libpow.so",RTLD_LAZY);
  if (!handle) {
      fprintf(stderr,"Failed to open the library!");
      abort();
  }
```

Jeśli się powiodło, spróbujemy załadować naszą funkcję. Do tego celu tworzymy *wskaźnik na funkcję* który w języku C ma koszmarną składnię:
`<zwracana wartosc> (*<nazwa zmiennej>)(<typy argumentow>)`
wszystkie nawiasy w powyższej konstrukcji są konieczne. Pominięcie pierwszych spowoduje, że parser zrozumie to jako zwykłą deklarację funkcji zwracającej wskaźnik do `<zwracana wartosc>`. Drugie nawiasy są konieczne, bo określają one parametry funkcji.

```
  int (*function)(int, int);
  function=dlsym(handle, funcname);
  if (!function) {
      fprintf(stderr,"Failed to find function %s!", funcname);
      abort();
  }
```

Jeśli operacja zakończyła się pomyślnie, to `function` wskazuje teraz na odpowiednią funkcję w naszej bibliotece. Możemy teraz ją użyć w normalny sposób:

```
printf("%d\n",function(a,b));
```

Na koniec zamykamy bibliotekę i kończymy program:

```
  dlclose(handle);
  return 0;
}
```

Bibliotekę dynamiczną kompilujemy jak zawsze. Kompilacja programu głównego jest trochę prostrza:

```
$ gcc main.c -c -o main.o
$ gcc main.o -ldl -Wl,-rpath=. -o main
```

Zwróćmy uwagę, że do kompilacji nie potrzebujemy biblioteki `libpow.so`. Ważne tylko by ona istniała gdy program uruchomimy.

```
$ ./main power 2 4
16
$ ./main sum 2 4
6
$ ./main max 2 4
4
$ ./main minus 2 4
Failed to find function minus!Aborted
```

Zauważmy, że chociaż bibliotekę ładujemy dynamicznie, nie zwalnia nas to z obowiązku znania sygnatury funkcji. Nie jest to dziwne: musimy wiedzieć ile parametrów przekazać i jak interpretować wynik.

Gdy nawet ta informacja nie jest znana statycznie, trzeba korzystać z bardziej zaawansowanych bibliotek, jak np. `libffi`. Nie będziemy jednak jej już opisywać na tym kursie.

Jeszcze raz, plik main.c w jednym kawałku:

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
int main(int argc, char** argv) {
  const char* funcname = argv[1];
  int a = atoi(argv[2]);
  int b = atoi(argv[3]);
  void* handle = dlopen("libpow.so",RTLD_LAZY);
  if (!handle) {
      fprintf(stderr,"Failed to open the library!");
      abort();
  }
  int (*function)(int, int);
  function=dlsym(handle, funcname);
  if (!function) {
      fprintf(stderr,"Failed to find function %s!", funcname);
      abort();
  }
  printf("%d\n",function(a,b));
  dlclose(handle);
  return 0;
}
```