Systemy wbudowane

Wykład 4

Łukasz Piwowar

Operacje bitowe

Operatory bitowe:

- negacja bitowa ("~"),
- koniunkcja bitowa ("&"),
- alternatywa bitowa ("|") i
- alternatywa rozłączna (XOR) ("^").

```
 "~" | 0 1     "&" | 0 1     "|" | 0 1     "^" | 0 1
-----+-----   -----+-----   -----+-----   -----+-----
     | 1 0       0 | 0 0       0 | 0 1       0 | 0 1
               1 | 0 1       1 | 1 1       1 | 1 0

   a    | 0101  =  5
   b    | 0011  =  3
--------+------
  ~a    | 1010  = 10
  ~b    | 1100  = 12
a & b   | 0001  =  1
a | b   | 0111  =  7
a ^ b   | 0110  =  6
```

Operacje bitowe

Operatory bitowe:

- negacja bitowa ("~"),
- koniunkcja bitowa ("&"),
- alternatywa bitowa ("|") i
- alternatywa rozłączna (XOR) ("^").

a ^ b ^ b = a

Przesunięcie bitowe

Dodatkowo, język C wyposażony jest w operatory przesunięcia bitowego w lewo ("<<") i prawo (">>"). Przesuwają one w danym kierunku bity lewego argumentu o liczbę pozycji podaną jako prawy argument.

Rozważmy 4-bitowe liczby bez znaku (taki hipotetyczny unsigned int), wówczas:

a	a<<1	a<<2	a>>1	a>>2
0001	0010	0100	0000	0000
0011	0110	1100	0001	0000
0101	1010	0100	0010	0001
1000	0000	0000	0100	0010
1111	1110	1100	0111	0011
1001	0010	0100	0100	0010

Przesunięcie bitowe a znak

Dla przesunięcia bitowego w lewo **a << b** jeżeli **a** jest nieujemna i wartość **a*2^b** mieści się w zakresie liczby to jest to wynikiem operacji. W przeciwnym wypadku działanie jest niezdefiniowane.

Przy przesuwaniu w prawo bit znaku nie zmienia się:

Przykład:

a = (b << 1); a = b*2;
a = (b << 3); a = b*8;
a = (b >> 4); a = b/16;

a	a>>1	a>>2
0001	0000	0000
0011	0001	0000
0101	0010	0001
1000	1100	1110
1111	1111	1111
1001	1100	1110

Triki związane z operacjami bitowymi

Kod w języku C	Opis działania

a ^= b;
b ^= a;
a ^= b;	

c & -c lub c & (~c + 1)	
~c & (c + 1)	
unsigned i, c = <number>; for (i = 0; c; i++) c ^= c & -c;	

Triki związane z operacjami bitowymi

Kod w języku C	Opis działania

a ^= b;
b ^= a;
a ^= b;
	Podmiana liczb a i b.

c & -c lub c & (~c + 1)	
~c & (c + 1)	
unsigned i, c = <number>; for (i = 0; c; i++) c ^= c & -c;	

Triki związane z operacjami bitowymi

Kod w języku C	Opis działania

a ^= b;
b ^= a;
a ^= b;
	Podmiana liczb a i b.

c & -c lub c & (~c + 1)	Zwraca pierwszy ustawiony bit.
~c & (c + 1)	
unsigned i, c = <number>; for (i = 0; c; i++) c ^= c & -c;	

Triki związane z operacjami bitowymi

Kod w języku C	Opis działania

a ^= b;
b ^= a;
a ^= b;
	Podmiana liczb a i b.

c & -c lub c & (~c + 1)	Zwraca pierwszy ustawiony bit.
~c & (c + 1)	Zwraca pierwszy nieustawiony bit.
unsigned i, c = <number>; for (i = 0; c; i++) c ^= c & -c;	

Triki związane z operacjami bitowymi

Kod w języku C	Opis działania

a ^= b;
b ^= a;
a ^= b;
	Podmiana liczb a i b.

c & -c lub c & (~c + 1)	Zwraca pierwszy ustawiony bit.
~c & (c + 1)	Zwraca pierwszy nieustawiony bit.
unsigned i, c = <number>; for (i = 0; c; i++) c ^= c & -c;	Zmienna “i” będzie zawierać liczbę ustawionych bitów w “c”.

```
m = (m & 0x55555555) +                  ((m & 0xaaaaaaaa) >> 1);
m = (m & 0x33333333) +                  ((m & 0xcccccccc) >> 2);
m = (m & 0x0f0f0f0f) +                  ((m & 0xf0f0f0f0) >> 4);
m = (m & 0x00ff00ff) +                  ((m & 0xff00ff00) >> 8);
m = (m & 0x0000ffff) +                  ((m & 0xffff0000) >> 16);
```

```
v -= ((v >> 1) & 0x55555555);
v = (v & 0x33333333) +                  ((v >> 2) & 0x33333333);
v = (v + (v >> 4)) & 0x0F0F0F0F;
v = (v * 0x01010101) >> 24;
```

```
#define BITCOUNT(x) (((BX_(x)+(BX_(x)>>4)) & 0x0F0F0F0F) % 255)
#define BX_(x) ((x) -                   (((x)>>1)&0x77777777) \
- (((x)>>2)&0x33333333) \
- (((x)>>3)&0x11111111))
```

```
m = (m & 0x55555555) +                  ((m & 0xaaaaaaaa) >> 1);
m = (m & 0x33333333) +                  ((m & 0xcccccccc) >> 2);
m = (m & 0x0f0f0f0f) +                  ((m & 0xf0f0f0f0) >> 4);
m = (m & 0x00ff00ff) +                  ((m & 0xff00ff00) >> 8);
m = (m & 0x0000ffff) +                   ((m & 0xffff0000) >> 16);
```

Zmienna „m” będzie zawierać liczbę bitów ustawionych w swojej pierwotnej postaci.

```
v -= ((v >> 1) & 0x55555555);
v  = (v & 0x33333333) + ((v >> 2) & 0x33333333);
v  = (v + (v >> 4)) & 0x0F0F0F0F;
v  = (v * 0x01010101) >> 24;
```

```
#define BITCOUNT(x) (((BX_(x)+(BX_(x)>>4)) & 0x0F0F0F0F) % 255)
#define BX_(x) ((x) -     (((x)>>1)&0x77777777) \
- (((x)>>2)&0x33333333) \
- (((x)>>3)&0x11111111))
```

m = (m & 0x55555555) + ((m & 0xaaaaaaaa) >> 1);
m = (m & 0x33333333) + ((m & 0xcccccccc) >> 2);
m = (m & 0x0f0f0f0f) + ((m & 0xf0f0f0f0) >> 4);
m = (m & 0x00ff00ff) + ((m & 0xff00ff00) >> 8);
m = (m & 0x0000ffff) + ((m & 0xffff0000) >> 16);
	Zmienna „m” będzie zawierać liczbę bitów ustawionych w swojej pierwotnej postaci.

v -= ((v >> 1) & 0x55555555);
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
v = (v + (v >> 4)) & 0x0F0F0F0F;
v = (v * 0x01010101) >> 24;
	Zmienna „v” będzie zawierać liczbę bitów ustawionych w swojej pierwotnej postaci.

#define BITCOUNT(x) (((BX_(x)+(BX_(x)>>4)) & 0x0F0F0F0F) % 255) #define BX_(x) ((x) - (((x)>>1)&0x77777777) \ - (((x)>>2)&0x33333333) \ - (((x)>>3)&0x11111111))	

m = (m & 0x55555555) + ((m & 0xaaaaaaaa) >> 1);
m = (m & 0x33333333) + ((m & 0xcccccccc) >> 2);
m = (m & 0x0f0f0f0f) + ((m & 0xf0f0f0f0) >> 4);
m = (m & 0x00ff00ff) + ((m & 0xff00ff00) >> 8);
m = (m & 0x0000ffff) + ((m & 0xffff0000) >> 16);
	Zmienna „m” będzie zawierać liczbę bitów ustawionych w swojej pierwotnej postaci.

v -= ((v >> 1) & 0x55555555);
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
v = (v + (v >> 4)) & 0x0F0F0F0F;
v = (v * 0x01010101) >> 24;
	Zmienna „v” będzie zawierać liczbę bitów ustawionych w swojej pierwotnej postaci.

#define BITCOUNT(x) (((BX_(x)+(BX_(x)>>4)) & 0x0F0F0F0F) % 255) #define BX_(x) ((x) - (((x)>>1)&0x77777777) \ - (((x)>>2)&0x33333333) \ - (((x)>>3)&0x11111111))	Makro zwracające liczbę bitów w zmiennej „x” (przy założeniu ze działamy na 32 bitowym integerze).

```
n = ((n >> 1)  & 0x55555555) |          ((n << 1)  & 0xaaaaaaaa);
n = ((n >> 2)  & 0x33333333) |          ((n << 2)  & 0xcccccccc);
n = ((n >> 4)  & 0x0f0f0f0f) |          ((n << 4)  & 0xf0f0f0f0);
n = ((n >> 8)  & 0x00ff00ff) |           ((n << 8)  & 0xff00ff00);
n = ((n >> 16)& 0x0000ffff) |            ((n << 16)& 0xffff0000);
```

```
popcnt(x ^ (x - 1)) & 31
```

n = ((n >> 1) & 0x55555555) | ((n << 1) & 0xaaaaaaaa);
n = ((n >> 2) & 0x33333333) | ((n << 2) & 0xcccccccc);
n = ((n >> 4) & 0x0f0f0f0f) | ((n << 4) & 0xf0f0f0f0);
n = ((n >> 8) & 0x00ff00ff) | ((n << 8) & 0xff00ff00);
n = ((n >> 16)& 0x0000ffff) | ((n << 16)& 0xffff0000);
	Odwraca bity w zmiennej typu 32 bit integer

`popcnt(x ^ (x - 1)) & 31`	fukcja popcnt() zwraca liczbę ustawionych bitów.

n = ((n >> 1) & 0x55555555)	((n << 1) & 0xaaaaaaaa); n = ((n >> 2) & 0x33333333)	((n << 2) & 0xcccccccc); n = ((n >> 4) & 0x0f0f0f0f)	((n << 4) & 0xf0f0f0f0); n = ((n >> 8) & 0x00ff00ff)	((n << 8) & 0xff00ff00); n = ((n >> 16)& 0x0000ffff)	((n << 16)& 0xffff0000);	Odwraca bity w zmiennej typu 32 bit integer
`popcnt(x ^ (x - 1)) & 31`	Zwraca pierwszy ustawiony bit w „x”, gdzie fukcja popcnt() zwraca liczbę ustawionych bitów.

Słowniczek pojęć

Wąskie gardło – część programu której wykonanie zajmuje większość czasu. Zgodnie z zasadą 90/10, 90% czasu wykonywania się programu zawiera się 10% kodu.

Własny system alokacji - jeden z wielu różnych algorytmów implementacji instrukcji „malloc” i „free” z biblioteki C. Własny system alokowania zarządza fragmentami pamięci w zależności od ich rozmiarów nie bazuje na kolejności ich alokowania.

Ograniczenie obliczeniowe - Program używa różnych zasobów komputera podczas pracy. Program który mocno polega na wydajności procesora, nazywany jest ograniczonym obliczeniowo. Taki program nie przyśpieszy (w stopniu znaczącym) nawet po zainstalowaniu większej ilości pamięci czy szybszego dysku.

Słowniczek pojęć

Rozwijanie w miejscu (inlining) – wywołanie funkcji powoduje zmianę wskaźnika stosu, rejestrów licznikowych programu, przekazywanie parametrów, I alokowanie miejsca na wynik funkcji.

To oznacza, że niemal cały stan lokalny procesora jest zapisywany I wznawiany przy każdym wywołaniu funkcji.

Procesory są projektowane z myślą o tym, wiec robią to bardzo szybko, ale dodatkowe przyspieszenie, możemy uzyskać przez rozwinięcie funkcji, lub zintegrowanie funkcji z kodem wywołującym.

Wywoływana funkcja wtedy używa stosu kodu w którym jest zintegrowana do przechowywania zmiennych lokalnych I może (jeżeli semantyka na to pozwoli) używać bezpośrednich odwołań do parametrów zamiast do ich kopii.

Słowniczek pojęć

Rozwijanie w miejscu (inlining) – wywołanie funkcji powoduje zmianę wskaźnika stosu, rejestrów licznikowych programu, przekazywanie parametrów, I alokowanie miejsca na wynik funkcji.

To oznacza, że niemal cały stan lokalny procesora jest zapisywany I wznawiany przy każdym wywołaniu funkcji.

Procesory są projektowane z myślą o tym, wiec robią to bardzo szybko, ale dodatkowe przyspieszenie, możemy uzyskać przez rozwinięcie funkcji, lub zintegrowanie funkcji z kodem wywołującym.

Wywoływana funkcja wtedy używa stosu kodu w którym jest zintegrowana do przechowywania zmiennych lokalnych I może (jeżeli semantyka na to pozwoli) używać bezpośrednich odwołań do parametrów zamiast do ich kopii.

Słowniczek pojęć

ograniczenie wejścia/wyjścia (i/o bound)

Wywołania funkcji systemu operacyjnego do odczytu i zapisu danych zajmują większość czasu. Program najwięcej czasu spędza na czekaniu aż zasób będzie dostępny, lub na zakończenie żądania (np. zapisu). Program który większość czasu spędza czekając na wejście/wyjście jest zwanym ograniczonym wejściem/wyjściem.

ograniczenie pamięciowe (memory bound)

Program który opiera się głównie na użyciu pamięci, czy to wirtualnej czy fizycznej, jest nazywany ograniczonym pamięciowo. Taki program będzie spędzał większość czasu czytając i zapisując dane w pamięci.

Słowniczek pojęć

Notacja O (O-notation)

Miara złożoności obliczeniowej programu zwykle wyrażana jako **O(f(N))** gdzie **f(N)** jest matematyczną funkcją która definiuje górny limit (razy pewna stała) oczekiwanego czasu wykonania programu dla pewnego **N**.

Przykładowo proste algorytmy sortowania mają złożoność **O(n^2)** dla **n** sortowanych elementów. Notacja **O** może być także wykorzystana do opisu złożoności pamięciowej.

Optymalizacja (optimization)

Proces poprawiania czasu wykonania programu.

W zależności od kontekstu może oznaczać proces wprowadzania poprawek przez człowieka na poziomie kodu źródłowego, lub wysiłek kompilatora nad przeorganizowaniem kodu na poziomie assemblerowym (lub na innym niskim poziomie).

Słowniczek pojęć

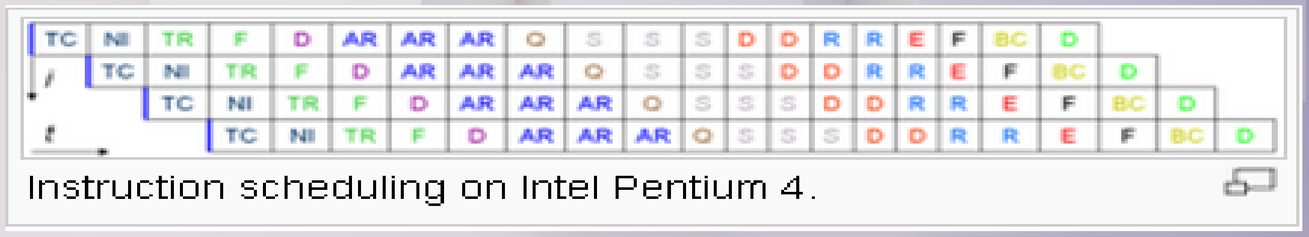
środowisko wywołań (pipeline)

architektura procesora w której etapy wywołań są przeplatane w kolejności, w taki sposób, że w momencie kiedy instrukcja jest wykonywana, następna instrukcja jest pobierana a wynik działania poprzedniej instrukcji jest zapisywany.

Niektóre procesory posiadają kilka równych etapów w ich środowisku wywołań.

profiler

program komputerowy który mierzy wydajność innego programu. Typowo pomiar składa się z samplowania rejestrów programowych, lub stosu. Po zakończeniu pomiaru, profiler zbiera statystyki i generuje raport. Niektóre profilery wymagają przekompilowania programu z specjalną opcją, i/lub linkowania z specjalnymi bibliotekami.



Instruction scheduling on Intel Pentium 4.

Słowniczek pojęć

czysta funkcja (pure function) (static)

funkcja której wynik zależy tylko od parametrów, a jej wykonanie nie ma efektów ubocznych.

ograniczenie stosowe (stack bound)

program który spędza większość dostępnego czasu na dodawaniu i usuwaniu rekordów aktywacji ze stosu jest zwany ograniczonym stosowo. Tylko kilka profilerów potrafi to zmierzyć bezpośrednio, ale można taki wniosek wyciągnąć również samemu, widząc, że większość czasu jest spędzanych w funkcji rekurencyjnej, której złożoność jest zbyt prosta, żeby wliczać się do ogólnego czasu.

volatile

rzadko używane słowo kluczowe w C, które oznacza, że zmienna powinna być nadpisywana w miejscu zamiast w rejestrze (to może powodować efekty uboczne).

Optymalizacja programów w C

Spora część pracy może polegać na czekaniu na wyniki programu. Użytkownicy i firmy poprawiają ten stan poprzez kupowanie szybszych komputerów, dodawanie pamięci, lub używanie szybszych połączeń sieciowych.

Programiści jako twórcy, są odpowiedzialni za projektowanie swoich aplikacji w ten sposób, aby zmaksymalizować użycie tych limitowanych i drogich zasobów.

wstęp

Jedną z najbardziej efektywnych technik optymalizacji jest użycie **profilera** do wykrycia wąskich gardeł programu.

Po identyfikacji wąskiego gardła, np. pętli wykonywanej tysiące razy, najlepszą rzeczą jaką możemy zrobić jest takie przerobienie programu aby nie wykonywać jej tysiące razy.

To da nam dużo lepszy efekt niż przyspieszenie pętli o 10% (co często może zrobić sam kompilator).

Optymalizacja jest czysta stratą czasu jeżeli którekolwiek z założeń jest prawdziwe:

- fragmenty programu nie zostały jeszcze napisane

- program nie został w pełni przetestowany i przedebugowany

- działa wystarczająco szybko

Optymalizacja związana z pamięcią

Adresowanie kolumn

Indeksując dane w tablicach wielowymiarowych, pamiętajmy o zwiększaniu indeksu najbardziej wysuniętego z prawej strony najpierw.

Anty-przykład:

```
float array[20][100];
int i, j;

for (j = 0; j < 100; j++)
  for (i = 0; i < 20; i++)
    array[i][j] = 0.0;
```

Optymalizacja związana z pamięcią

Nie kopiuj dużych rzeczy

Zamiast kopiować text, tablice, duże struktury, możemy skopiować wskaźniki do nich. Tak długo jak używamy wskaźników przed modyfikacją właściwych struktur, możemy to robić.

ANSI C wymaga aby struktury były przekazywane przez wartość tak jak wszystkie inne typy proste, więc przy bardzo dużych strukturach, lub przy funkcjach wywoływanych miliony razy na średnich strukturach, możemy przekazać adres struktury, zamiast kopii struktury.

```
double Len(Vector3D v);

double Len(const Vector3D &v);
```

Optymalizacja związana z pamięcią

Dziel lub łącz tablice

Jeżeli części twojego programu które charakteryzują się częstym dostępem do pamięci, robią to poprzez dostęp do elementów w „równoległych tablicach”, można je połączyć w tablicę struktur. Dzięki temu dane pod zadanym adresem są trzymane wspólnie w pamięci.

Jeżeli mamy tablice struktur, a krytyczna część programu używa tylko pojedynczych pól z każdej struktury, możemy podzielić te pola w rozłączne tablice tak aby nie używane pola nie były czytane niepotrzebnie do cache.

```
Vector3D normal[100];
Vector3D v1[100];
Vector3D v2[100];
Vector3D v3[100];
```

lub:

```
struct Triangle
{
    Vector3D normal,v1,v2,v3;
};
Triangle tris[100];
```

Optymalizacja związana z pamięcią

Zredukuj PADDING

W systemach z restrykcjami wyrównania możemy czasami zaoszczędzić trochę miejsca poprzez przeorganizowanie pól o podobnej wielkości sortując je (malejąco względem wielkości).

Po takiej operacji może nadal zostać trochę wypełnienia (padding), ale usunięcie go mogło by zniszczyć cały zysk wynikający z wyrównywania.

ANSI C często zapewnia nam wewnętrzne wypełnianie struktur,ale ręczne ustawienie jak w kodzie poniżej gwarantuje nam najmniejsze straty nawet na najprostszych RISC'owych maszynach.

Często używamy typów char i short do przechowywania flag (prawda lub fałsz). Możemy połączyć kilka z takich flag w jednym bajcie, i używać operatora & (and), do czytania wartości, oraz | (lub) i << (przesunięcie bitowe) do ich ustawiania.

```
/* sizeof = 64 bytes */
struct foo {
  float    a;
  double   b;
  float    c;
  double   d;
  short    e;
  long     f;
  short    g;
  long     h;
  char     i;
  int      j;
  char     k;
  int      l;
};
```

```
/* sizeof = 48 bytes */
struct foo {
  double   b;
  double   d;
  long     f;
  long     h;
  float    a;
  float    c;
  int      j;
  int      l;
  short    e;
  short    g;
  char     i;
  char     k;
};
```

Optymalizacja związana z pamięcią

Zwiększ PADDING

Paradoksalnie, zwiększenia wielkości wyrównania danych w strukturze tak aby pasowała (lub była wielokrotnością) wielkości linii cachu może zwiększyć wydajność.

Uzasadnienie jest proste, jeżeli wielkość struktury danych jest niedostosowana do wielkości linii cachu, może nachodzić na dwie linie cachu podwajając czas potrzebny do odczytania jej z pamięci.

Wielkość struktur danych może być łatwo zwiększona poprzez dodawanie nieużywanych pól na jej końcu,najczęściej tablicy bajtów.

Wyrównywanie głównego adresu jest trudniejsze do kontrolowania, najczęściej jedna z tych technik działa:

- Użyj malloc'a zamiast statycznej tablicy.
- Niektóre malloc'i automatycznie alokują pamięć z uwzględnieniem wyrównania do linii cach'u (niektóre nie)
- Zaalokuj blok większy niż potrzebujesz (weź pierwszy wskaźnik z prawidłowym wyrównaniem)
- Użyj alternatywnego alokatora (np. memalign) który gwarantuje poprawne wyrównanie.

Optymalizacja związana z pamięcią

Wycieki pamięci

malloc i free czasami się ciekawie zachowują, i często są kompletnie inaczej implementowane na różnych maszynach.

Kilka malloc'ow jest "ztuningowanych" na wydajność (np. mallopt).

W niektórych aplikacjach, malloc jest bardzo szybki, a free powolny, wieć jeśli nasz program używa trochę pamięci ale tak naprawde nie używa jej ponownie przed zamknięciem, możemy przyspieszyć program poprzez jawne usunięcie zwalniania pamięci:

```
#define free(x)    /* no-op */
```

umieszczamy to w jakimś nagłówku linkowanym w całym programie.

Chcę podkreślić, że to bardzo niebezpieczne narzędzie...

Programy o długim czasie życia (np. demony działające w tle) i programy używające sporej ilości pamięci powinny ostrożnie zwalniać pamięć w momencie kiedy jej już nie potrzebują.

Optymalizacja związana z pamięcią

Adresowanie kolumn

Indeksując dane w tablicach wielowymiarowych, pamiętajmy o zwiększaniu najbardziej wysuniętego indexu z prawej strony najpierw.

Anty-przykład:

```
float array[20][100];
int i, j;

for (j = 0; j < 100; j++)
  for (i = 0; i < 20; i++)
    array[i][j] = 0.0;
```

Pułapki

1. Programiści mają tendencję do przeceniania swoich programów. Wartość wynikającą z optymalizacji można przybliżyć wzorem:

   ```
   ilość_uruchomień x ilość_użytkowników x oszczędność_czasu x płaca_użytkownika
   - czas spędzony na optymalizacji x płaca programisty
   ```

Pułapki

1. Programiści mają tendencję do przeceniania swoich programów. Wartość wynikającą z optymalizacji można przybliżyć wzorem:

   `ilość_uruchomień x ilość_użytkowników x oszczędność_czasu x płaca_użytkownika`

   `- czas spędzony na optymalizacji x płaca programisty`

Pułapki

1. Programiści mają tendencję do przeceniania swoich programów. Wartość wynikającą z optymalizacji można przybliżyć wzorem:

   ```
   ilość_uruchomień x ilość_użytkowników x oszczędność_czasu x płaca_użytkownika
   - czas spędzony na optymalizacji x płaca programisty
   ```

2. Wiele z optymalizacji o których wspomnieliśmy może być wykonywana automatycznie przez kompilator!

Pułapki

1. Programiści mają tendencję do przeceniania swoich programów. Wartość wynikającą z optymalizacji można przybliżyć wzorem:

```
ilość_uruchomień x ilość_użytkowników x oszczędność_czasu x płaca_użytkownika
- czas spędzony na optymalizacji x płaca programisty
```

2. Wiele z optymalizacji o których wspomnieliśmy może być wykonywana automatycznie przez kompilator!

3. Nie nabierzcie nawyku pisania całego kodu zgodnie z powyższymi zasadami optymalizacji.

Używajcie ich tylko do funkcji będących czasowym wąskim gardłem wykonania programu.

Pułapki

1. Programiści mają tendencję do przeceniania swoich programów. Wartość wynikającą z optymalizacji można przybliżyć wzorem:

   ```
   ilość_uruchomień x ilość_użytkowników x oszczędność_czasu x płaca_użytkownika
   - czas spędzony na optymalizacji x płaca programisty
   ```

2. Wiele z optymalizacji o których wspomnieliśmy może być wykonywana automatycznie przez kompilator!

3. Nie nabierzcie nawyku pisania całego kodu zgodnie z powyższymi zasadami optymalizacji.

   Używajcie ich tylko do funkcji będących czasowym wąskim gardłem wykonania programu.

4. Tydzień spędzony przez programistę na optymalizacji, może kosztować tysiące dolarów, czasami łatwiej zainwestować te pieniądze w szybszy procesor, więcej pamięci lub szybszy dysk.

Pułapki

1. Programiści mają tendencję do przeceniania swoich programów. Wartość wynikającą z optymalizacji można przybliżyć wzorem:

   ```
   ilość_uruchomień x ilość_użytkowników x oszczędność_czasu x płaca_użytkownika
   - czas spędzony na optymalizacji x płaca programisty
   ```

2. Wiele z optymalizacji o których wspomnieliśmy może być wykonywana automatycznie przez kompilator!

3. Nie nabierzcie nawyku pisania całego kodu zgodnie z powyższymi zasadami optymalizacji.

   Używajcie ich tylko do funkcji będących czasowym wąskim gardłem wykonania programu.

4. Tydzień spędzony przez programistę na optymalizacji, może kosztować tysiące dolarów, czasami łatwiej zainwestować te pieniądze w szybszy procesor, więcej pamięci lub szybszy dysk.

5. Bardzo początkujący programiści, czasami myślą, że pisanie wielu instrukcji w jednej linii, oraz usuwanie spacji i tabulatorów przyspieszy kod. O ile taka technika może pomóc w przypadku języków interpretowanych, w C nie zadziała.