Konspekt

[Optymalizacja](#_mqmkcr4h50fn)

[-O0](#_zsrqnpg4l06)

[-O1](#_t6rwb172vtfg)

[-O2](#_h7knerraf6ae)

[-O3](#_wis3hxue3sls)

[-Os](#_tg3c5y4csq7b)

[-Ofast](#_l7qkek9nkl7j)

[-Og](#_6nnujitpt7r2)

[-flto (Link Time Optimization)](#_nk3iqp9qt0dy)

[Użyteczne komendy](#_9vkgp1c8env7)

[Zadania](#_ll194qhcadve)

[Zadanie 0: Porównanie różnych poziomów optymalizacji](#_e75u1shfkx8k)

[Zadanie 1: Zmienne globalne oraz lokalne](#_n8xcvotf5om7)

[Zadanie 2: Pętle](#_wdodkkguh6bq)

[Zadanie 3: Instrukcje warunkowe](#_cowj2mxaoggi)

[Zadanie 4?: Eliminacja wspólnych podwyrażeń](#_ug4xt79c1f1e)

[Zadanie 5: Optymalizacje O0](#_ywwzhhk1p2th)

[Zadanie 6: Ofast](#_qihfwk2va7rz)

[Zadanie 7: Proste optymalizacje a optymalizacja kompilatora](#_r9jtvfgdbt55)

[Zadanie 8: Link Time Optimization](#_4bvu32ljr6x0)

# Optymalizacja

W celu optymalizacji kodu za pomocą kompilatora gcc, korzystamy z opcji optymalizacji, opisanych poniżej:

### -O0

Domyślna opcja kompilatora. Kod źródłowy jak najdokładniej zamieniany jest na kod maszynowy, bez żadnych optymalizacji. W przypadku braku opcji optymalizacji, żadna flaga optymalizacji nie będzie miała wpływu na kompilacje. Nawet jeśli zostaną podane. Stosowany podczas tworzenia oprogramowania, ze względu na możliwość debugowania oraz najszybszy czas kompilacji.

### -O1

Kompilator próbuje zmniejszyć ilosć kodu i czas wykonania programu, bez wprowadzania optymalizacji które zajmują dużo czasu podczas kompilacji.

### -O2

Kompilator wykonuje prawie wszystkie optymalizacje, które nie wymagają zwiększenia wielkości pliku wykonywalnego. Poziom ten stosuje m.in. jądro Linuxa oraz wszystkie projekty GNU. Przez to jest najdokładniej testowany i generalnie uważany za najlepszy pod względem równowagi pomiędzy bezpieczeństwem i prędkością.

### -O3

Kompilator robi wszystko to co przy poziomie 2, ale włącza flagi które mogą zwiększać wielkość pliku wykonywalnego. Wątpliwe korzyści z korzystania z tego poziomu sprawiają, że jest dość rzadko używany.

### -Os

Opcja próbująca za wszelką cenę zmniejszyć wielkość kodu wykonywalnego. Włącza opcje podobne do -O2, ale z wyłączonymi niektorymi flagami.

### -Ofast

Opcja która przyspiesza kod jeszcze lepiej niż -O3, ale może wprowadzać niebezpieczne operacje, które mogą zmienić niektóre elementy działania programu.

### -Og

Poziom optymalizacji, który nie przeszkadza w debugowaniu. Optymalizuje mniej niż -O1

### -flto (Link Time Optimization)

Opcja kompilacji umożliwiająca optymalizację w czasie linkowania.

Pliki kompilowane przy użyciu tej opcji mogą zostać zoptymalizowane

tak jakby stanowiły pojedynczą jednostkę kompilacji.

Przykład: g++ -c -flto foo.cpp

Zaleca się aby inne opcje optymalizacji (-O2, -O3) zostały podane

w trakcie kompilacji i linkowania.

Opcja -flto nie jest wymaga w trakcie linkowania.

# Użyteczne komendy

Time - komenda służąca do zmierzenia czasu działania komendy podanej przez argumenty wywołania.

Przykład użycia: time ./a.out

gprof - profilowanie, przydatne gdy chcemy wykryć punkty krytyczne naszego programu

--demangle - wypisuje nazwy funkcji tak, jak zostały zdefiniowane

-b - krótka wersja wyników, brak opisów

Aby móc wywołać gprof należy wykonać kompilację za pomocą gcc z flagą -pg

objdump -d OFILE.o - deasemblacja kodu binarnego

# Zadania

### Zadanie 0: Porównanie różnych poziomów optymalizacji

Zadanie pierwsze polega na wykorzystaniu linuksowej komendy time, w celu znalezienia najlepszej opcji optymalizacji dla przykładowego programu. Używając opcji optymaliacji, napisz skrypt bashowy do sprawdzania czasu wykonania programu z każdą opcją optymalizacji i znajdź dla której opcji:

1. czas wykonania jest najkrótszy
2. Plik wynikowy jest najmniejszy (najbardziej miarodajne będzie sprawdzenie kodu w assemblerze)

### Zadanie 1: Zmienne globalne oraz lokalne

1) Sprawdź kod asemblera powstały z kompilacji programu (np. zatrzymaj gcc przed wykonaniem asemblacji, użyj objdump lub wejdź na stronę godbolt)

2) Sprawdź w jaki sposób kompilator obsługuje zmienne lokalne oraz globalne (gdzie oraz w jaki sposób są przechowywane, oraz jak są obsługiwane w trakcie trwania programu)

3) Dodaj flagę optymalizacji np. O3 - czy są widoczne różnice pomiędzy traktowaniem zmiennych globalnych oraz lokalnych?

4) Zamień zmienne globalne na lokalne, sprawdź kod assemblera z włączoną flagą O3, które linie kodu zostały wykonane w trakcie kompilacji? (odpowiedź: wszystkie poza printf)

### Zadanie 2: Pętle

1) Sprawdź kod asemblera powstały z kompilacji programu za pomocą clang oraz gcc bez flag kompilacji,

przeanalizuj ilość instrukcji warunkowych oraz częstotliwość ich wywołania, jakie wady widać?

2) Wykonaj optymalizacje -O1 w compilatorze clang, co stało się z pętlą oraz operacjami wewnątrz niej? Porównaj kod zoptymalizowany za pomocą gcc z flagą -O3

3) porównaj czas wykonania programu stosując różne flagi gcc oraz clang

4) porównaj rozmiar pliku wykowywalnego

### Zadanie 3: Instrukcje warunkowe

1) porównaj kod asemblerowy flag O0 oraz O1

2) postaraj się zmienić kod źródłowy tak, aby przypominał on w działaniu zoptymalizowaną wersję

(pomiń wpływ kopiowania zmiennych do rejestrów oraz operacje na stosie),

tj. doprowadź do postaci w której istnieje tylko jedna etykieta oraz jest możliwy tylko jeden skok

3) Warto także zobaczyć różnicę po wybraniu flagi Og. Co powoduje ta zmiana?

### Zadanie 4?: Eliminacja wspólnych podwyrażeń

1) Porónaj kod maszynowy powstały w wyniku kompilacji z flagami O0 oraz O3

### Zadanie 5: Optymalizacje O0

1) zakładając, że x musi być const, oraz nie jest możliwa zmiana funkcji f, zmodyfikuj kod tak, aby funkcja wypisywała wartość zgodną z oczekiwaniami

### Zadanie 6: Ofast

1) Porównaj kod assemblerowy przy użyciu flag O0, O3,oraz Ofast, jakie widać różnice? Kiedy taka optymalizacja jest pożądana?

2) Zaproponuj takie wartości x, y, z, oraz w wewnątrz programu main.cpp, aby wynik obliczeń z flagą Ofast był niepoprawny, natomiast z O3 tak.

### Zadanie 7: Proste optymalizacje a optymalizacja kompilatora

1) Skompiluj program bez asemblacji z flagą O0, porównaj ze sobą różne wersje tych samych operacji, czy są widoczne różnice?

2) Za pomocą gcc -gp test.c skompiluj program, wykonaj go, a następnie za pomocą gprof sprawdź które operacje są bardziej optymalne.

Czy zgadza się to z przewidywaniami?

3) Czy po dodaniu flag optymalizacji kompilator potrafi sam wykonać za nas takie optymalizacje?

### Zadanie 8: Link Time Optimization

Skompiluj podane pliki (przy użyciu O2 lub O3), dokonaj linkowania i sprawdź czas wykonania przy pomocy polecenia time. Następnie użyj opcji flto. Jak zmienił się czas wykonania i jak rozmiar pliku?

### 